Artix™

Designing Artix Solutions

Version 2.1, July 2004

Making Software Work Together™

Updated: 25-Mar-2005

M 3 2 0 8

Contents

List of Figures

List of Tables

Preface

What is Covered in this Book

Designing Artix Solutions outlines how to design, develop, and deploy integration solutions with Artix using the graphical user interface (GUI), the Artix command line tools, or both. It also guides you through producing Web Services Description Language (WSDL), source code, and runtime configuration files for your Artix integration solution.

Who Should Read this Book

This guide is intended for all users of Artix. This guide assumes that you have a working knowledge of the middleware transports that are being used to implement the Artix system. It also assumes that you are familiar with basic software design concepts, and that you have a basic understanding of WSDL.

If you would like to know more about WSDL concepts, see the Introduction to WSDL in *Learning about Artix*.

How to Use this Book

If you are new to Artix

You may want to do one or more of the following:

- Learn about Artix - see "Introduction to Artix"
- Read a walkthrough of how to create a Web Service client, or server, or both - see "Use Case Examples" and "Command Line Use Case Examples"

If you find any terms you aren unfamiliar with, turn to the "Glossary" on page 529 for a list of Artix terms and definitions.

If you've worked with Artix before

You probably have a clear idea of what you want to use Artix to do. In this case, one of the following suggestions may help.

Using the Artix designer is discussed in the section "Using Artix Designer". It includes chapters on the following:

- If you are creating a new workspace, see "Creating an Artix Workspace"
- If you're creating or editing a collection, see "Working with Artix Collections"
- If you're creating or editing resources, see "Working with Artix Resources"
- If you're creating or editing a binding, see "Adding Bindings"
- If you're creating or editing a service, see "Adding Services"
- If you're creating or editing a route, see "Routing Messages"
- If you're ready to generate code, see "Deployment"

The second section in this guide, "Using Artix Command Line Tools", provides a detailed description of how to describe Artix endpoints using WSDL and the WSDL extenstions used by WSDL. The chapters in this section parallel the chapters in the first section of the guide.

In addition the following appendecies are included to provied reference material on using some the Artix bindings:

- "SOAP Binding Extensions" on page 427
- "CORBA Type Mapping" on page 441
- "WebSphere MQ Artix Extensions" on page 485
- "Tibco Transport Extensions" on page 521

If you are migrating from Artix 2.0 to Artix 2.1

Between Artix 2.0 and Artix 2.1, we made several changes to the Artix user interface, in an effort to make it more intuitive and easier to use. The changes you will notice include:

- There is now an additional Editor mode of the interface in which you can create and edit WSDL and Schema documents
- Schemas can now be created or imported into Artix as resources
- Support for two additional binding types - Fixed and Tagged - has been added to the Binding wizard

- The Resource Navigator diagram can now display relationships as well as groupings
- The Contract menu has been renamed the Resource menu
- In Deployer mode, seven new fast track templates have been added
- In Deployer mode, Deployment Bundles and Profiles are now listed on the Tree
- In Deployer mode, the Current View filter that was at the bottom of the Tree has been removed and the functionality has been placed into a new menu called View
- In Deployer mode, Configuration functionality has been added to the workspace details panel

Finding Your Way Around the Library

The Artix library contains several books that provide assistance for any of the tasks you are trying to perform. The Artix library is listed here, with a short description of each book.

If you're new to Artix

You may be interested in reading:

- *Release Notes* - contain release-specific information about Artix.
- *Learning about Artix* - this book describes basic Artix and WSDL concepts. It also guides you through programming Artix applications against all of the supported transports.

  This book is a combination of two books from the Artix 2.0 library - the Getting Started Guide and the Tutorial.

To design and develop Artix solutions

You should read one or more of the following:

- *Designing Artix Solutions* - (this book) - provides detailed information about designing Artix solutions, either from the command line or by using the Artix Designer. Also includes use case examples for both.
- *Developing Artix Applications in C++* - this book discusses the technical aspects of programming applications using the C++ API.
- *Developing Artix Applications in Java* - this book discusses the technical aspects of programming applications using the Java API.
- *Command Line Reference* - this book contains reference information about the Artix command line tools.

To manage and configure your Artix solution

You should read one or more of the following:

- *Deploying and Managing Artix Solutions* - describes how to deploy Artix-enabled systems, and provides detailed examples for a number of typical use cases.
- *IONA Tivoli Integration Guide* - explains how to integrate Artix with IBM Tivoli.
- *IONA BMC Patrol Integration Guide* - explains how to integrate Artix with BMC Patrol.
- *Artix Security Guide* - provides detailed information about using the security features of Artix.

Have you got the latest version?

The latest updates to the Artix documentation can be found at http://www.iona.com/support/docs. Compare the version details provided there with the last updated date printed on the inside cover of the book you are using (under the copyright notice).

Artix online help

While using the Artix Designer you can access contextual online help, providing:

- A description of your current Artix Designer screen
- Detailed step-by-step instructions on how to perform tasks from this screen
- A comprehensive index and glossary
- A full search feature

There are two ways that you can access the Online Help:

- Click the Help button on the Artix Designer panel, or
- Select **Contents** from the Help menu

Additional Resources for Help

The IONA knowledge base 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 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

This book uses the following typographical and keying conventions

Typographical Convention

`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

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.

Part I

Using Artix Designer

In this part

This part contains the following chapters:

CHAPTER 1

Introduction to Artix

You can use Artix to design, develop, and deploy integration solutions that are middleware-neutral.

In this chapter

This chapter discusses the following topics:

Overview

Artix is a flexible and easy-to-use tool for integrating your existing applications across a number of different middleware platforms. Artix also makes it easy to expose your existing applications as Web services or as a service for any number of applications using other middleware transports. In addition, Artix provides a flexible programming model that allows you to create new applications that can communicate using any of the protocols that Artix supports.

Despite the flexibility and power of Artix, designing solutions using Artix is a straightforward process which requires a minimum of coding.

The Artix user interface, *the Artix Designer*, has two modes of operation. The first is a pure WSDL or schema creation and edit mode, and the second, referred to as "deployer mode" provides the ability to create and deploy Artix contracts. This isn't to say that the two modes are mutually exclusive; the only real functional difference is that the Editor mode contains no deployment capability. Appearance-wise, they are identical other than that the Editor mode does not contain the Designer tree.

Artix Designer - Deployer mode

The deployer mode of the Artix Designer provides a full suite of wizards to guide you through the modeling of your systems, the generation of Artix components, and the deployment of your system. Artix also ships with a number of command line tools that can be used to generate Artix components. For more information about working with the Artix command line tools, see part two of this book.

When you start working with in workspace mode, you will see the following components listed in the Designer Tree:

- Workspace
- Workspace Services
- Deployment Profiles
- Shared Resource
- Collection
- Deployment Bundles
- Resources

Each component is documented in detail throughout this book, but following is a brief description of what they are and how they relate to each other.

Workspace

The Workspace defines your Artix solution. It contains collections and resources, and all the required deployment information to build your solution.

Workspace Services

to come

Deployment Profiles

The Deployment Profile defines machine level-information such as the Artix save location, the compiler location, and the operating system being used. This profile can be used multiple times as it is not specific to any particular collection defined within the workspace.

Shared Resource

Shared Resources are WSDL contracts that are stored at a workspace level, and are included, by default, in every collection in that workspace. When creating a workspace, you are given the option of also creating shared resources, or else you can add them to the workspace later.

Resources that are not shared, and that exist only in one collection, are collection-specific. A collection-specific resource can be changed to a shared resource, and therefore added to all existing collections, if required.

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. A collection can also be deployed as a switch process.

Deployment Bundles

The Deployment Bundle defines the deployment characteristics for a collection, such as the deployment type (client, server, or switch), code generation options, and configuration details. You can also modify the service WSDL for each deployment bundle, if necessary.

Every deployment bundle is associated with one deployment profile. You can have as many deployment bundles as you like for every collection in your workspace, but you could quite easily get by with only one deployment profile.

Resources

Can be either WSDL contracts or schema files - each is explained here.

WSDL contracts define the interaction of an endpoint with the Artix bus. Contracts are written in WSDL. Following the procedure described by W3C, IONA has extended WSDL to support the bus' advanced functionality, and to use transports and formats other than HTTP and SOAP.

In Artix, contracts can be created from a variety of resources including:

- Existing WSDL files
- Existing IDL files
- WSDL URLs
- Existing Data Sets, such as a COBOL Copybook

A contract consists of two parts:

- **Logical** - defines the namespaces, messages, and operations that the collection exposes. This part of the contract is independent of the underlying transports and wire formats. It fully specifies the data structures and possible operation/interaction with the application interface. It is made up of the WSDL tags `<type>`, `<message>`, and `<portType>`.
- **Physical** - defines the transports, wire formats, and routing information used to deliver messages to and from collections, over the bus. This portion of the contract also defines which messages use each of the defined transports and bindings. The physical portion of the contract is made up of the standard WSDL tags `<binding>`, `<service>`, and `<route>`. It is also the portion of the contract that may contain IONA WSDL extensions.

For more information, see "What is a Contract?" on page 53.

Schemas define types. They can be standalone resources, or imported into a WSDL contract to define the types for that contract.

For more information, see "What is a Schema?" on page 56.

Artix Designer - Editor mode

The Editor mode of the Artix Designer is virtually the same as the Deployer mode, but for two main differences:

- There is no Designer Tree visible on the left of the details panel.
- There is no access to the deployment functionality offered in Deployer mode.

The Editor mode is a simplified version of the Designer. You do not need to create the workspace/collection/resource structure to work on your files, and there is therefore only a sub-set of the wizards and dialogs you would normally see in Deployer mode. The Editor mode has been created for those times when you simply want to work directly with WSDL or Schema files, giving you a tool in which you can create and edit them as required.

Switching between modes

You can switch from the Deployer mode to the Editor mode and vice versa, as long as you have a resource selected in the deployer mode.

To make the switch, either click on the icon in the tool bar, or select the mode you wish to switch to from the View menu.

When you switch from Editor mode to Deployer mode, a default workspace is created for you.

When you switch from Deployer mode to Editor mode, the Designer Tree is simply hidden to remove all references to workspaces, collections, and deployment entities.

---

5 easy steps

Regardless of the complexity of your Artix project or the tools you chose to develop it, there are five basic steps in developing a solution using Artix:

1. Create an Artix workspace to define the structure your proposed solution.
2. Create an Artix collection to manage the resources that define the Artix contract.
3. Create an Artix contract to describe how you intend to integrate or expose your systems.
4. Deploy the solution.
5. Develop any application level code needed to complete the solution.

Of course, if all you want to do in Artix is to work in Editor mode, then the list is even simpler:

1. Create a WSDL or Schema file or import an existing one.
2. Add to or change the file as required using the wizards and dialogs provided from the Resource menu.
3. Save your file.

Using Artix for the first time

Welcome dialog

The first time you start the Artix user interface, you will see the Welcome dialog, as shown in Figure 1.

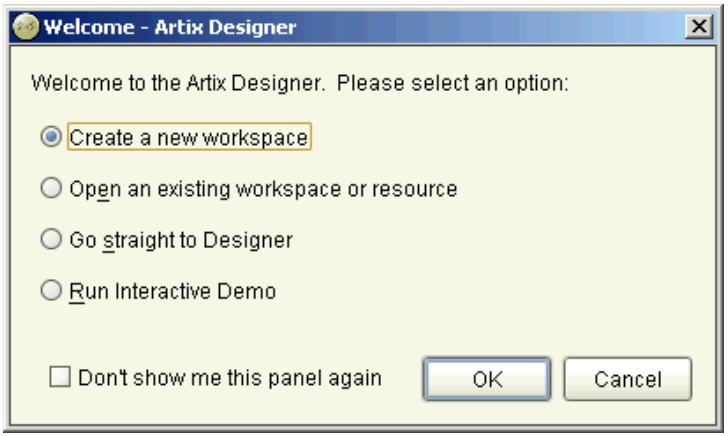

Figure 1: *Welcome dialog*

You have four options from this dialog:

- Create a new workspace - takes you to the New Workspace dialog, where you can select options for creating your Artix workspace.
- Open an existing workspace or resource - takes you to a file chooser dialog from where you can navigate to any previously created workspaces or any resource files (WSDL or Schema) you have stored. **Note:** Choosing a resource file will open the Designer in Editor mode.
- Go straight to the Designer - opens Artix without loading a workspace.
- Run interactive demo - launches an demo of the Designer (requires a plug-in which is available for download if necessary).

Tip: Click the check box at the bottom of the panel (Don't show me this panel again) to stop this panel displaying every time you start Artix. Instead, Artix will automatically load the last workspace accessed. To change it back, go to the Start-up options in the User Preferences dialog (**Edit** menu).

Working in Deployer Mode

Overview

The biggest difference between the two modes of the Artix Designer is that in the deployer mode you can deploy your Artix collections. The Editor mode allows you only to create or edit WSDL or Schema documents - it has no deployment functionality. If this is how you want to use the Designer, turn to "Working in Editor Mode" on page 10.

To be able to use the Artix deployment feature, you need to structure your Artix solution in a certain way - this is where workspaces come in.

Creating a workspace

The Artix workspace defines the structure of your proposed solution, and determines what is contained in the Artix Designer Tree.

There are two ways to create a workspace:

- Follow the New Workspace wizard (from the New Workspace dialog, as shown in Figure 2) to guide you through the process - recommended for first-time users of Artix.
- Select one of the Workspace Templates provided in the New Workspace dialog to create one of the common workspaces.

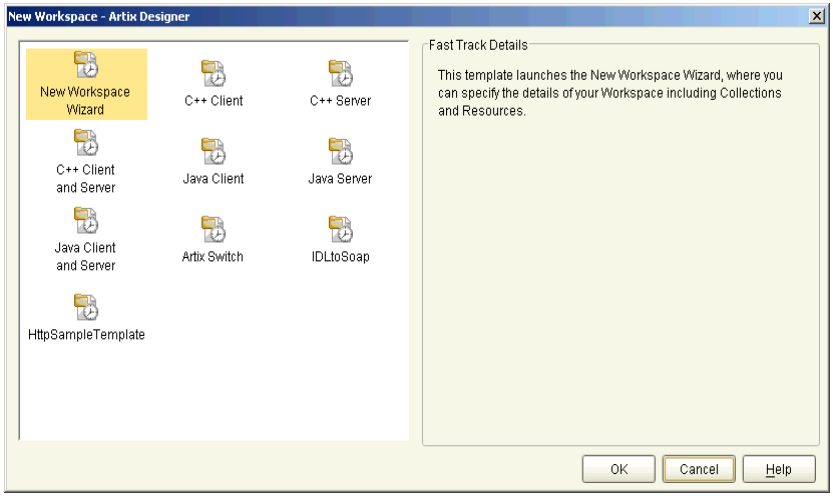

Figure 2: *New Workspace dialog*

Tip: To access the New Workspace dialog, select **File | New | Workspace** from the menu bar, or click the New Workspace icon in the toolbar.

Shared resources

When you have created your workspace, you have the option of adding resources at the workspace level that can be applied to every collection contained in your workspace. In Artix, these are called "Shared Resources".

For more information on workspaces and shared resources, see "Creating an Artix Workspace" on page 19.

Adding collections and resources

A *collection* is a group of *resources* that can be deployed one or more times to meet your solution requirements. As such, it defines the Artix *contract*. This contract models the services you want to integrate.

While you can only have one workspace at a time, you can have as many collections as you like. They can comprise shared resources, collection-specific resources, or a mix of both.

When you create a workspace using the New Workspace wizard you are given the opportunity to create a collection, but you can also add collections to workspaces by selecting **File | New | Collection** from the menu bar.

For more information about Artix collections, see "Working with Artix Collections" on page 35.

Collection-specific resources

After you have created a collection you can add *collection-specific resources.* Resources can be created from existing WSDL files, or from WSDL generated from IDL files. They can also include contracts generated from data sets such as COBOL Copybooks, or contracts created from scratch using the wizards and dialogs provided by the Resource Editor.

Regardless of your mix of resources, the process of creating the Artix contract involves creating logical descriptions of the data and the operations you want the services to share, and mapping them to the physical payload formats and transports used by the services to expose themselves to the network. Artix uses the industry standard Web Services Description Language (WSDL) to model services.

For more information about resources, see "Working with Artix Resources" on page 47.

Generating code for your solution

Generating code with the Artix Designer is a three-step process:

- Create the **Deployment Profile** to define machine-level information that you can use for one or more of your solutions
- Create the **Deployment Bundle** to define the characteristics of the collection you are deploying, including the type of deployment (client, server, or switch), configuration information, and environment scripts
- Generate the code - once your deployment profile and bundle are in place, actual deployment is performed using the **Generate Code** dialog

For more information, see "Deployment" on page 177.

For a detailed discussion of Artix configuration, see *Deploying and Managing Artix Solutions*.

For a detailed description of generating Artix stubs and skeletons, see *Developing Artix Applications with C++*.

Developing additional code

Unless your services share identical interfaces, you will need to develop some additional application code. Artix can only map between services that share a common interface.

Typically, you can make the required changes to only one side of the services you are integrating and you can write the application code using a familiar programming paradigm. For example, if you are a CORBA developer integrating a CORBA system with a Tuxedo application, Artix will generate the IDL representing the interface used in the service integration. You can then implement the interface using CORBA.

If you are developing new applications using Artix, you will have to write the application logic from scratch using the stubs and skeletons generated by Artix. For a detailed discussion of developing applications using Artix, see one of the following:

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

Working in Editor Mode

Overview

The Artix Designer in Editor mode is a powerful XML editor. You can create and edit WSDL documents, and you can also open and edit Schema.

The full graphic representation of the WSDL model provided by the Artix Designer in Deployer mode is still available to you in Editor mode, but only when you are working with valid WSDL. If the WSDL is invalid, it can't be modeled graphically and can only be viewed as XML text, as is also the case in Deployer mode.

All of the Resource Editing wizards and dialogs are available to you in Editor mode, making it easy for you to create your WSDL or Schemas without having to write them in XML from scratch.

Working with WSDL

If you've used the Artix Designer before, the first thing you'll notice about the Editor view is that there is no Designer Tree on the left of the details panel. Assuming you have valid WSDL however, the graphical view of the WSDL file will be as you remember it from the Deployer view.

You can create new WSDL or XSD documents, or open existing ones, by selecting **using the File** menu.

When you have a WSDL model displayed in the Resource Navigator, as shown in Figure 3, you can add and edit the components via options from either the Resource menu, or from the contextual menus accessed by right-clicking on any of the component names.

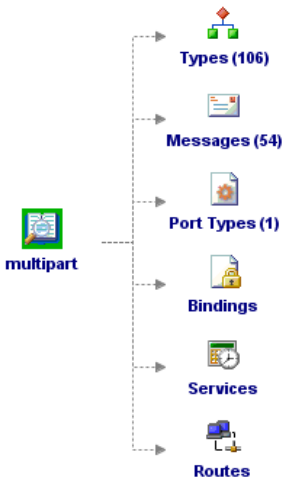

Figure 3: *Resource Navigator displaying WSDL model*

For more information about working with WSDL, see "Creating New Resources" on page 57.

Working with Schemas

Working with Schemas is similar to working with WSDL, except that it only defines the types while WSDL defines all of the contract components. You can create a schema to be a stand-alone resource or to be the type definition of a larger piece of WSDL - an import statement in the WSDL will simply refer to the external schema. In this way, WSDL and Schema resources can together define Artix contracts.

You can open existing Schemas by selecting **File | Open | Resource** and navigating to the file. For more information about adding Schemas, see "Creating an XSD Schema" on page 105.

Using this guide while in Editor mode

If you are using the Designer only in Editor mode, only a subset of the chapters in this book will be of interest to you. They are:

- "Working with Artix Resources" on page 47
- "Adding Bindings" on page 115
- "Adding Services" on page 139
- "Routing Messages" on page 167

Setting user preferences

Overview

The Artix User Preferences dialog enables you to define the way Artix looks and behaves. For example, you can use this dialog to set:

- The look and feel of the interface
- What is displayed first every time you start Artix
- A default location for workspaces and resources

Setting your User Preferences

To access the User Preferences dialog:

1. Select **Edit | User Preferences** to display the User Preferences dialog, as shown in Figure 4 .

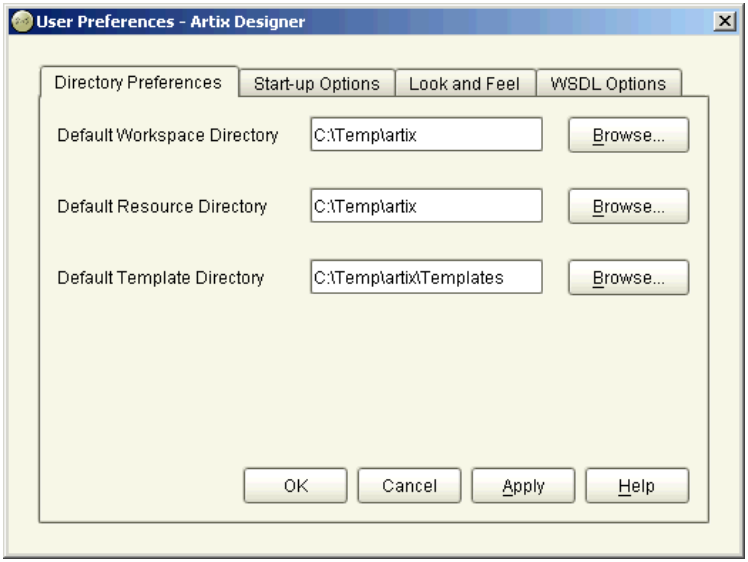

Figure 4: *User Preferences dialog—Directory Preferences panel*

2. Use the **Directory Preferences** panel to:
   - Set a default workspace directory - the directory to open when browsing for existing workspaces

- Set a default resource directory - the directory to open when browsing for existing resources
- Set a default templates directory - the directory where any custom workspace templates are stored

3. Use the **Start-up Options** panel to define whether to:
   - Set your default user interface mode to Editor
   - Display the Start-up dialog every time you open Artix
   - Include workspace history in your File menu
   - Nominate a number of history files to include in the File menu
   - Display the Diagram or Text view as your default
4. Use the **Look and Feel** panel to select an appearance for the Designer from the list provided.
5. Use the **WSDL Options** panel to define a default namespace for your WSDL contracts.
6. Click **OK** when you have finished making your changes to close this dialog and return to the Artix Designer.

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

Elements of a WSDL document

A WSDL document is made up of the following elements, which you will see represented throughout the Artix Designer. You can use the Designer to create and edit these 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
- `<message>` – the abstract definition of the data being communicated
- `<operation>`– the abstract description of an action
- `<portType>` – the set of operations representing an absract endpoint
- `<binding>` – the concrete data format specification for a port type
- `<port>` – the endpoint defined by a binding and a physical address
- `<service>` – a set of ports

Example WSDL file

On the following pages is an example of a WSDL file. It is the HelloWorld WSDL used in many of the demos shipped with the Artix product

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="HelloWorld"
    targetNamespace="http://www.iona.com/hello_world_soap_http"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:http-conf="http://schemas.iona.com/transports/http/
     configuration"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/hello_world_soap_http"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Note the types, messages, port types, and bindings defined in this section.

```
<types>
        <schema
   targetNamespace="http://www.iona.com/hello_world_soap_http"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <element name="responseType" type="xsd:string"/>
            <element name="requestType" type="xsd:string"/>
        </schema>
    </types>
 <message name="sayHiRequest"/>
    <message name="sayHiResponse">
        <part element="tns:responseType" name="theResponse"/>
    </message>
    <message name="greetMeRequest">
        <part element="tns:requestType" name="me"/>
    </message>
    <message name="greetMeResponse">
        <part element="tns:responseType" name="theResponse"/>
    </message>
 <portType name="Greeter">
    <operation name="sayHi">
        <input message="tns:sayHiRequest" name="sayHiRequest"/>
        <output message="tns:sayHiResponse"
           name="sayHiResponse"/>
    </operation>
    <operation name="greetMe">
       <input message="tns:greetMeRequest"
           name="greetMeRequest"/>
       <output message="tns:greetMeResponse"
           name="greetMeResponse"/>
    </operation>
 </portType>
 <binding name="Greeter_SOAPBinding" type="tns:Greeter">
    <soap:binding style="document"
        transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sayHi">
        <soap:operation soapAction="" style="document"/>
        <input name="sayHiRequest">
            <soap:body use="literal"/>
        </input>
        <output name="sayHiResponse">
            <soap:body use="literal"/>
        </output>
    </operation>
```

Note the service defined in this section.

```
      <operation name="greetMe">
          <soap:operation soapAction="" style="document"/>
          <input name="greetMeRequest">
              <soap:body use="literal"/>
          </input>
          <output name="greetMeResponse">
              <soap:body use="literal"/>
          </output>
      </operation>
  </binding>
  <service name="SOAPService">
      <port binding="tns:Greeter_SOAPBinding" name="SoapPort">
          <soap:address location="http://localhost:9000"/>
          <http-conf:client/>
          <http-conf:server/>
      </port>
  </service>
</definitions>
```

For more information

For a more extensive WSDL discussion, see *Learning About Artix.*

CHAPTER 2

Creating an Artix Workspace

The Artix Designer provides a canvas within which you can design Artix solutions. The packaging mechanism for these solutions is the workspace.

In this chapter

This chapter discusses the following topics:

What is a Workspace?

Overview

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

Workspaces contain *collections* and *resources*. While you can only have one workspace open at a time, you can have as many collections and resources within that workspace as you like.

A collection is a group of resources that can be deployed as one or more systems, for example a client, a server, or a switch. A resource is a WSDL file that, either by itself or with other resources, defines the Artix Contract.

Resources can be stored at the workspace level and applied to one or more collections (shared resources), or can be stored at the collection level and apply only to that collection. The Workspace Details panel shows you the contents of your workspace, as shown in Figure 5.

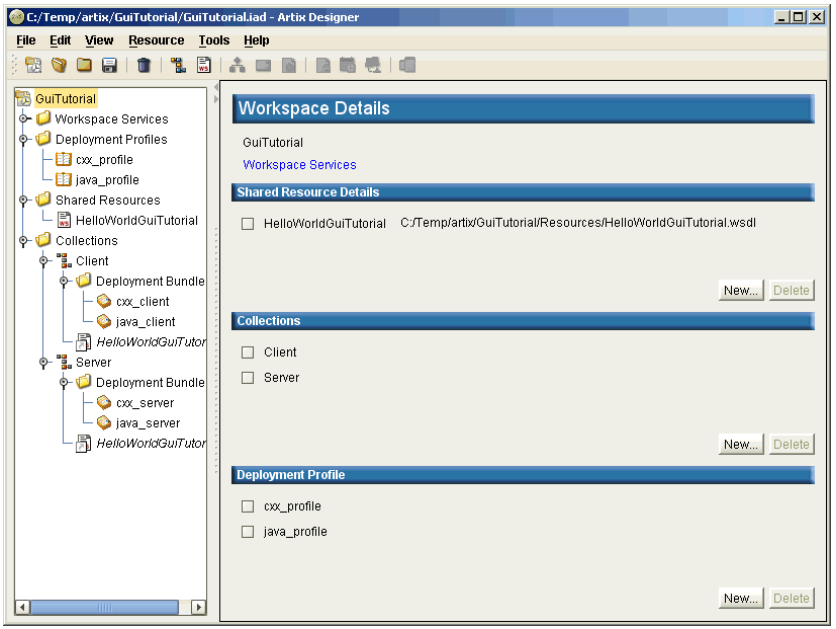

Figure 5: *Workspace Details panel*

The Workspace Details panel provides another view of your workspace, besides the Designer Tree view. It lists the collections and resources contained in your workspace, and also provides Add and Delete functions.

Deployment Profiles for the workspace are also listed on the Workspace Details panel. A profile is needed for each different operating system that will host your Artix deployment, such as Windows or Unix.

For more information about Artix Deployment, including Deployment Profiles, see "Deployment Explained" on page 178.

Creating a workspace

There are two ways to create a new workspace:

- Select the New Workspace wizard (from the New Workspace dialog, as shown in Figure 6) - recommended for first time users of Artix. - see page 26 for more information.
- Select one of the Workspace Templates provided in the New Workspace dialog to have Artix assume all of the necessary defaults to get your workspace up and running quickly. See page 30 for details.

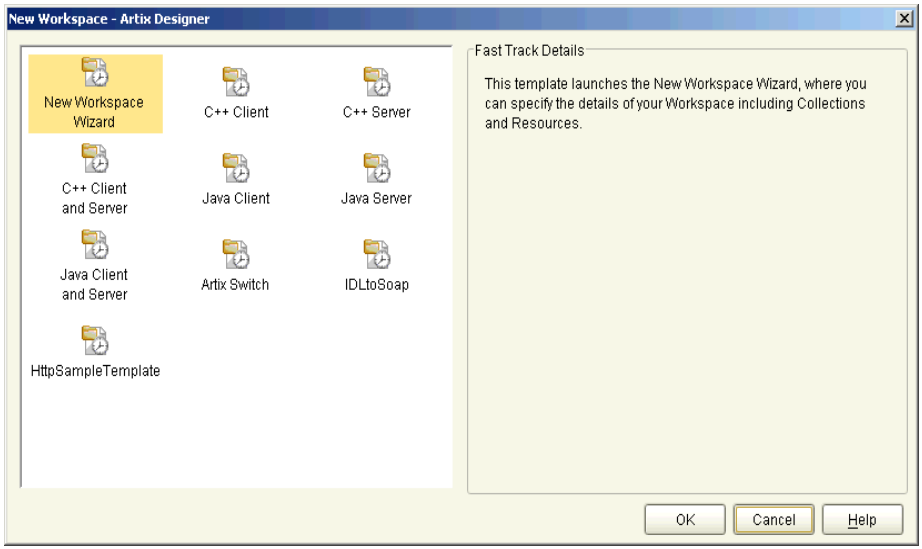

Figure 6: *New Workspace dialog*

Shared resources

Shared resources are resources that are stored at the workspace level, and by default are included in every collection you create within the workspace. All instances of this resource are linked, however - they are not individual *copies* of the resource. Thus, if you edit a shared resource, you are actually editing every instance of that resource.

When you create a new workspace, you are given the option of adding shared resources via the Shared Resources panel, as shown in Figure 7.

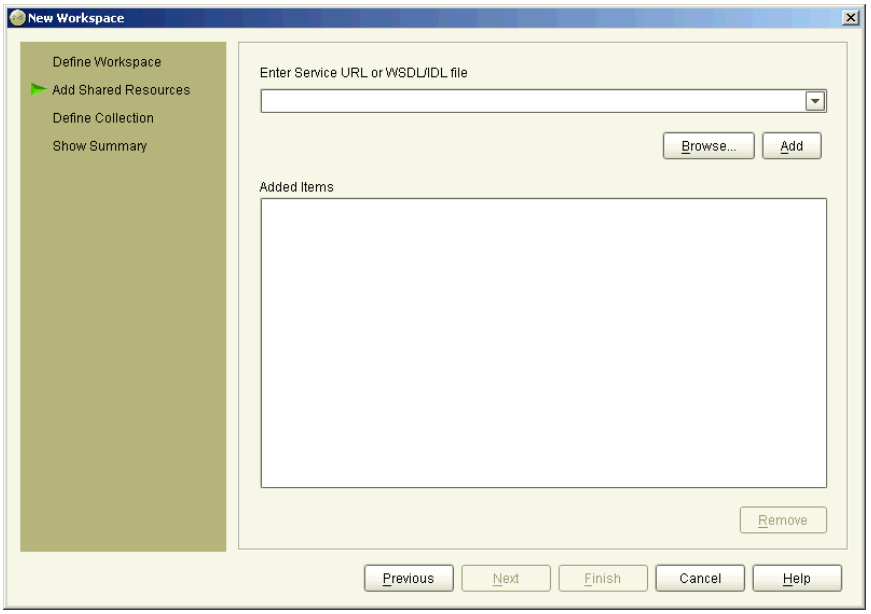

Figure 7: *New Workspace wizard—Shared Resources panel*

You can also add shared resources at other times from the Workspace Details panel by clicking the **Add** button under the Shared Resources Details list.

A shared resource is represented in two ways in the Designer Tree. The *original* version of the resource is listed under the Shared Resources folder, and the *reference* to the Shared Resource in each Collection is shown with the name of the resource italicized, and its icon having a dimmed shortcut

arrow, as shown in Figure 8. An **X** icon next to the resource name indicates invalid WSDL. The error information provided in the WSDL Text view of the contract will explain how to fix invalid WSDL.

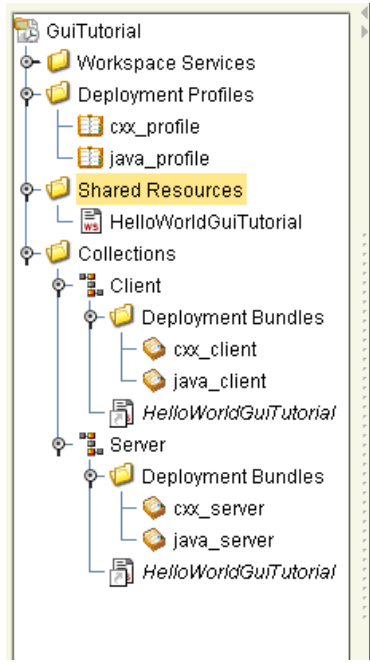

Figure 8: *Designer Tree Showing Collections and Shared Resources*

Collection-specific resources

In contrast, you can also create resources that are collection-specific. That is, they apply only to the collection within which they are created and are not also included in other collections within the workspace. These are indicated in Figure 8 by the resource names within collections that are not italicized, and are created by selecting a the Collection name and selecting **File | New | Resource**. See "Working with Artix Resources" on page 47.

Collections

A collection is a group of related resources within your workspace. It can be deployed as one or more systems, such as a client, a server, a switch, or any combination of all three.

When you create a workspace, you are given the opportunity to create a collection via the Define Collection panel, as shown in Figure 9.

Otherwise you can add them to your workspace from the Workspace Details panel by clicking on the **Add** button under the Collections list.

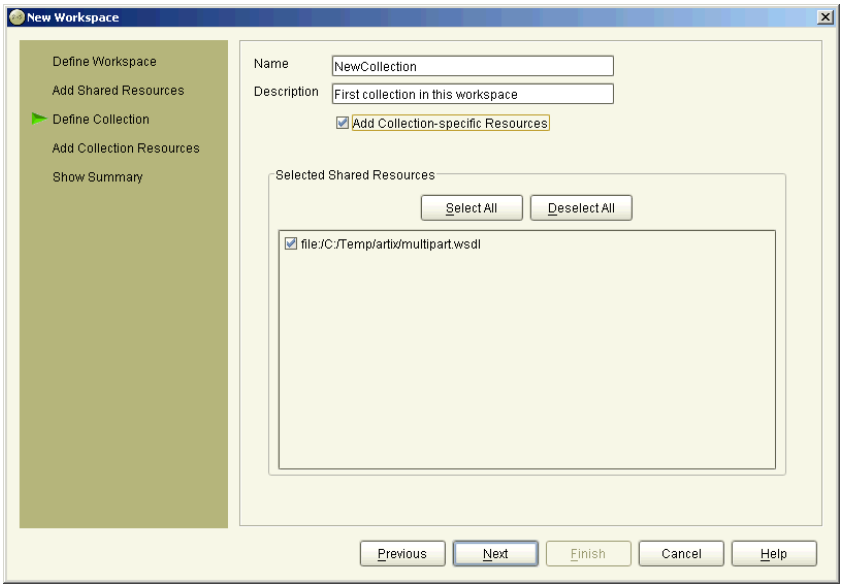

Figure 9: *New Workspace wizard—Collection panel*

Collections contain resources that together define the Artix contract. These resources can be based on one or more items, including URLs, and WSDL or IDL files. If an IDL file is added to a collection, it is converted to WSDL and this WSDL is what is actually listed on the Designer Tree.

For more information see "Working with Artix Collections" on page 35.

Deployment entities

There are two deployment entities in Artix that you need to be aware of when working in Workspace mode:

- Deployment Profiles, which are stored at the workspace level and apply to all collections in that workspace
- Deployment Bundles, which are stored at the collection level and apply to only that collection. A Deployment Bundle must also be associated with a Deployment Profile, meaning that you cannot create a bundle before creating a profile.

Creating a Workspace using a Wizard

To add a workspace using the New Workspace wizard:

1. From the New Workspace dialog, select the **New Workspace wizard** icon to display the New Workspace wizard, as shown in Figure 10.

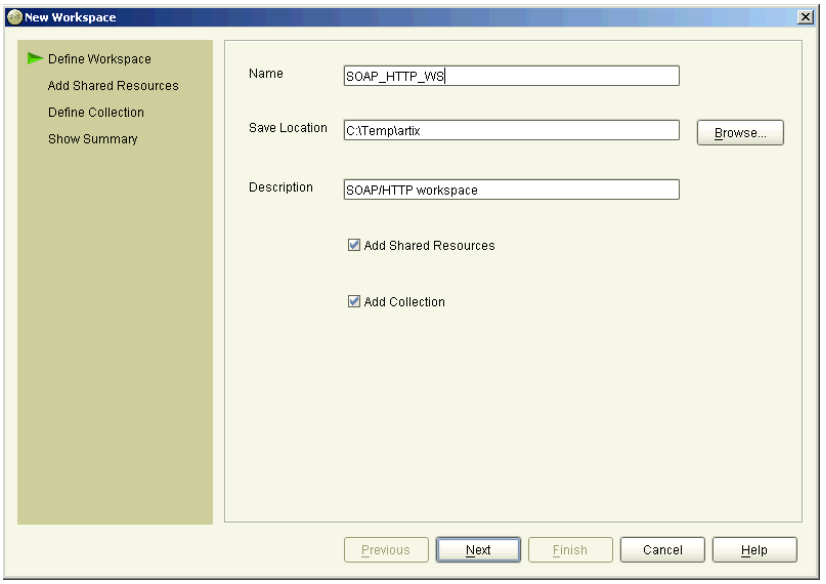

Figure 10: *New Workspace Wizard*

2. Enter a name for the workspace, or accept the default provided.
3. Select the location where you would like to save your workspace, or accept the default provided.

   Tip: To define a new default save location for all future workspaces, go to the User Preferences dialog (under the **Edit** menu).
4. Add a description for this workspace in the field provided.
5. Select the **Add Shared Resources** check box if you want to add resources to this workspace that will be shared between all the collections in the workspace. This step is optional.

Selecting this option will add an extra panel to the wizard for you to enter the shared resource details.

6. Select the **Add Collection** check box if you want to add a collection to this workspace now. Note that this is optional - you can always add a collection later if you don't want to add one now.

   Selecting this option will add an extra panel to the wizard for you to enter the collection details.

7. Click **Next** to display one of the following panels, depending on which check boxes you selected on the first panel:

   - If you checked the Add Shared Resources option, the Shared Resources panel is displayed, as shown in Figure 11. Continue with **step 8.**

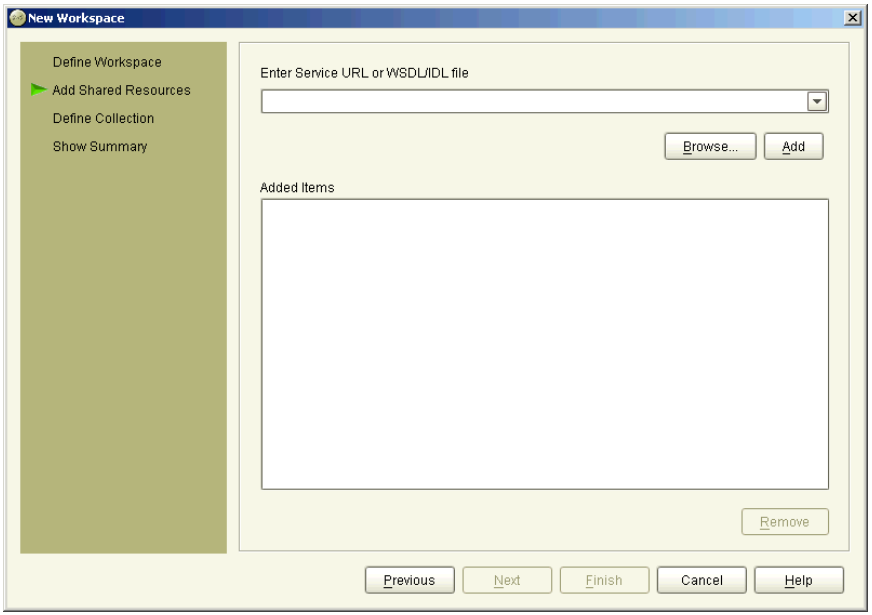

Figure 11: *New Workspace wizard—Shared Resources panel*

- If you did not check the Add Resources option but did check the Add Collection option, the Define Collection panel is displayed, as shown in Figure 12. Continue with **step 10**.
- If you did not check either of the options on the first panel, the Summary panel is displayed as shown in Figure 13 on page 29. Continue with **step 14**.

8. Type the location of either a WSDL file or an IDL file in the Enter Service URL or WSDL/IDL file field, or click **Browse** to navigate to the file you would like to use.

   When you have selected a file to use, click **Add** to list it in the Added Items list.

9. Repeat **step 8** as many times as you like to continue adding resources to the list, then click **Next** to display Define Collection panel as shown in Figure 12. If you did not choose to Add a Collection, go to **step 14**.

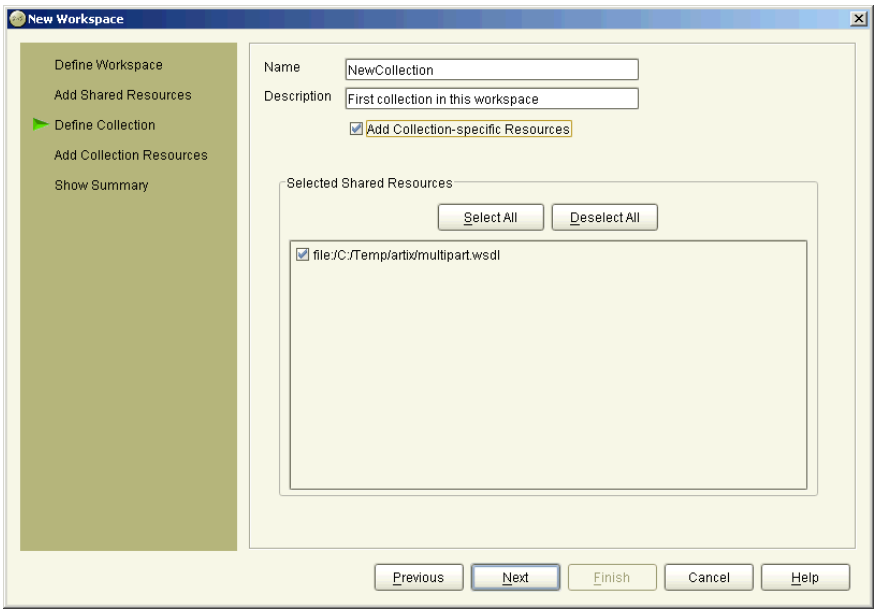

Figure 12: *New Workspace wizard—Collection panel*

10. Enter a name for the new collection, or accept the default provided.
11. Enter a description for the new collection in the Description field.
12. By default, all shared resources you added to this workspace on the previous panel are selected to be added to this collection. If there are any resources you do not want added, click on their check box to deselect them.
13. Click **Next** to display the Summary panel, as shown in Figure 13. This panel lists everything you just specified in the wizard.

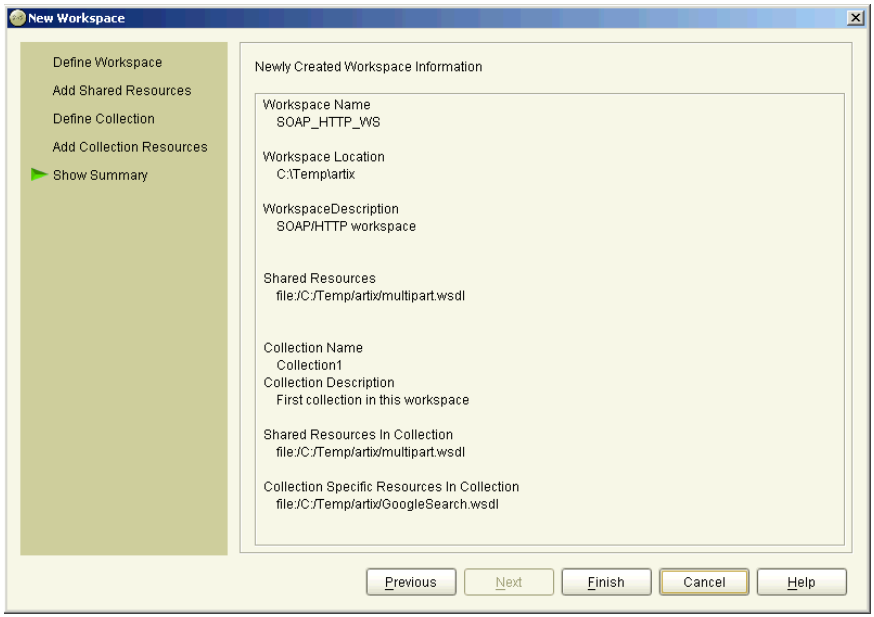

Figure 13: *New Workspace wizard—Summary panel.*

14. Click **Finish** to close the wizard and display the Artix Designer, where the Designer Tree displays your newly created workspace.

Creating a Workspace using a Template

To add a workspace using a template:

1. From the New Workspace dialog, select one of the templates listed to create a workspace for that type of Artix deployment. As shown in Figure 14, the workspace templates provided are:
   - C++ Client
   - C++ Server
   - C++ Server and Client
   - Java Client
   - Java Server
   - Java Client and Server
   - Artix Switch
   - IDL to SOAP
   - HttpSample Template (editable as a custom template) - for specific instructions on using this custom template, see "Working with Custom Templates" on page 32.

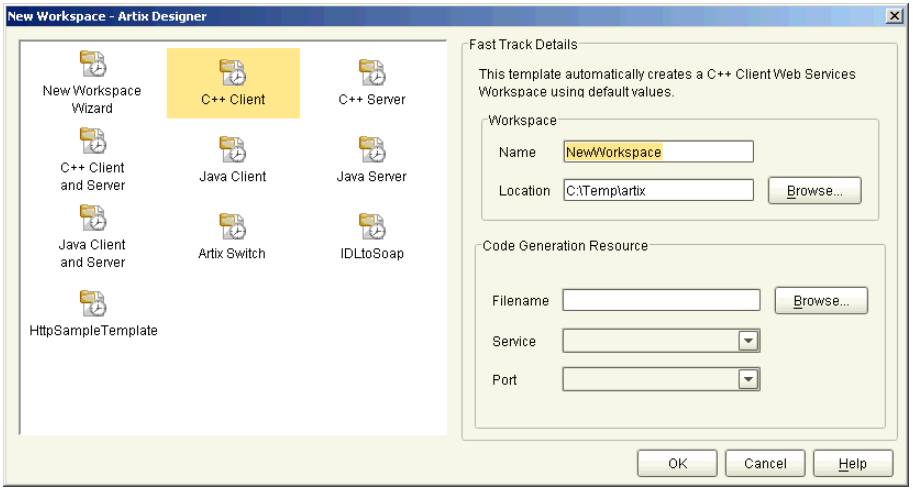

Figure 14: *New Workspace dialog showing Template options*

2. Enter a name and save location for your workspace, or accept the defaults provided. Click **Browse** to navigate to a different save location if you wish.
3. Enter the file name or URL for your WSDL file in the field provided, or click **Browse** to navigate to a suitable file.
4. Enter the name of a WSDL file to use to build this workspace, or click **Browse** to navigate to one.
5. Select a **service** to use for this workspace from the list of services that have been loaded into the Service drop-down list. These are the services contained in your WSDL file.
6. Select a **port** to use for this workspace from the list of ports that have been loaded into the Port drop-down list. These are the ports contained in your WSDL file.
7. Click **OK** to display the Artix Designer with your new workspace loaded into the Designer Tree.

Behind the scenes

When creating your new template-based workspace, Artix has automatically performed the following tasks:

- Created a workspace directory and file in the save location you specified
- Imported your WSDL file and added it to the workspace file
- Depending on which system you decided to create, it has created a local deployment profile including configuration options
- Created a deployment bundle containing all required code and a makefile for your target service

You could now create code for the workspace by selecting **Tools | Generate Code**, which would generate all files required.

Note than some hand editing of the implementation file will be required. For help with this, see *Developing Artix Applications in C++* or *Developing Artix Applications in Java*, depending on which type of code you're working with.

Working with Custom Templates

Overview

Artix 2.1 contains a sample template, HttpSample Template, that you can use in two ways:

- To gain a better understanding of how custom templates work
- To enable you to create custom workspaces that by default contain the same entities (profiles, bundles, resources)

Future versions of the Artix Designer will provide the capability for you to create and edit custom templates via the GUI - currently there is no capability for creating new templates, other than saving the sample as a different name. All editing must be done within a text editor.

Storing custom templates

The sample template is stored in the `\\artix\2.1\etc\xml\templates` installation directory. If you edit this template and save it as another name, then this edited template will be listed on the Workspace dialog just as the sample is now. Equally, if you saved various copies of this sample to that location then all of those copies would also be listed in the Workspace dialog.

If you want to change the directory that the Workspace dialog points to for these templates, you can do so via the Default Template Directory setting in the User Preferences dialog (under the **Edit** menu).

Procedure

To create a workspace based on the HttpSample template, select it in the Workspace dialog, and then:

1. Enter a name for the new workspace, or accept the default provided.
2. Enter a save location for the workspace, or accept the default provided.

3. Click Template Settings to display a dialog where you can define the settings for this workspace, as shown in Figure 15.

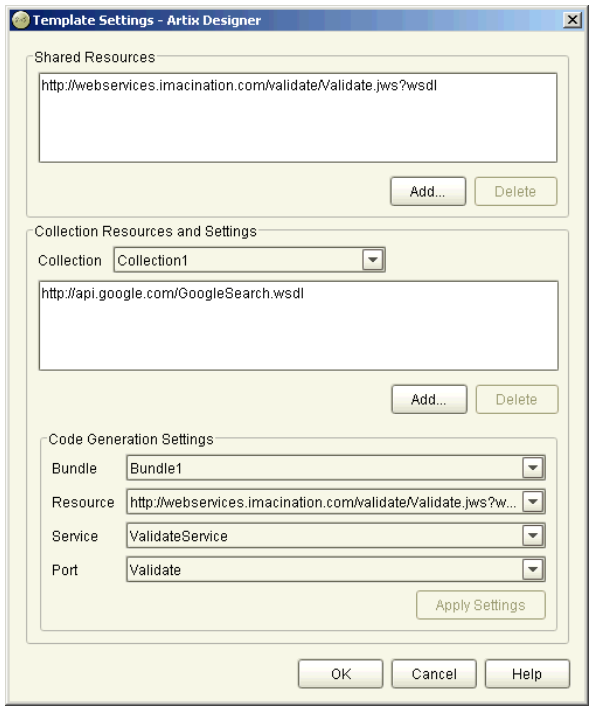

Figure 15: *Template Settings dialog*

4. The first field contains Shared Resources that will be included in this workspace. Click **Add** to locate other resources you want to add to the workspace as shared.

   If you edit the template file, you can have other resources listed in this field, as well as the other fields on this dialog, by default.

5. The Collection Resources and Settings section defines the resources that will be included in each Collection. In this template, two collections have been created and each contains a different resource.

   Again, you can click **Add** to locate other resources to add to the collections.

6. The Code Generation Settings section defines the necessary information for creating the deployment bundles for this workspace so that you can generate the application code.

   Two bundles have been created in this template and each uses different resources, services, and ports. If you change any of the pre-defined values in these fields, click Apply Settings to update the bundle.
7. Click **OK** to close this dialog and return to the New Workspace dialog.
8. Click **OK** to close the Workspace dialog and return to the Artix Designer, where your new workspace is loaded into the Designer Tree.

CHAPTER 3

Working with Artix Collections

A Collection is 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.

In this chapter

This chapter discusses the following topics:

What is a Collection?

Overview

A collection is a group of related resources that create the Web Service definition. Resources are WSDL *contracts* that can be created by importing WSDL files or by importing IDL files which are automatically converted into WSDL by Artix. A collection may contain one or more WSDL contracts.

At deployment time, a collection can be generated into physical entities such as Java, C++, or CORBA based applications. Contracts can also be based on data sets, such a COBOL Copybooks.

Collections are listed in the Designer Tree, as are the resources belonging to that collection, as shown in Figure 16.

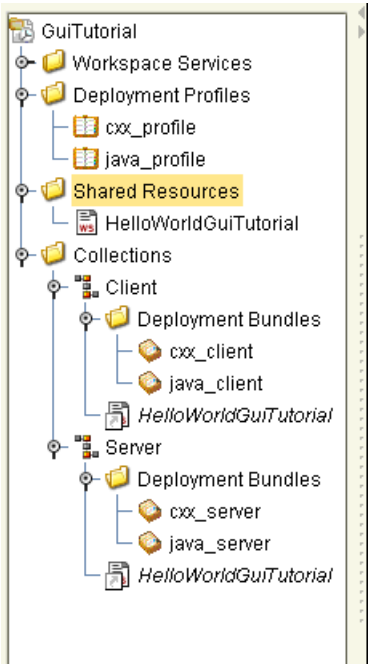

Figure 16: *Designer Tree showing Collections and Resources*

If you select a collection in the Designer Tree, the details for that collection are shown in the details panel on the right, as shown in Figure 17. In this panel you can view information about the resources contained in that collection, and about any Deployment Bundles that have been created for that collection.

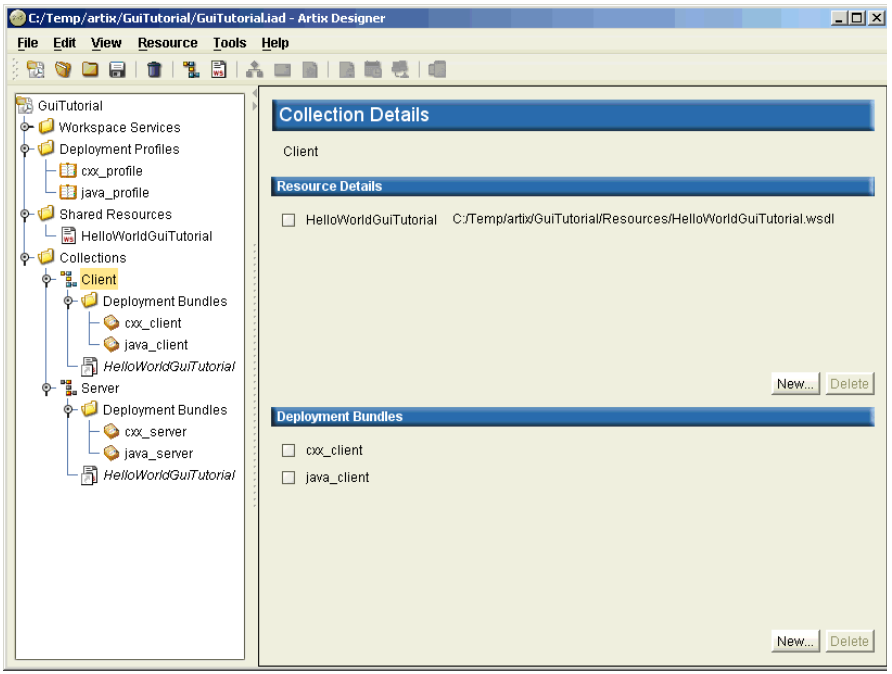

Figure 17: *Collection Details panel*

Creating a Collection

Overview

When you create a workspace using the New Workspace wizard (see page 26), you are given the option of creating a collection. Even though you are only given the option of creating one collection in this wizard, you can actually have as many collections in your workspace as you like.

Adding new collections is easy. You can click on the **Add** button under the Collections list on the Workspace Details panel (Figure 18), or you can select **File | New | Collection** from the menu bar.

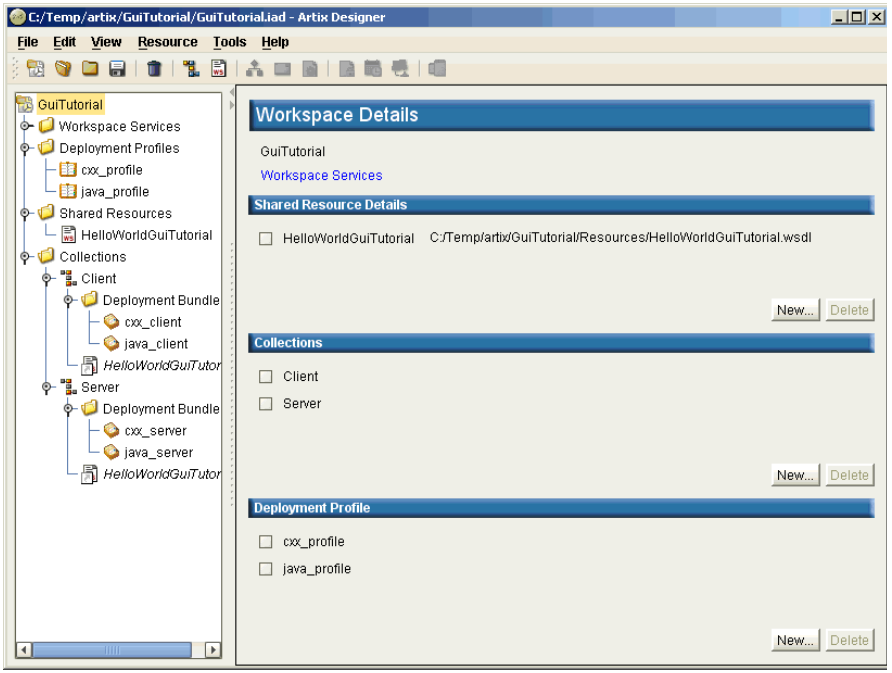

Figure 18: *Workspace Details panel*

Either way, you arrive at the New Collection wizard, as shown in Figure 19, and you can proceed through the procedure outlined below.

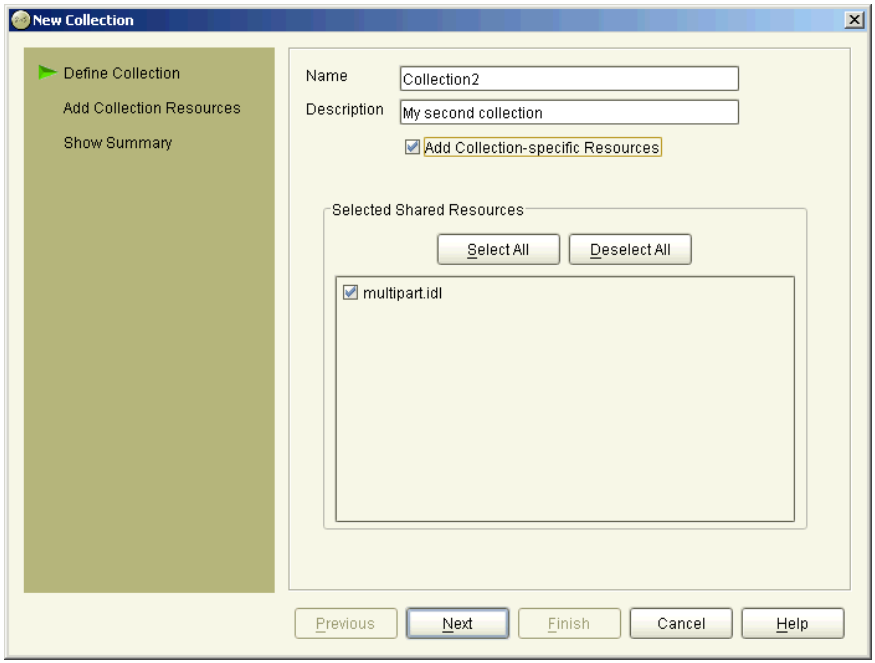

Figure 19: *New Collection wizard*

Procedure

1. Enter a name for your collection, or accept the default provided.
2. Enter a description of your collection. This description should explain the purpose of the collection.
3. If you want to add resources to this collection that are collection-specific, check the box provided. This will cause an extra panel (Add Collection Resources) to be added to the wizard.
4. By default, the shared resources contained in your workspace will be added to this collection. They are listed in the Shared Resources table. If you do not want to add any or all of these resources to your collection, click the check boxes to deselect them.

5. Click **Next** to display the Add Collection Resources panel, as shown in Figure 20. If you did not choose to add resources to this collection, the Summary panel will be displayed - see **step 8** for details.

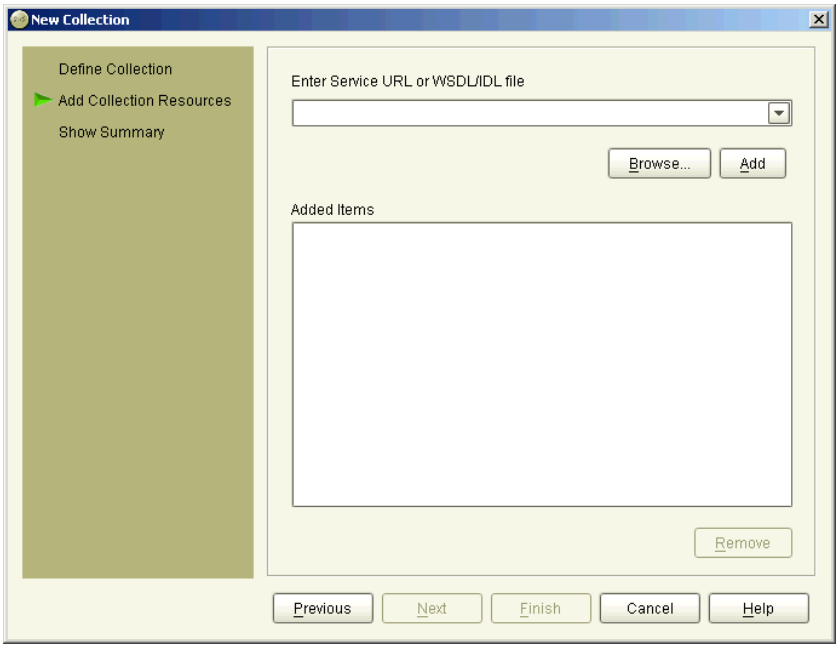

Figure 20: *New Collection wizard—Add Collection Resources panel*

6. Enter the URL or name of an existing file you would like to import into this collection as a resource. If you need to, click **Browse** to navigate to the file's location.
7. Click **Add** to add the file to the Added Items list and repeat as many times as necessary until you have added all the resources you want.

8. Click **Next** to display the Collection summary, as shown in Figure 21.

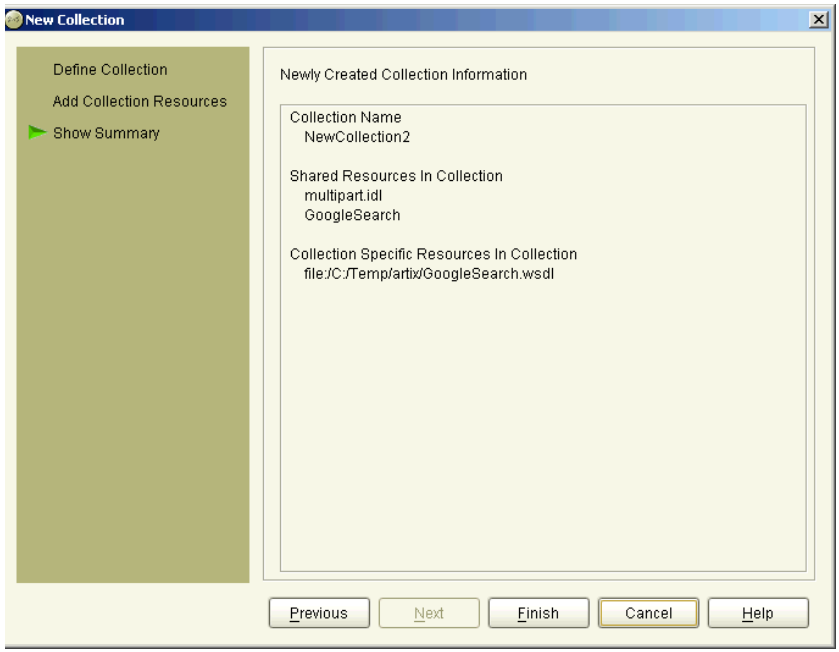

Figure 21: *New Collection wizard—Summary panel*

9. Click **Finish** to close this wizard and return to the Artix Designer, where you will see your new collection added to the Designer Tree.

Editing a Collection

Overview

You can make changes to your collection, or the resources within it, at any time using the Collection Details panel, as shown in Figure 22.

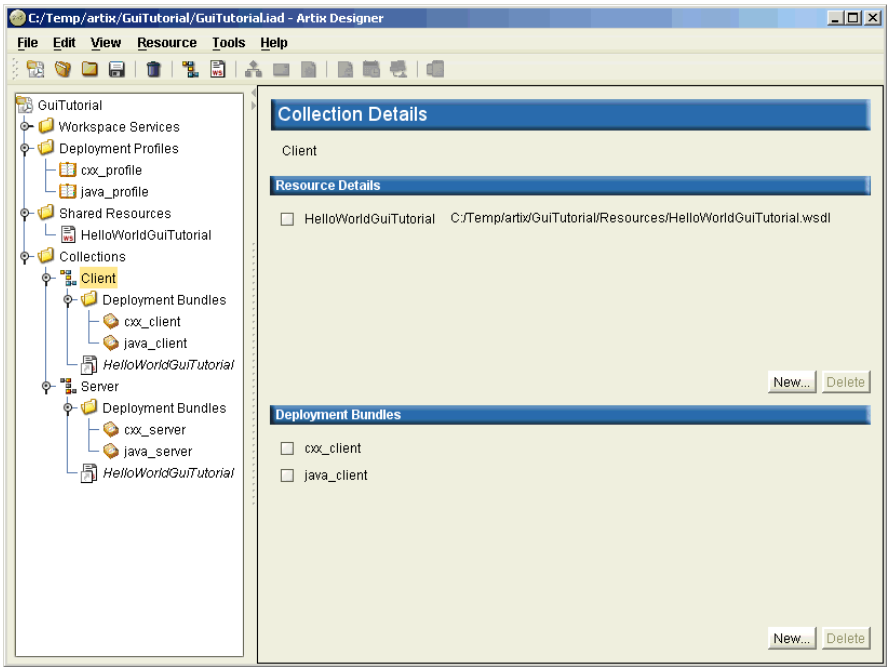

Figure 22: *Collection Details panel*

Adding and deleting resources

To add a resource, click **Add** to display the New Resource from File/URL dialog, as shown in Figure 23.

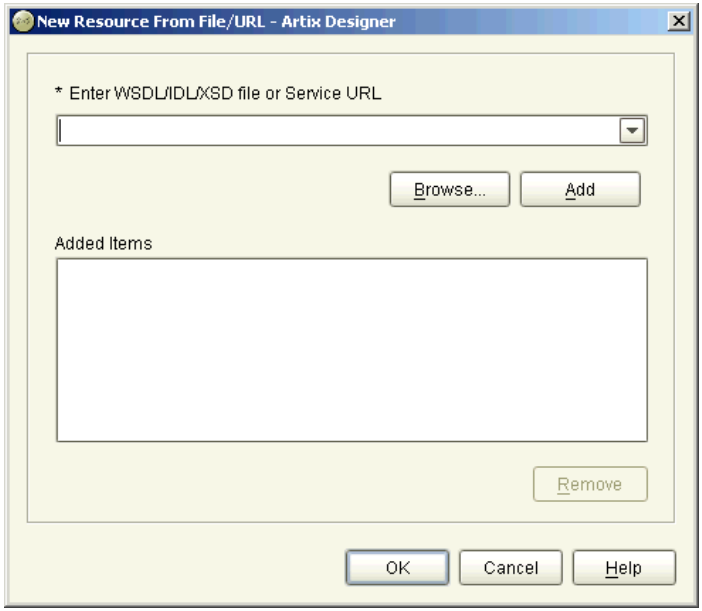

Figure 23: *New Resource from File/URL dialog*

For help with this dialog, and other information about adding resources, see "Working with Artix Resources" on page 47.

To delete a resource you need to first select it on the Collection Details panel using the check box provided, and then click **Delete**.

Adding and deleting Deployment Bundles

To add a Deployment Bundle, click the **Add** button under the Deployment Bundles list on the Collection Details panel to display the Deployment Bundle wizard.

To delete a Deployment Bundle, you need to first select it using the check box provided, and then click **Delete**.

Editing deployed collections

If you make changes to any contract in a collection that has had code generated, you should be aware that these changes could make the code for that collection invalid. It is recommended, therefore, that you regenerate the code any time that you change a previously deployed collection.

For more information, see "Generating Code" on page 192 for more information.

Invalid WSDL

Also, be aware that any changes you make to a resource could leave its underlying WSDL document invalid - if this happens you will only be able to view the contract by selecting the WSDL tab of the Resource Navigator, as shown in Figure 25 on page 49. In this view, any problems with the WSDL are listed, as shown in Figure 24, so that you can fix them and return the WSDL to a valid state.

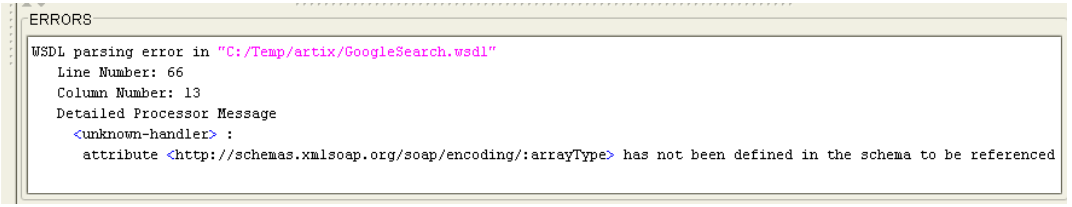

Figure 24: *Artix Designer Invalid WSDL Indicator*

Converting resources to shared

You can change a collection-specific resource within a collection to a shared resources, and thus have the opportunity to add it to all of your collections.

To do this, select the resource name in the Designer Tree and select **Resource | Convert to Shared**. This will invoke a dialog asking which other collections you would also like to include this resource - by default, all collections are selected.

Click **OK** to close this dialog and return to the Artix Designer. The resource is now in the Shared Resources list, plus all collections that you specified.

Generating Code for a Collection

As mentioned at the beginning of this chapter, collections can be generated into physical entities such as Java, C++, or CORBA based applications. Further, collections can be deployed as clients, servers, or switches, depending on your solution requirements.

The Artix deployment process has three steps:

1. Create a **Deployment Profile** - contains machine level information such as the Artix save location, the compiler location, and the operating system being used. A profile can be used multiple times as it is not specific to any particular collection defined within the workspace.

   To create a deployment profile, select **File | New | Deployment Profile** from the menu bar.

2. Create a **Deployment Bundle** - defines specific information about the deployment of a collection, such as the deployment type (client, server, or switch), configuration details, and code generation options.

   To create a deployment bundle, select a collection from the Designer Tree, then select **File | New | Deployment Bundle** from the menu bar.

3. Deploy the bundle - a very simple procedure once the profile and bundle are in place. Artix deploys the solution based on the information you provided in the bundle, and generates the code, environment scripts, and configuration files as specified in the locations you provided.

   To deploy a bundle, select a collection from the Designer Tree, then select **Tools | Generate Code** from the menu bar.

After generating the code, you need to perform some editing of the implementation code, and then you can run and compile the code.

For more information and detailed procedures for each of the deployment steps, see "Deployment" on page 177.

CHAPTER 4

Working with Artix Resources

A resource is an XML document that defines an interface to a collection.

In this chapter

This chapter discusses the following topics:

What are Resources?

Overview

An Artix Resource is an XML document that can be used to define the interface to a collection. In Artix 2.1 there are two resource types:

- Contracts, which can comprise one or more of the following:
  - WSDL documents
  - WSDL created from IDL files
  - WSDL created from data sets, such as COBOL Copybooks
- Schemas, which define types. Schemas can also be referenced from within contracts, if desired, to define the types for that contract.

If a resource is added to a workspace, it can take one of two roles:

- A "Shared" resource, which is automatically added to every collection in that workspace.
- A "collection-specific" resource, which only applies to the collection to which it is added.

How Artix helps you create a contract

When building a WSDL contract, the Artix Designer guides you through the process by making only relevant options available to you depending on the current state of that contract.

For example, if you are building your contract from scratch you need to add components to it in a certain order as there are dependencies between the components. In short, the contents of the Resource menu, as shown below, reflect the order in which components need to be added to the resource.

- Types - the first item to be created. You may not have to create extra ones though, as some primitive types exist by default.
- Messages - cannot be created without a Type. The primitive types will suffice if you don't want to add new types.
- Port Types - cannot be created without a Message.
- Bindings - cannot be creating without a Port Type.
- Services - cannot be creating without a Binding.
- Routes - cannot be created without two compatible Services.

Contracts and Schemas are explained in more detail later in this chapter.

Navigating Resources

Overview

The Artix Designer provides an interface tool, called the Resource Navigator, that gives you two views of a resource - diagram or text. These views are accessed via tabs at the bottom of the Designer's details panel.

Diagram view

Depending on how you want to work with your resource, you can use either of the available views. If you aren't very familiar with XML, you will find it easier working in the diagram view, as shown in Figure 25.

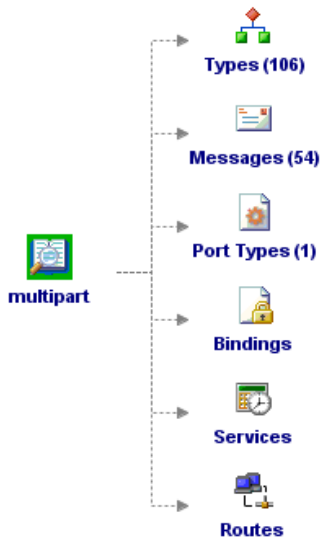

Figure 25: *Resource Navigator—Diagram view*

The WSDL model

The diagram view shows you the WSDL model. As seen in Figure 25, the multi-part contract has 106 types, 54 messages, and 1 port type. It currently contains no bindings, services, or routes.

If you right-click on one of the components, such as types, you are given the option to create a new type, or to edit or view the existing types. You can also expand and collapse the list of existing types.

The model expanded

When you expand the existing types, the Resource Navigator changes to look more like Figure 26.

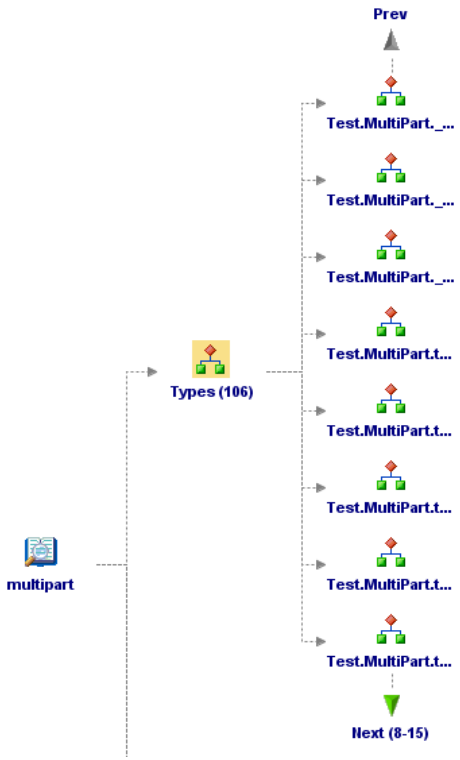

Figure 26: *Resource Navigator showing Types Expanded*

Navigating the expanded model

Now you can scroll through the individual types and right-click on any of them to edit or view the attributes for that type. The Resource Navigator lists eight "child" components at any one time. To scroll to view the next or the previous eight, you just need to click on the **Next** or **Previous** arrows.

If you want to see a list of all of the child components, right-click on the parent node and click Go To, from where you can jump to any of the listed types.

Viewing relationships between components

As you work down through each of the components in a WSDL contract, you can expand them to a point that displays the relationships between them. For example, for a contract that contains define services, bindings, port types, messages, and types you can expand the each service right out to view every related component, right out to its types.

Text view

If you prefer, you can work directly in the XML text for the resource by selecting the **Text** tab, as shown in Figure 27.

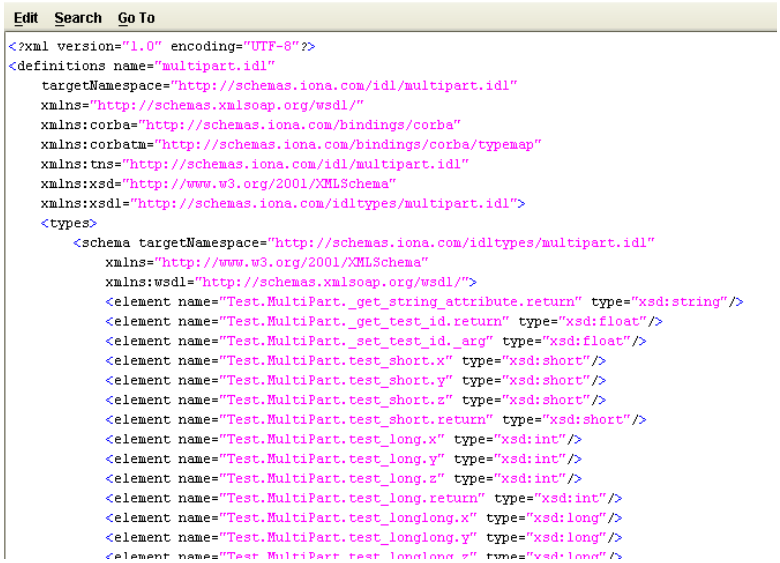

Figure 27: *Resource Navigator—Text view*

Editing tools

In this view you can hand edit the resource. Tools under the **Edit** menu in this view make the task easier. You can also use the **Search** and **Go To** functions to locate segments or specific lines within the text.

Validating your changes

When you make changes to the text and click **Apply Edits**, Artix checks that your changes have not compromised the XML for the resource. If your changes have made the resource invalid, the errors are listed in the Errors panel so that you can go to the relevant line and correct them. Figure 28 shows an example of the Error panel.

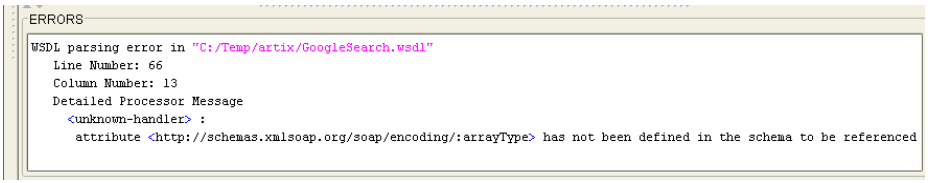

Figure 28: *Error panel*

What is a Contract?

Overview

Artix contracts describe Artix resources and their integration. They are written in WSDL. Each mapping of a port type to a binding and port defines an Artix collection. The contract also describes the routing between collections. It has two sections:

- Logical - describes the abstract operations, messages, and data types used by a collection.
- Physical - describes the concrete message formats and transports used by a collection. The routing information defining how messages are mapped between different collections is also specified here.

The Logical Section

The logical section of an Artix Contract defines the abstract operations that the collections offer. The logical view includes the `<types>`, `<message>`, and `<portType>` tags in a WSDL document. This portion of the contract also specifies the namespaces used in defining the contract.

Types

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

For information about adding Types to your Artix contract, see "Adding Types" on page 60.

Messages

WSDL is designed to describe how data is passed over a network and because of this 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, that are the formal data elements of the abstract message. Each part is identified by a name and an attribute specifying its data type.

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.

For information about adding messages to your Artix contract, see "Adding Messages" on page 73.

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.

For information about adding Port Types to your Artix contract, see "Adding Port Types" on page 77.

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 port type must have a unique name, specified using the `name` attribute. The `name` attribute is required to define 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.

For information about adding Operations to your Artix contract, see the Operations section in "Adding Port Types" on page 77.

The Physical Section

The physical section of an Artix contract defines the bindings and transports used by the collections. It includes the information specified in the `<binding>` and `<service>` tags of a WSDL document. It also includes the routing rules defining how the messages are routed between the endpoints defined in the document.

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.

WSDL is intended to describe services offered over the Web and therefore most bindings are specified using SOAP as the message format. WSDL can bind data to other message formats however.

Artix provides bindings for several message formats including CORBA, SOAP, and XML. For specific information on using bindings see "Adding Bindings" on page 115.

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. For more information, see "Adding Services" on page 139.

Routing

To fully describe the integration of collections across an enterprise, Artix contracts include routing rules for directing data between the collections. Routing rules are described in "Routing Messages" on page 167.

For more information

For more detailed information about Artix contracts and their components, see either the *Artix Getting Started Guide*, or *Developing Artix Solutions from the Command Line*.

What is a Schema?

Overview

An XML Schema is similar to a contract, except that it only defines types. As such, it cannot really be called a contract. It is possible, however, to create a contract containing a reference to a schema to define the contract's types. Figure 29 shows a Schema in the Resource Navigator diagram view.

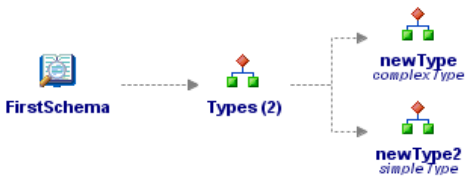

Figure 29: *Schema—diagram view*

Figure 30 shows the same schema, this time in text view.

File Edit Search Go To

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://www.iona.com/artix/2.1.1/FirstSchema"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="http://www.iona.com/artix/2.1.1/FirstSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <complexType name="newType">
        <all>
            <element maxOccurs="3" minOccurs="0" name="new string" type="xsd:string"
        </all>
    </complexType>
    <simpleType name="newType2">
        <restriction base="xsd:string">
            <length value="8"/>
            <maxLength value="12"/>
        </restriction>
    </simpleType>
</schema>
```

Figure 30: *Schema—text view*

Creating New Resources

Overview

Creating Artix resources from scratch takes a little time, but is still easy to do using the Designer. Wizards guide you through the process.

As explained in "Creating a Collection" on page 38, a WSDL contract is made up of a logical and a physical part. The logical part contains types, messages, port types and operations. The physical part contains services and bindings. A Schema is much simpler - it just defines types.

This section explains how to create schemas and how to create the logical part of Artix contracts. The topics discussed are:

For information on adding bindings to your resource, see "Adding Bindings" on page 115.

For information on adding services to your resource, see "Adding Services" on page 139.

Creating a Contract

Overview

The first thing you need to do is create a contract shell. Depending on which mode of the Designer you are working in (Deployer or Editor), the steps you follow to do this will be slightly different.

If you are working in Editor mode, select **New | WSDL Contract** to display the New Contract dialog, as shown in Figure 32 on page 59.

If you are working in Deployer mode:

1. Select either the Shared Resources folder or a Collection from the Designer Tree.
2. Select **File | New | Resource** from the **File** menu to display the New Resource dialog, as shown in Figure 31.

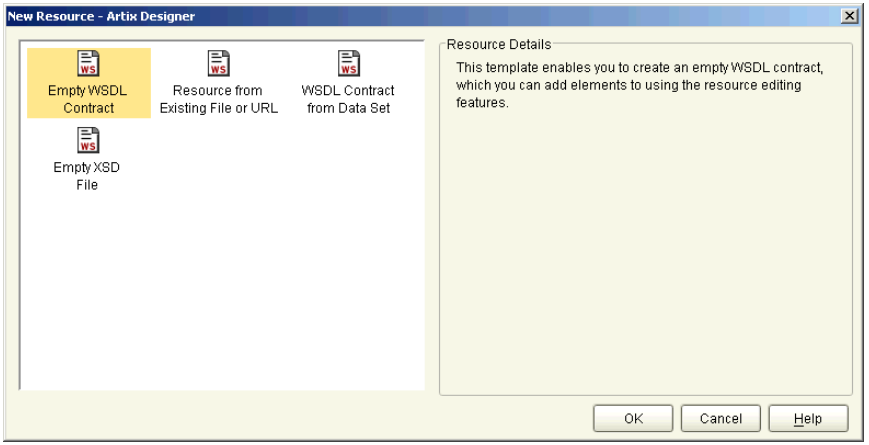

Figure 31: *New Resource dialog*

3. Select the **Empty WSDL Contract** icon and click **OK** to display the New Contract dialog, as shown in Figure 32.

Figure 32: *New Contract dialog*

4. Enter a name in the **Name** field, or accept the default provided.
5. Enter a value in the **Target Namespace** field, or accept the default provided.
6. Click **OK** to close this dialog and return to the Artix Designer. Your new contract will be shown, and you can now add types, messages, and port types to it using the procedures documented over the following pages.

Adding Types

Procedure

To add a Type to your resource:

1. Select **Resource | New | Type** from the menu bar to display the New Type wizard, as shown in Figure 33.

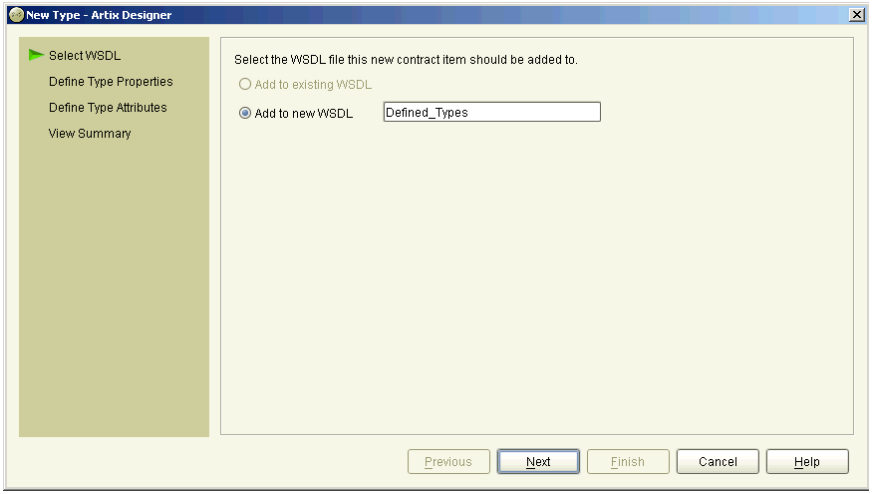

Figure 33: *New Types wizard*

2. Select where to create the WSDL entry for the new type.
   - **Add to existing WSDL** adds the type information to the existing contract.
   - **Add to new WSDL** creates a new WSDL document that contains the type information.

   If, like in this example, you have an instance where the first option on this panel - Add to existing WSDL - is not able to be selected, it indicates that your WSDL file is read-only. Thus, you only have the option or creating a new WSDL file for the new type.

3. Click **Next** to display the Type Properties panel as displayed in Figure 34.

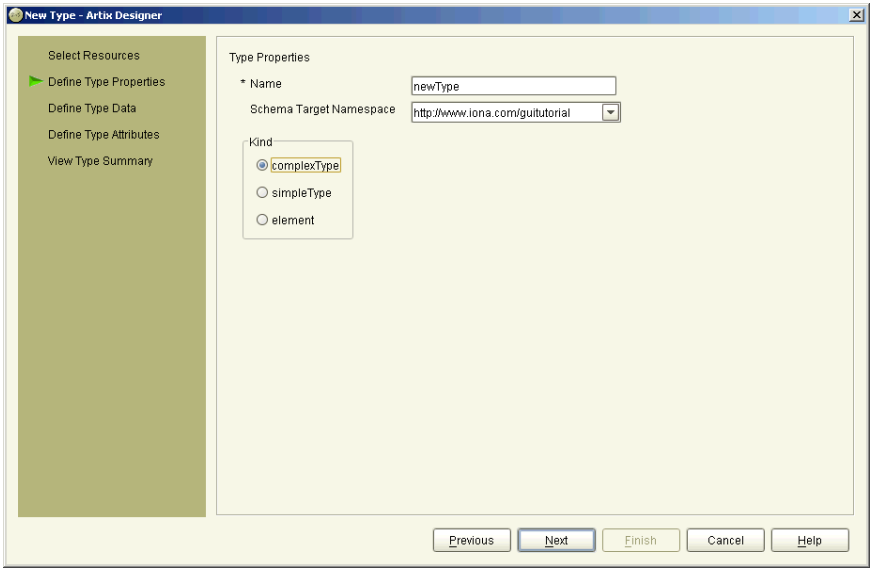

Figure 34: *New Types wizard—Type Properties panel*

4. Enter a name for the new type, or accept the default provided.
5. You can specify a target namespace for this type if you like - if you don't the default WSDL target namespace is applied.
6. Select the Kind value for the type - **complex, simple, or element**.

7. Click **Next** to display the Define Type Data panel, as shown in Figure 35.

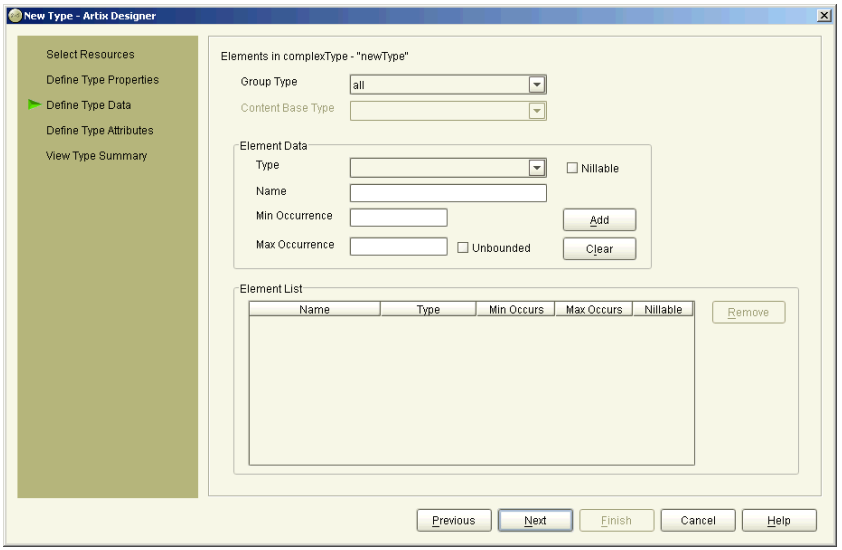

Figure 35: *New Types wizard—Define Type Data panel*

8. Depending on the Kind of type you selected, different options are displayed on the Type Attributes panel. The example shown in Figure 35 shows the options for a complex type. Continue with **step 8**, if this is the kind of type you are creating, otherwise go to one of the following steps:
   - **Step 19** for a simple type
   - **Step 24** for an element

Complex type attributes

9. At the Define Type Data panel, as shown in Figure 35, select a Group Type value from the list provided. This defines how the complex type elements will be mapped to data structures.
10. If you select one of the content types (`simplecontent` or `complexcontent`), the Content Base Type field is enabled. From this field you can select the type you would like to use as a starting point for your content type.
11. Provide values for each of the Element Data fields:
    - Type - the base type for this schema
    - Name - a unique string identifier for element
    - Minimum Occurrence - the minimum times you want the element to be present (not an option for content types)
    - Maximum Occurrence - the maximum times you want the element to be present (not an option for content types)

    If this element is going to be a required field in your application, then you should select the Required check box provided.
12. Select the **Unbounded** check box if there is no maximum occurrence limit. **Note**: this is not an option for content types.
13. Select the **Nillable** check box if you want to indicate that this element could potentially be omitted completely, or could pass an empty object across the wire. **Note**: this is not an option for content types.
14. Click **Add** to move the details you have provided for this element into the Element List table.

    To edit an element in this table, select it and then make the changes in the fields above the table. Click **Update** to refresh the values in the table.

    You can delete an element from this table by selecting it and clicking **Remove**.
15. Repeat **steps 9 - 14** until you have added all of your elements.

16. Click Next to display the Define Type Attributes panel, as shown in Figure 36.

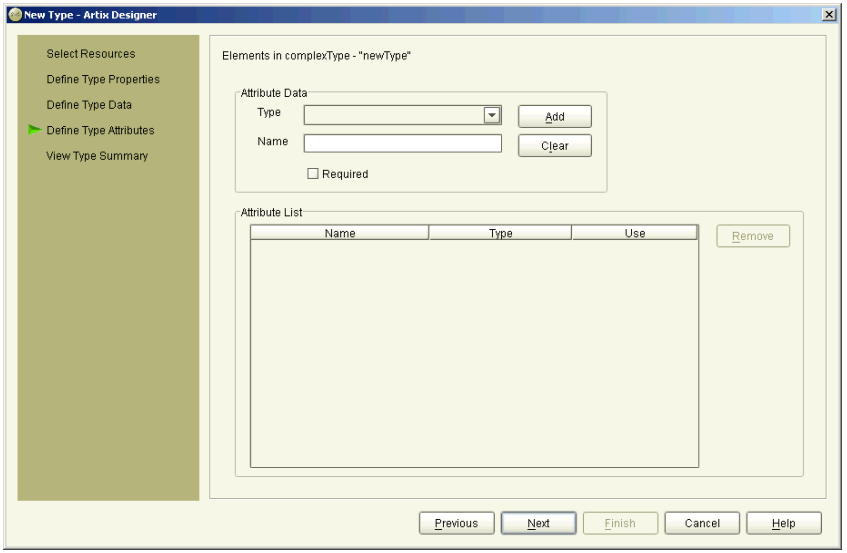

Figure 36: *New Types wizard—Define Type Attributes panel*

17. Click **Next** to view the Summary panel.
18. If you would like to add another Type, click the check box provided and click **Next**. This will return you to the Type Properties panel, as displayed in Figure 34 on page 61.

    Alternatively, click **Finish** to close this wizard and return to the Artix Designer.

Simple type attributes

19. At the Define Type Data panel, as shown in Figure 37, select a Base Type from the list provided, for example, **string** or **boolean**.

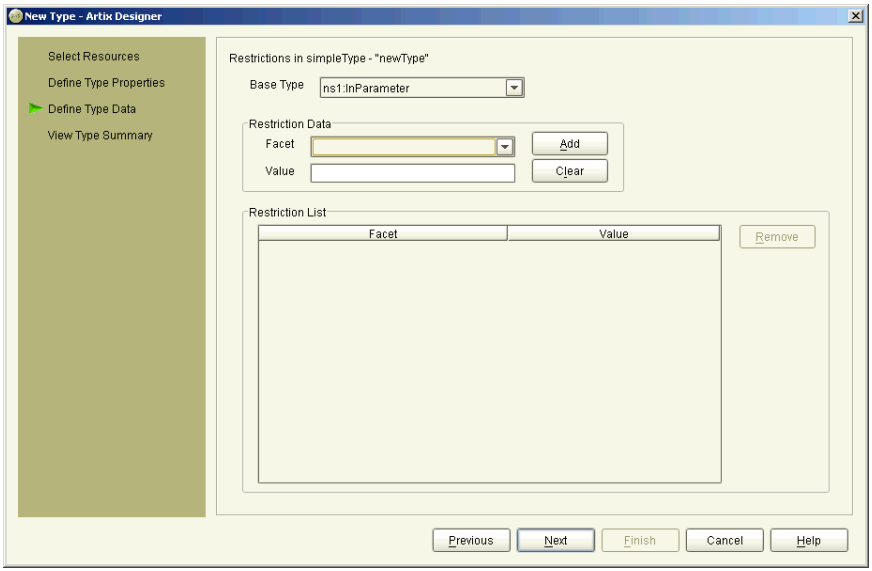

Figure 37: *New Types wizard—Type Data (simple) panel*

20. Provide values for each of the Restriction Data fields:
    - Facet - a characteristic of the base type, for example for a string, the available facets would be *enumeration*, *length*, or *maxLength.*
    - Value - the value for the facet, for example the value for *length* would be a non-negative integer.
21. Click **Add** to move the details you have provided for this restriction into the Restriction List table.

    To edit a restriction in this table, select it and then make the changes in the fields above the table. Click **Update** to refresh the values in the table.

    You can delete a restriction from this table by selecting it and clicking **Remove**.

22. Repeat **steps 16 - 18** until you have added all of your restrictions.
23. Click **Next** to view the Summary panel, as shown in Figure 38.

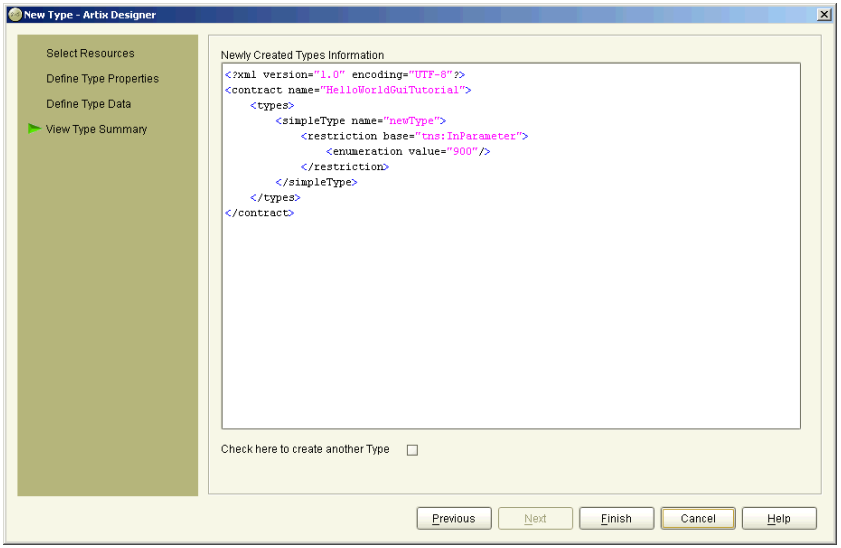

Figure 38: *New Type wizard—Summary panel for Simple Types*

24. If you would like to add another Type, click the check box provided and click **Next**. This will return you to the Type Properties panel, as displayed in Figure 34 on page 61.

    Alternatively, click **Finish** to close this wizard and return to the Artix Designer.

Element attributes

25. At the Define Type Data panel, as shown in Figure 39, select the Nillable check box if you want to indicate that this element could potentially be omitted completely, or could pass an empty object across the wire.

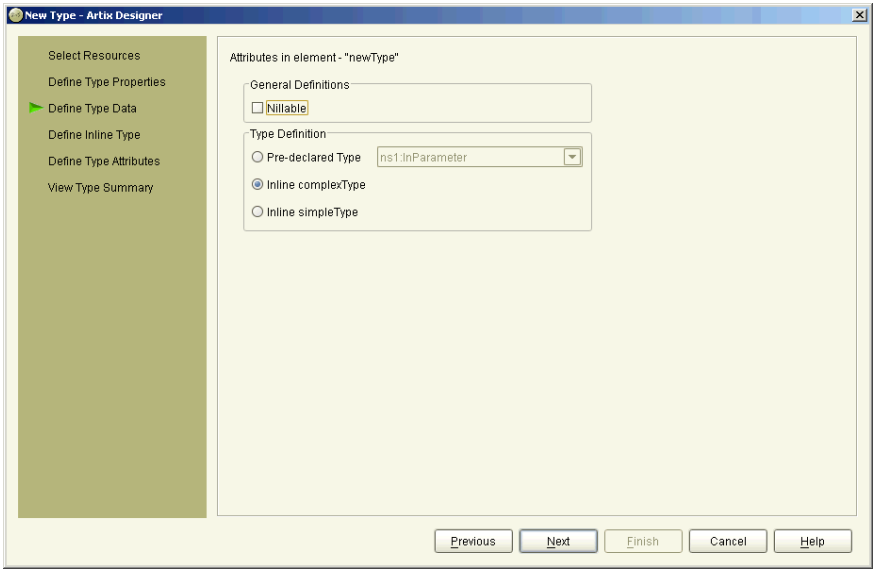

Figure 39: *New Types wizard—Type Attributes (element) panel*

26. Select an Attribute Type Definition. Options available are:
    - Pre-declared type - any type that has been pre-defined or is one of the standard primitive types
    - Inline complextype - an "anonymous" complex type that can be used within this element only; is not available for use by other elements or types
    - Inline simpletype - an "anonymous" simple type that can be used within this element only; is not available for use by other elements or types
27. Depending on what you select here, clicking **Next** will display one of the following:

- The Define Inline Type (complex) panel, as displayed in Figure 40, - continue with the next step in this procedure
- The Define Inline Type (simple) panel - jump to step **40** in this procedure
- The View Summary panel - jump to step **42** in this procedure.

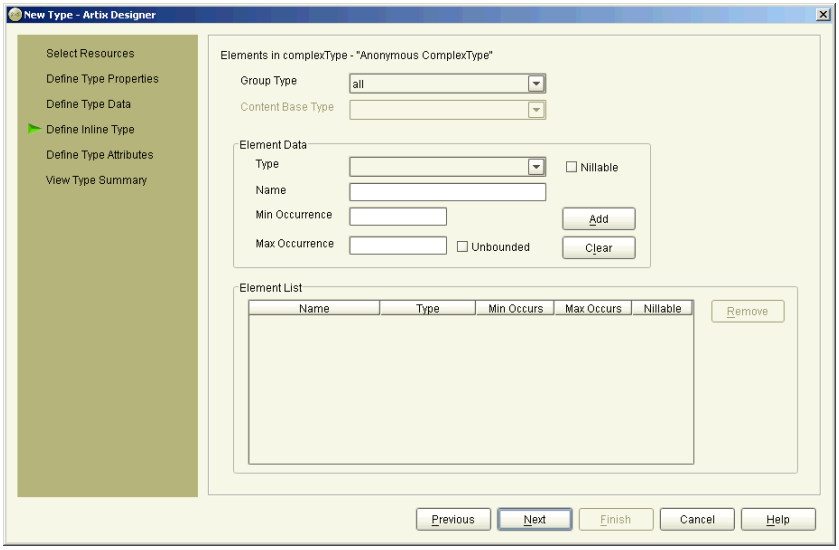

Figure 40: *New Types wizard—Define Inline Type panel (complex)*

Inline complextype

28. Select a Group Type value from the list provided. This defines how the complex type elements will be mapped to data structures.
29. If you select one of the content types (`simplecontent` or `complexcontent`), the Content Base Type field is enabled. From this field you can select the type you would like to use as a starting point for your content type.
30. Provide values for each of the Element Data fields:
    - Type - the base type for this schema
    - Name - a unique string identifier for element

- Minimum Occurrence - the minimum times you want the element to be present (not an option for content types)
- Maximum Occurrence - the maximum times you want the element to be present (not an option for content types)

If this element is going to be a required field in your application, then you should select the Required check box provided.

31. Select the **Unbounded** check box if there is no maximum occurrence limit. **Note**: this is not an option for content types.
32. Select the **Nillable** check box if you want to indicate that this element could potentially be omitted completely, or could pass an empty object across the wire. **Note**: this is not an option for content types.
33. Click **Add** to move the details you have provided for this element into the Element List table.

    To edit an element in this table, select it and then make the changes in the fields above the table. Click **Update** to refresh the values in the table.

    You can delete an element from this table by selecting it and clicking **Remove**.
34. Repeat **steps 28 - 33** until you have added all of your elements.

35. Click Next to display the Define Type Attributes panel, as shown in Figure 41.

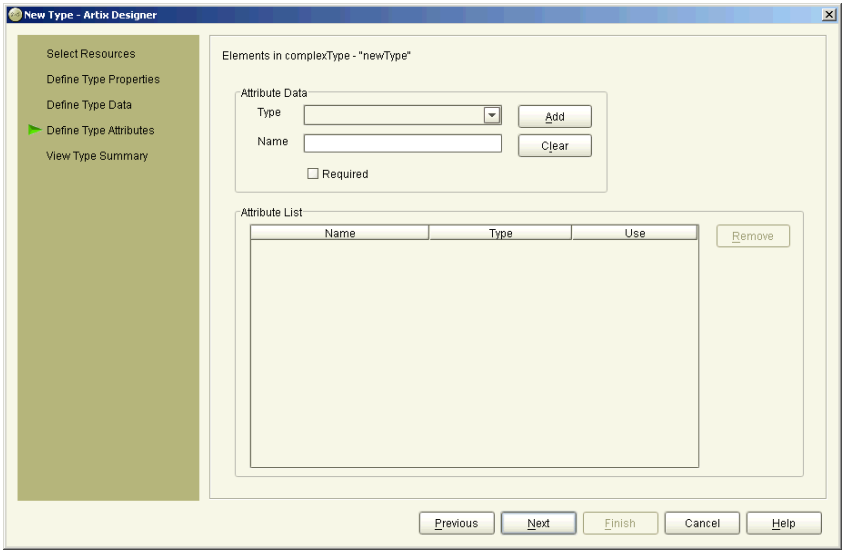

Figure 41: *New Types wizard—Define Type Attributes panel*

36. Click **Next** to view the Summary panel.
37. If you would like to add another Type, click the check box provided and click **Next**. This will return you to the Type Properties panel, as displayed in Figure 34 on page 61.

    Alternatively, click **Finish** to close this wizard and return to the Artix Designer.

Inline simpletype

38. At the Define Inline Type (simple) panel, as shown in Figure 42, select a Base Type from the list provided, for example, **string** or **boolean**.

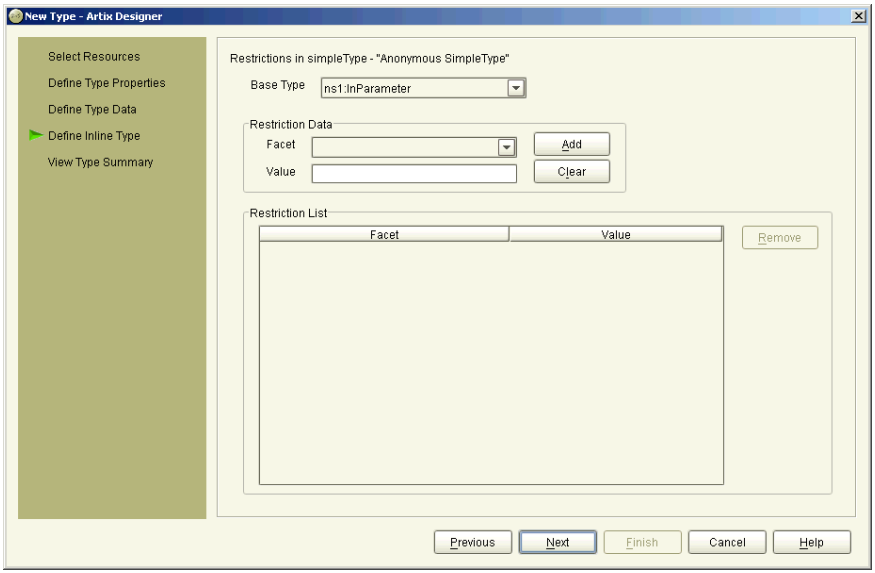

Figure 42: *New Types wizard—Define Inline Type (simple) panel*

39. Provide values for each of the Restriction Data fields:
    - Facet - a characteristic of the base type, for example for a string, the available facets would be *enumeration*, *length*, or *maxLength.*
    - Value - the value for the facet, for example the value for *length* would be a non-negative integer.
40. Click **Add** to move the details you have provided for this restriction into the Restriction List table.

    To edit a restriction in this table, select it and then make the changes in the fields above the table. Click **Update** to refresh the values in the table.

    You can delete a restriction from this table by selecting it and clicking **Remove**.

41. Repeat **steps 38 - 40** until you have added all of your restrictions.
42. Click **Next** to view the Summary panel.
43. If you would like to add another Type, click the check box provided and click **Next**. This will return you to the Type Properties panel, as displayed in Figure 34 on page 61.

    Alternatively, click **Finish** to close this wizard and return to the Artix Designer.

Adding Messages

Procedure

To add a Message to your resource:

1. Select **Resource | New | Message** from the menu bar to display the New Message wizard, as shown in Figure 43.

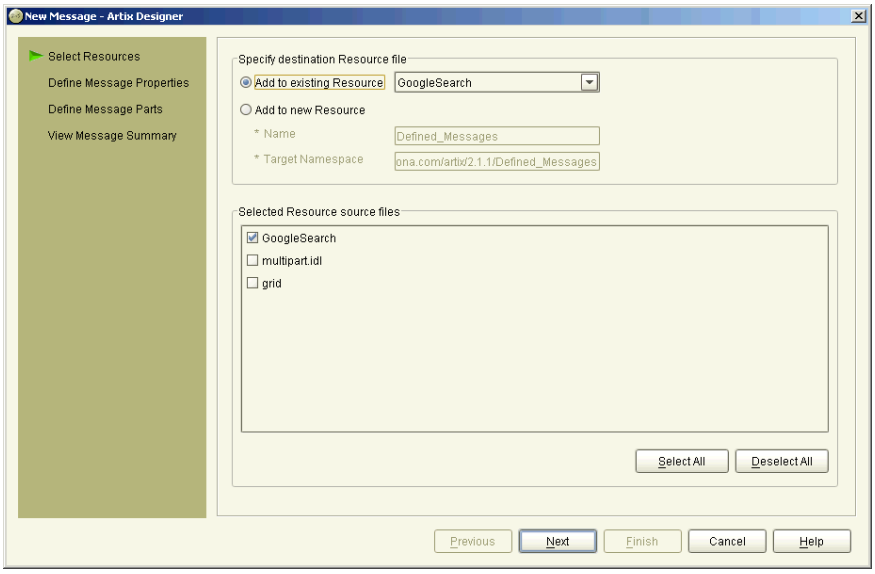

Figure 43: *New Message wizard*

2. Select where to create the WSDL entry for the new message.
   - **Add to existing WSDL** adds the message information to an existing contract.
   - **Add to new WSDL** creates a new WSDL document that contains the message information.
3. Select the resources from this collection that you want to use as the source for this new message. If you selected a resource before invoking the New Message wizard, that resource is selected by default.

You can also select other resources to use as sources for this message - this will give you more types to choose from when defining message parts later in this wizard.

4. Click **Next** to display the Message Properties panel, as shown in Figure 44.

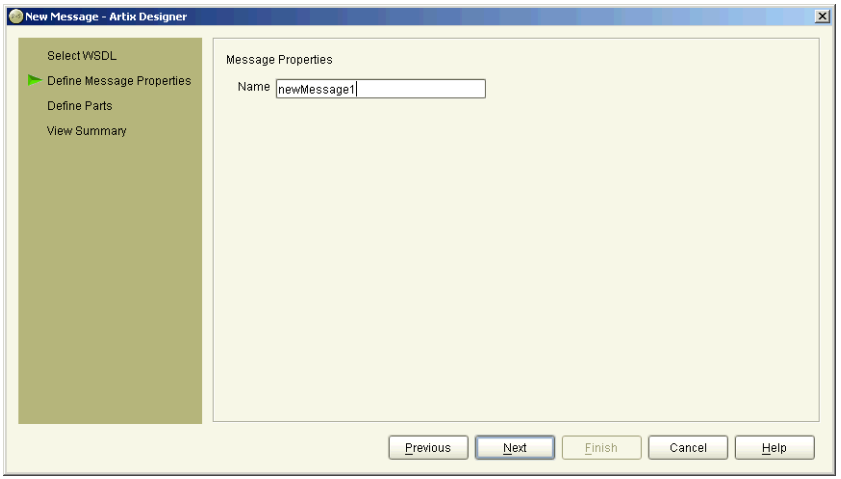

Figure 44: *New Message wizard—Message Properties panel*

5. Enter a name for the message, or accept the default provided.

6. Click **Next** to display the Message Parts panel, as shown in Figure 45.

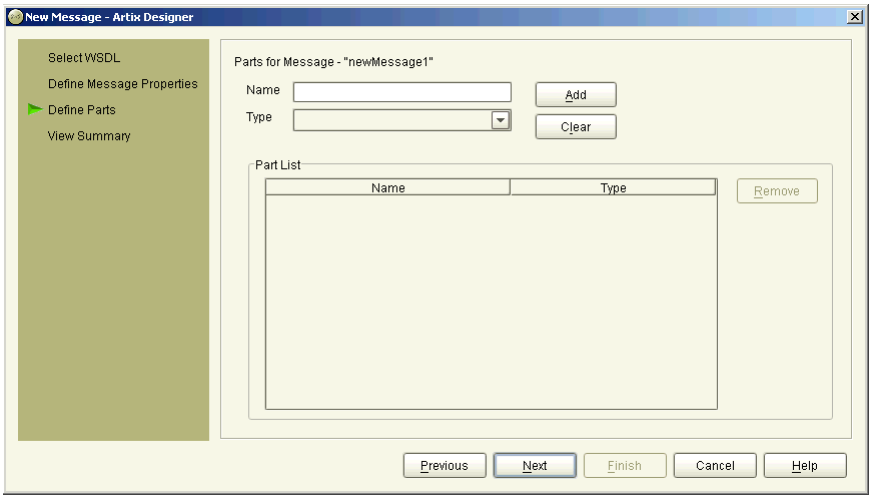

Figure 45: *New Message wizard—Message Parts panel*

7. Enter a name for the message part, and select a type from the list provided.
8. Click **Add** to move the details you have provided for this part into the Part List table.

   To edit a part in this table, select it and then make the changes in the fields above the table. Click **Update** to refresh the values in the table.

   You can delete a part from this table by selecting it and clicking **Remove**.
9. Repeat **steps 6 and 7** until you have added all of your parts.

10. Click **Next** to view the Summary panel, as shown in Figure 46.

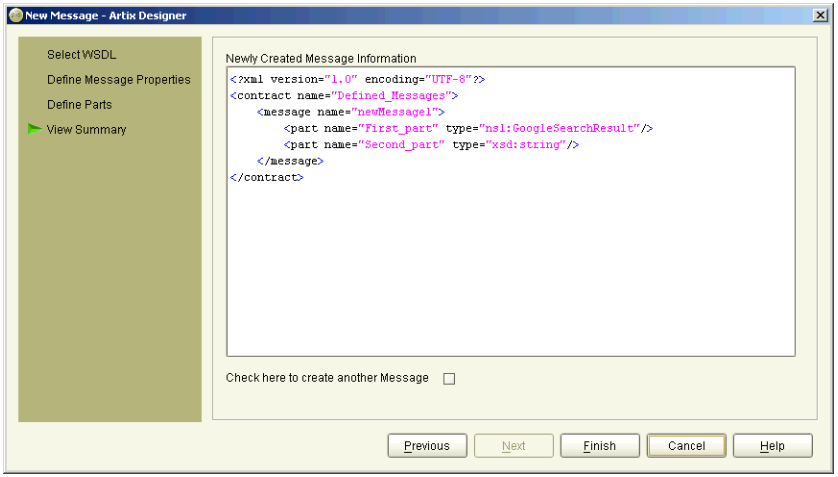

Figure 46: *New Messages wizard—Summary panel*

11. If you would like to add another Message, click the check box provided and click **Next**. This will return you to the Message Properties panel, as displayed in Figure 44 on page 74.

    Alternatively, click **Finish** to close this wizard and return to the Artix Designer.

Adding Port Types

Procedure

To add a Port Type to your resource:

1. Select **Resource | New | Port Type** from the menu bar to display the New Port Type wizard, as shown in Figure 47.

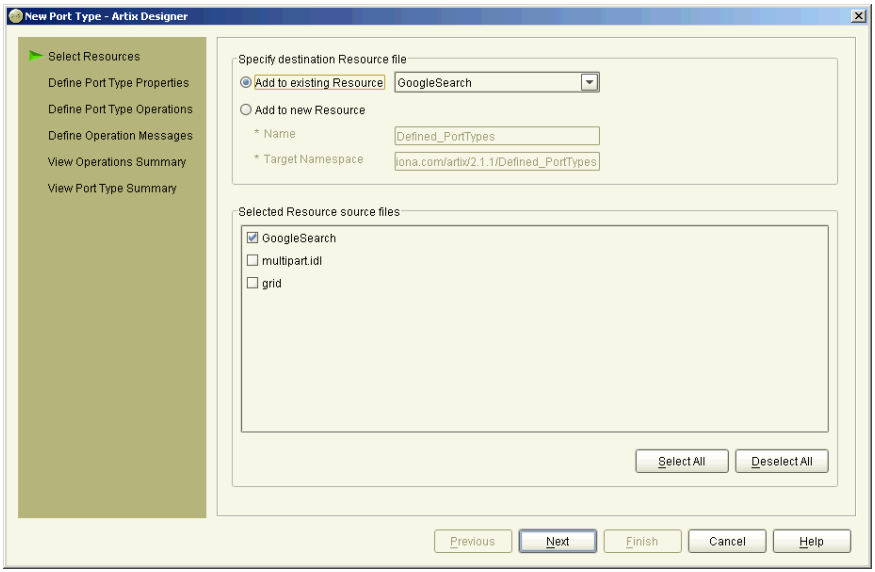

Figure 47: *New Port Type wizard*

2. Select where to create the WSDL entry for the new port type.
   - **Add to existing WSDL** adds the port type information to the existing contract.
   - **Add to new WSDL** creates a new WSDL document that contains the port type information.
3. Select the resources from this collection that you want to use as the source for this new port type. If you selected a resource before invoking the New Port Type wizard, that resource is selected by

default. You can also select other resources to use as sources for this port type - this will give you more messages to choose from when defining the operations later in this wizard.

4. Click **Next** to display the Port Type Properties panel, as shown in Figure 48.

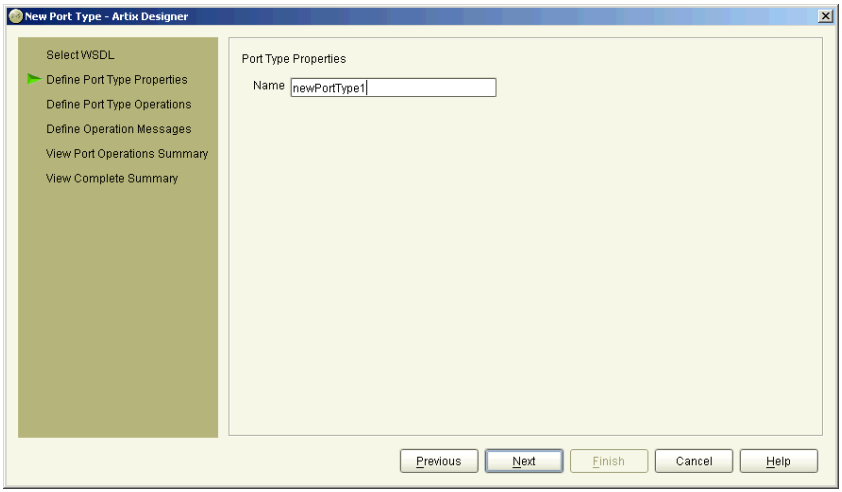

Figure 48: *New Port Type wizard—Port Type Properties panel*

5. Enter a name for the port type, or accept the default provided.

6. Click **Next** to display the Port Type Operations panel, as shown in Figure 49.

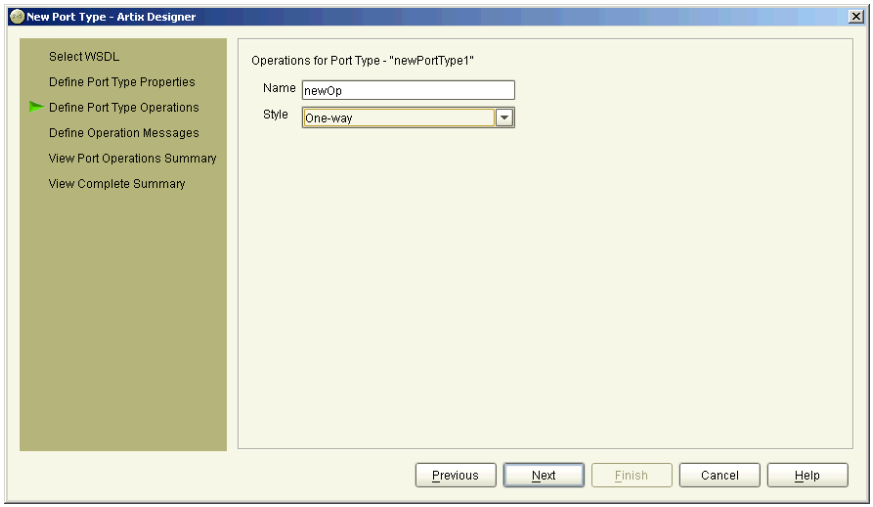

Figure 49: *New Port Type wizard—Port Type Operations panel*

7. Enter a name for the new operation and select a style from the list provided. Valid options are:
   - one-way
   - request-response

8. Click **Next** to display the Operation Messages panel, as shown in Figure 50.

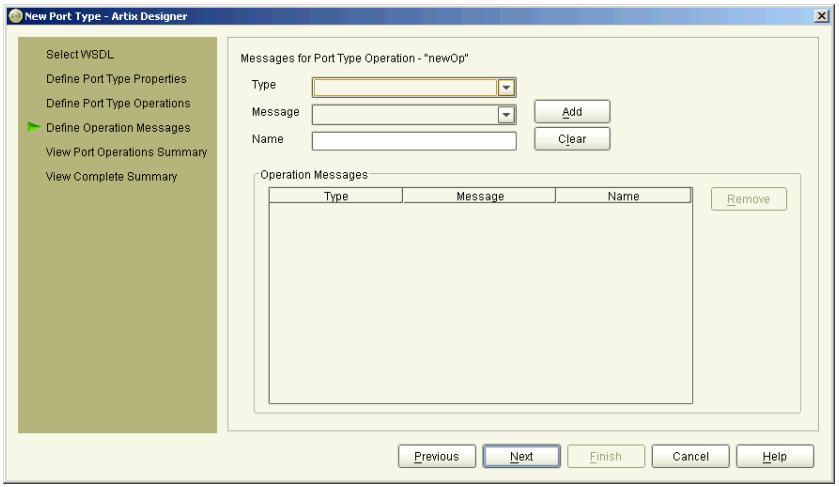

Figure 50: *New Port Type wizard—Operation Messages panel*

9. Select a Message Type from the list provided.
10. Select a Message from the list provided.
11. Enter a name for the message, or accept the one provided.
12. Click **Add** to move the details you have provided for this operation message into the Message List table.

    To edit a message in this table, select it and then make the changes in the fields above the table. Click **Update** to refresh the values in the table.

    You can delete a message from this table by selecting it and clicking **Remove**.

13. Click **Next** to display the Port Operations Summary panel, as shown in Figure 51.

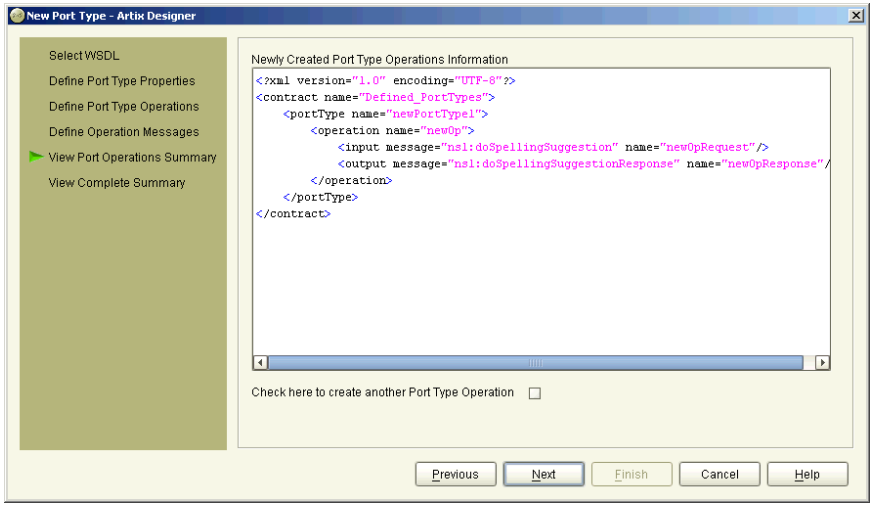

Figure 51: *New Port Type wizard—Port Operations Summary panel*

14. If you would like to add another Port Type Operation, click the check box provided and click **Next**. This will return you to the Port Type Operation panel, as displayed in Figure 49 on page 79.

Alternatively, click Next to display the Port Type Summary panel, as shown in Figure 52.

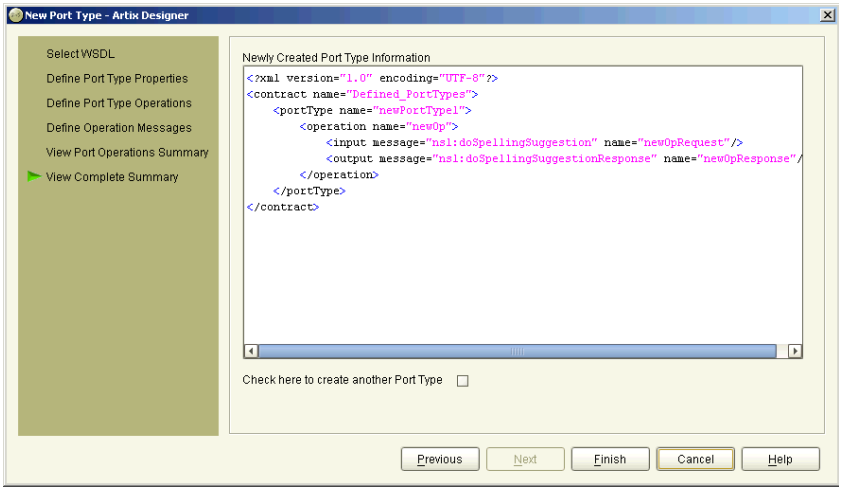

Figure 52: *New Port Type wizard—Port Type Summary panel*

15. If you would like to add another Port Type, click the check box provided and click **Next**. This will return you to the Port Type Properties panel, as displayed in Figure 48 on page 78.

    Alternatively, click **Finish** to close this wizard and return to the Artix Designer.

Adding Access Control Lists

Procedure

You can create access controls lists (ACL) to define roles for each operation in a port type.

To create an access control list:

1. Select a resource in the Designer Tree and select **Resource | New | Access Control List** to display the New Access Control List wizard, as shown in Figure 53.

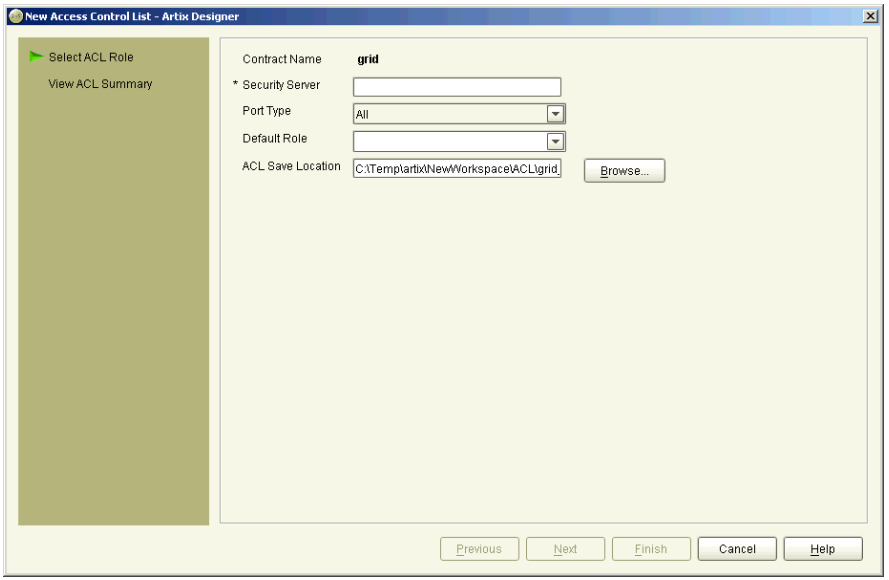

Figure 53: *New Access Control List wizard*

2. Type the name of your security server in the field provided.
3. In the Port Type drop-down, either accept the default of **All** to assign the same role to all port types in this resource, or select a port type to assign roles at the operation level.

   Selecting a port type will insert another panel into this wizard for you to use when specifying operation-specific roles.

4. Type a role name in the Default Role field, or select the one provided in the drop-down list. If you do not specify a role name for an operation on the next panel, the value you specify here will be assigned by default.
5. Accept the default save location provided for this ACL list, or type a new one in its place. You can click **Browse** to navigate to a different save location, if you prefer.
6. Click Next to display the Define ACL Operations panel, as shown in Figure 54.

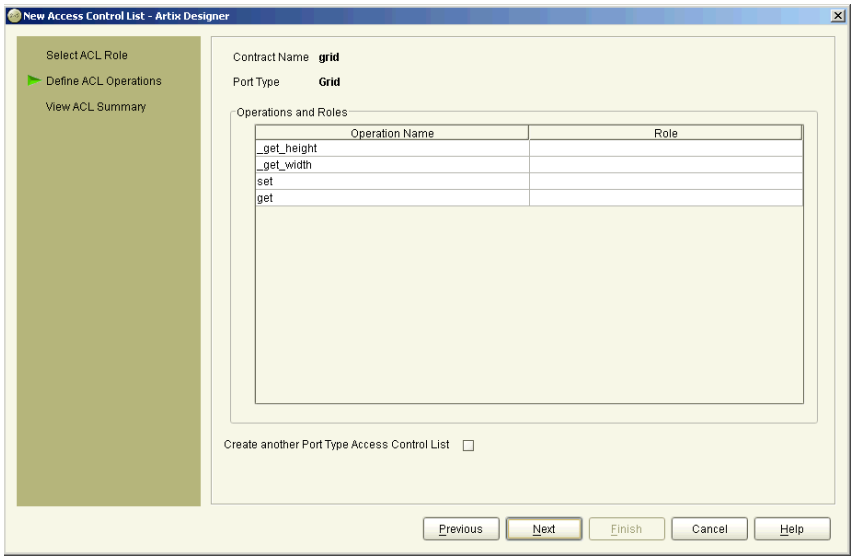

Figure 54: *New ACL wizard—Define ACL Operations panel*

7. Click in a cell in the Role column to assign a role to that operation. You can select roles from the drop-down with the cell, or type new roles into the cell. You can add multiple roles to an operation as long as you separate them with a comma.
8. Click the check box provided if you want to create another ACL - when you click **Next** after clicking the check box the ACL you just created will be saved and you will return to the first panel where you can repeat the process.

9. Otherwise, click **Next** to display the View ACL Summary panel, as shown in Figure 55.

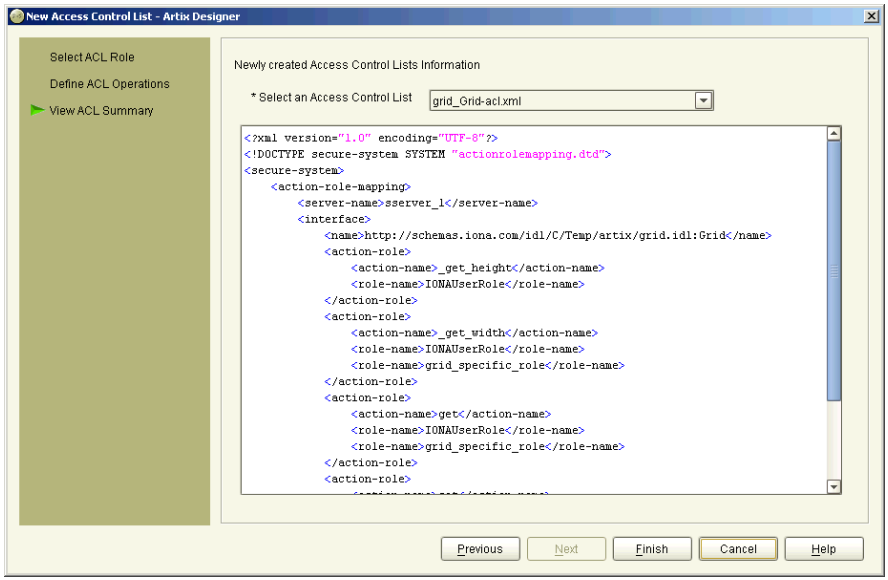

Figure 55: *New ACL Wizard—View ACL Summary panel*

10. This panel displays a summary of the ACL you have just created. To view an ACL created earlier, select it from the drop-down list provided.
11. Click **Finish** to close this wizard and return to the Artix Designer.

Creating Resources from a File/URL

Overview

If you don't want to create your resource from scratch, you might be able to base it on an existing URL or File. You have four options:

- URL - you can use WSDL located at a URL address. For more information, see "Using a File or a URL to create a Resource" on page 87.
- WSDL - if you have some existing WSDL, you can import this into Artix and use it as is, or edit it to change its components. For more information, see "Using a File or a URL to create a Resource" on page 87.
- XSD - You can create a resource based on an existing schema file. For more information, see "Using a File or a URL to create a Resource" on page 87.
- IDL - If you are starting from a CORBA server or client, Artix can generate the logical portion of the WSDL contract from IDL, automatically adding the required CORBA-specific entries and namespaces. For more information, see "Using IDL to create a Resource" on page 89.

  The IDL compiler also generates the binding information required to format the operations specified in the IDL. However, since port information is specific to the deployment environment, the port information is left blank, and you need to separately define a port using the Services wizard - "Adding Services" on page 139 for help with this task.

Using a File or a URL to create a Resource

Procedure

To use an existing WSDL or XSD file as the basis for your contract:

1. Select either the Shared Resources folder or a Collection from the Designer Tree.
2. Select **File | New | Resource** from the menu bar to display the New Resource dialog, as shown in Figure 58.

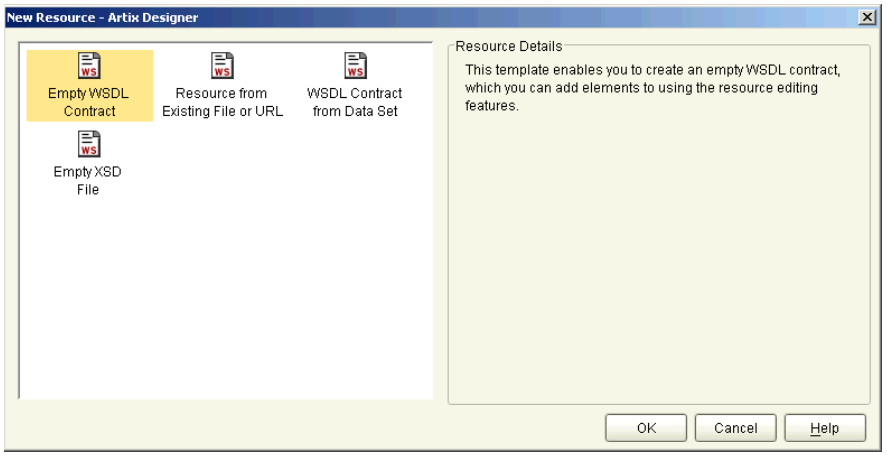

Figure 56: *New Resource dialog*

3. Select the **Resource from Existing File/URL** icon and click **OK** to display the New Resource from File/URL dialog, as shown in Figure 57.

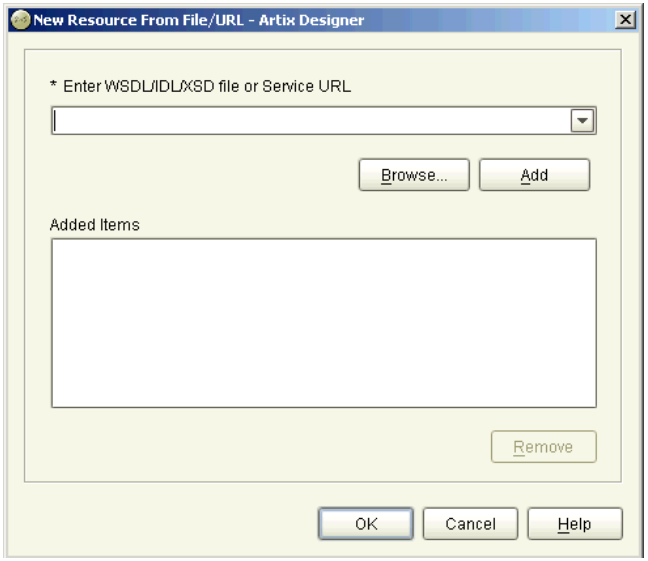

Figure 57: *New Resource from File/URL dialog*

4. Either type the URL address, or click **Browse** to locate the WSDL or XSD file.
5. Click **Add** to move this resource to the **Added Items** list. Repeat **steps 3 - 5** to add as many more WSDL or XSD resources as you like.
6. Click **OK** to close this dialog and return to the Artix Designer. One contract will be listed under the selected collection for each resource added or referenced.

Using IDL to create a Resource

Procedure

To use an IDL file as the basis for your resource:

1. Select either the Shared Resources folder or a Collection from the Designer Tree.
2. Select **New Resource** from the **File** menu to display the New Resource selection panel, as shown in Figure 58.

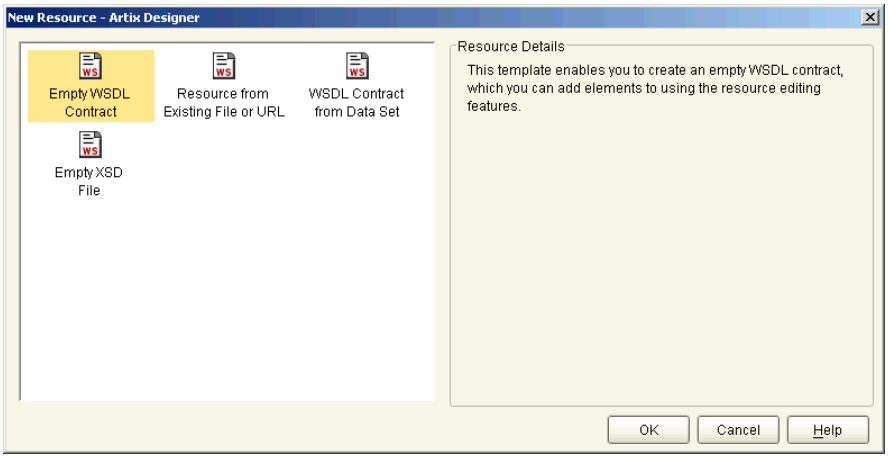

Figure 58: *New Resource dialog*

3. Select the **Resource from Existing File/URL** icon and click **OK** to display the New Resource from File/URL dialog, as shown in Figure 59.

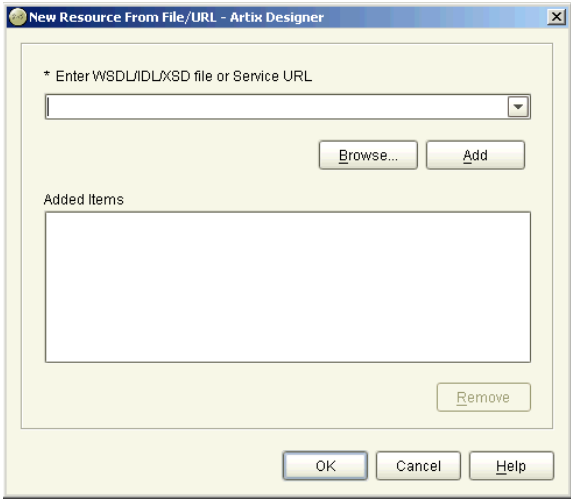

Figure 59: *New Resource from File/URL dialog*

4. Click **Browse** to locate the IDL file you want to use as the resource for your Artix contract.

5. Click **Add** to move this file to the Added Items list and display the IDL Compiler Options dialog, as shown in Figure 60.

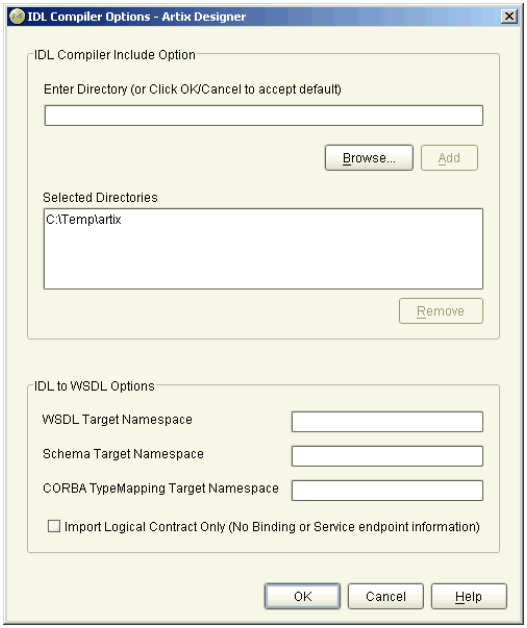

Figure 60: *IDL Compiler Options dialog*

6. Enter the names of the directories to search for included IDL files.
7. Click **Add** to add a directory to the list. Selecting a directory and clicking **Remove** will delete it from the list.
8. Add values to each of the namespace fields:
   - WSDL Target Namespace - the name the IDL Compiler will set for the *targetNamespace* value in the WSDL
   - Schema Target Namespace - the name the IDL compiler will set for the *targetNamespace* value in the Schema
   - CORBA TypeMapping Target Namespace - the name the IDL compiler will set for the CORBA *targetNamespace*

   If you do not set values for these fields, defaults will be assumed.

9. If you only wish to generate the logical portion of the contract select the **Logical Contract Only** check box.

   Note: If this option is selected the generated contracts will not contain any binding, CORBA typemap, or transport information.

10. Click **OK** to close this dialog and return to the New Resource from File/URL dialog
11. Repeat **steps 4 - 10** until you have added all of the IDL resources to import.
12. Click **OK** to close this dialog and return to the Artix Designer. One resource will be listed under the selected collection for each IDL file imported. The resources will include a CORBA binding (unless you specified in **Step 9** to create only the Logical Contract), a CORBA type map, and a CORBA port description.

Deploying a service with the CORBA port

You need to add location information to the CORBA port before you can deploy a service using the CORBA port. For more information, see "Adding a CORBA Port" on page 143.

For information about deploying Artix solutions, see "Deployment" on page 177.

Creating Contracts from Data Sets

Overview

The third way you can create new contracts is by basing them on a data set. Examples of this include:

- Defining fixed data
- Using an existing COBOL Copybook to define the fixed data
- Defining tagged data

When you create a contract in this way, you're actually also creating the associated binding at the same time. When creating contracts using the other methods described in this chapter, the binding definition is a separate step.

Procedure

To create a contract from a data set:

1. Select **New Resource** from the **File** menu to display the New Resource dialog, as shown in Figure 61.

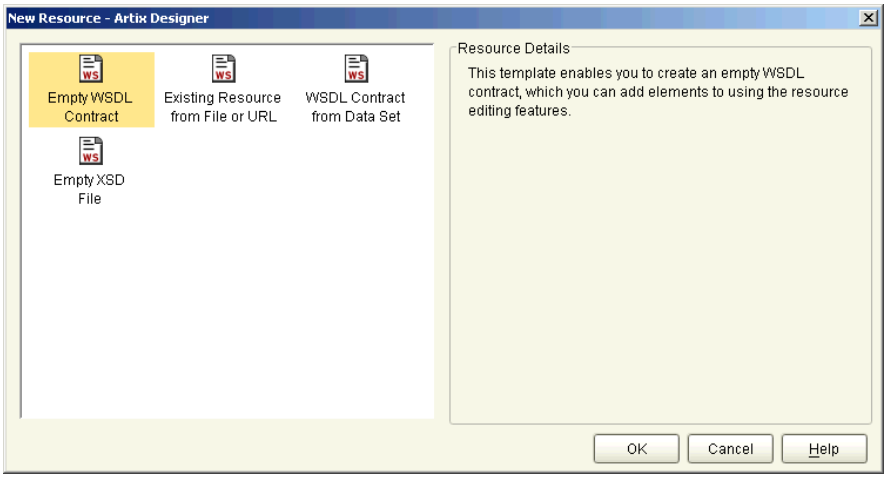

Figure 61: *New Resource dialog*

2. Select the **New Contract from Data Set** icon, and click **OK** to display the New Contract from Data Set wizard, as shown in Figure 62.

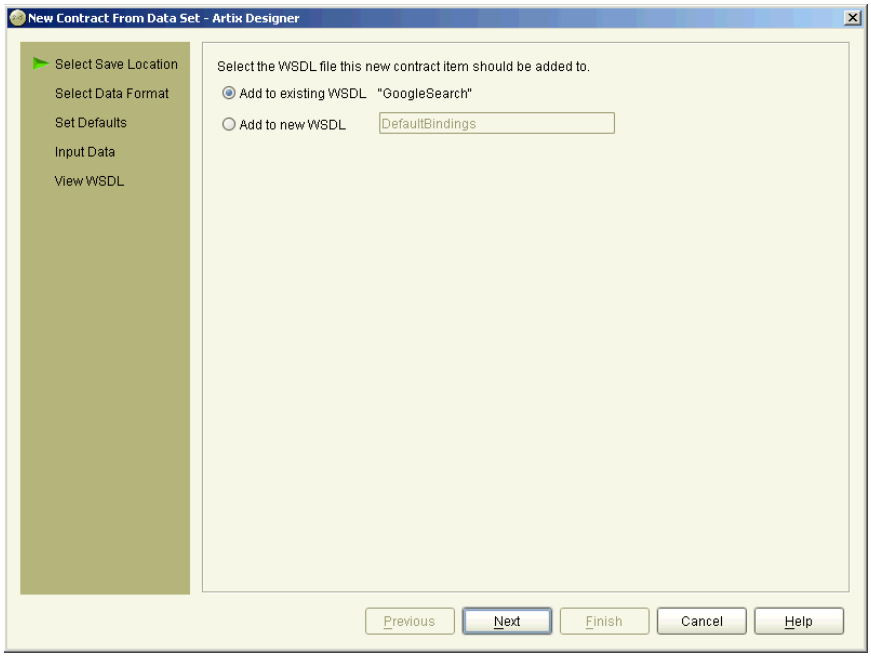

Figure 62: *New Contract from Data Set wizard*

3. Enter a name for the WSDL that will contain the new binding, or accept the default provided.
4. Click **Next** to display the Data Format panel.

Now you need to turn to the relevant page, depending on what type of contract and binding you are creating. You can create:

- A contract containing a fixed binding - see page 95
- A contract containing a fixed binding from a CCB - see page 98
- A contract containing a tagged binding - see page 101

Creating a Contract Containing a Fixed Binding

Overview

Many applications send data in fixed length records. For example, COBOL applications often send fixed record data over WebSphere MQ. Artix provides a binding that maps logical messages to concrete fixed record length messages. The fixed binding allows you to specify attributes such as encoding style, justification, and padding characters.

Procedure

To add a contract containing a Fixed binding:

1. At the Data Format panel, select **Fixed**.
2. Click **Next** to display the Set Defaults panel, as shown in Figure 63.

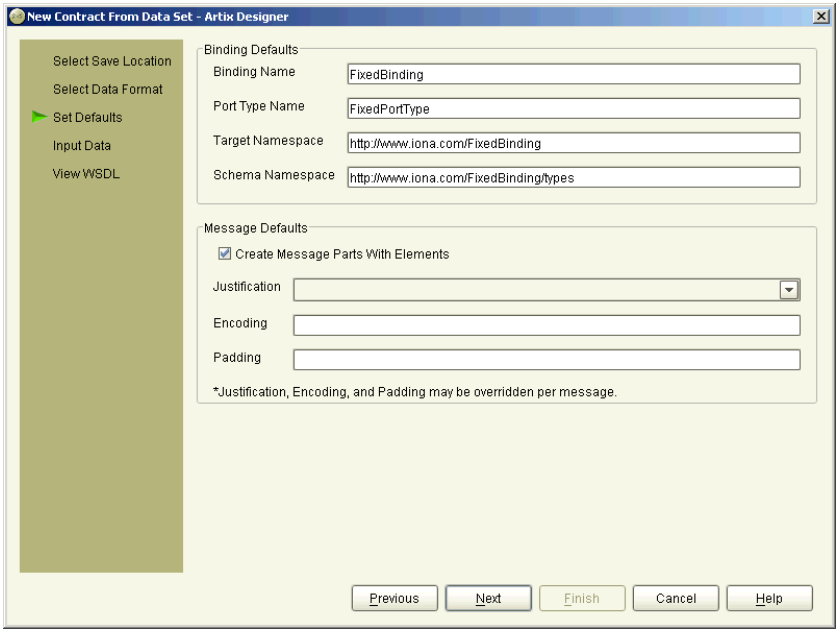

Figure 63: *New Contract from Data Set wizard—Set Fixed Defaults panel*

3. Under the **Binding Defaults**, enter a name for the binding being created in this new contract, or accept the default provided.
4. Enter a name for the new port type, or accept the default provided.

5. The **Target Namespace** and **Schema Namespace** values default to whatever is specified by the platform. Unless absolutely necessary, it is recommended that you do not change these.
6. Under the **Message Defaults,** check the box provided if you want to create your message parts as elements rather than types.
7. Select a justification value from the drop-down list. Options are **Left** and **Right**.
8. Enter an encoding value**.** Valid options are **UTF-8** and **UTF-16**.
9. Enter a value in the **Padding** field, if required. This is a character string to be used to fill unused space in the message field. You can use any character, or combination of characters, that you like.
10. Click **Next** to display the Input Data panel, as shown in Figure 64.

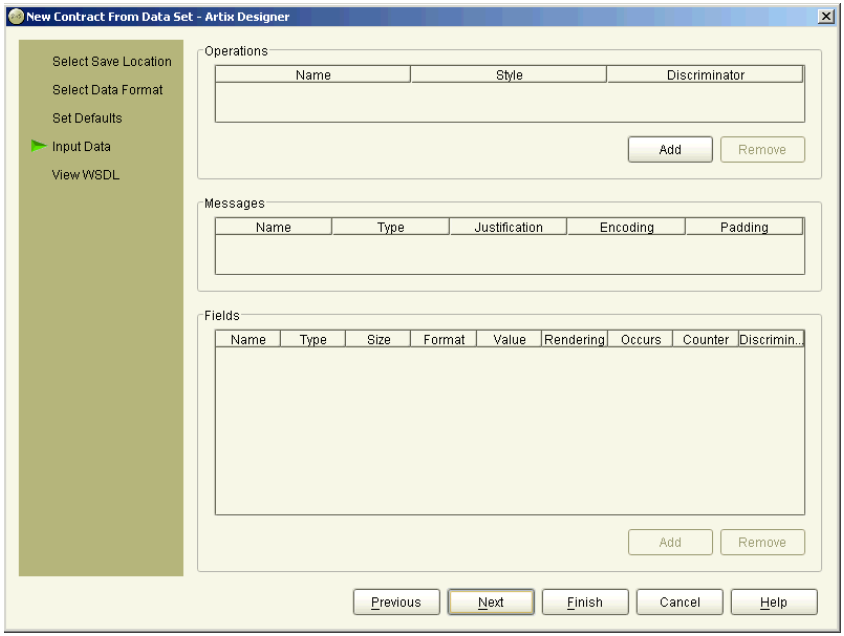

Figure 64: *New Contract from Data Set wizard—Input Data panel (Fixed)*

11. Click **Add** to create a new Operation.
12. Enter a name for the Operation, or accept the one provided.

13. Change the Operation **style** by clicking on the default Style value.
14. You can add a **discriminator** to filter the operations by adding one to the Discriminator cell for the new Operation.
15. Under **Messages** enter values for the attributes for the messages that have been created for the Operation.
16. Click **Add** to add fields to your messages and select each of the available cells to enter attributes for the fields as required.

    Click on the Type cell to change the field type. You can then add subsequent fields to the each of the field types.

    Message parts can be fields, enumerations, sequences, or choices.
17. When you have finished adding objects click **Finish** to create the contract with the fixed record binding, as shown in Figure 65.

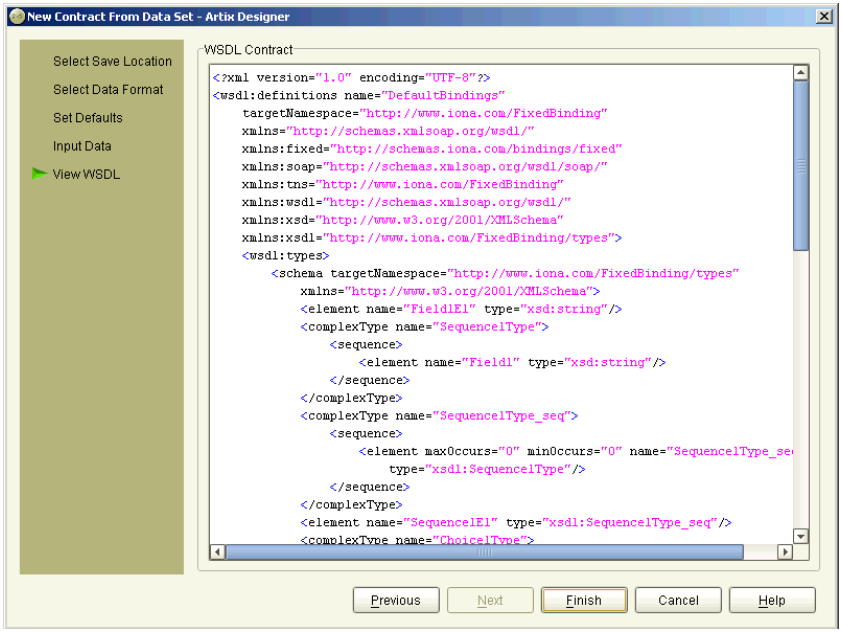

Figure 65: *New Contract from Data Set—Summary panel*

Creating a Contract Containing a Fixed Binding from a COBOL Copybook

Overview

The other way to create a contract containing a fixed binding is to base the messages in that binding on an existing COBOL Copybook. Your CCB can contain one or more messages - at the time that you associate each fixed message with a message from the CCB, you'll be asked to specify the message to use.

Procedure

To add a contract containing a Fixed binding from a COBOL Copybook:

1. At the Data Format panel, select **Fixed**.
2. Click **Next** to display the Set Defaults panel, as shown in Figure 66.

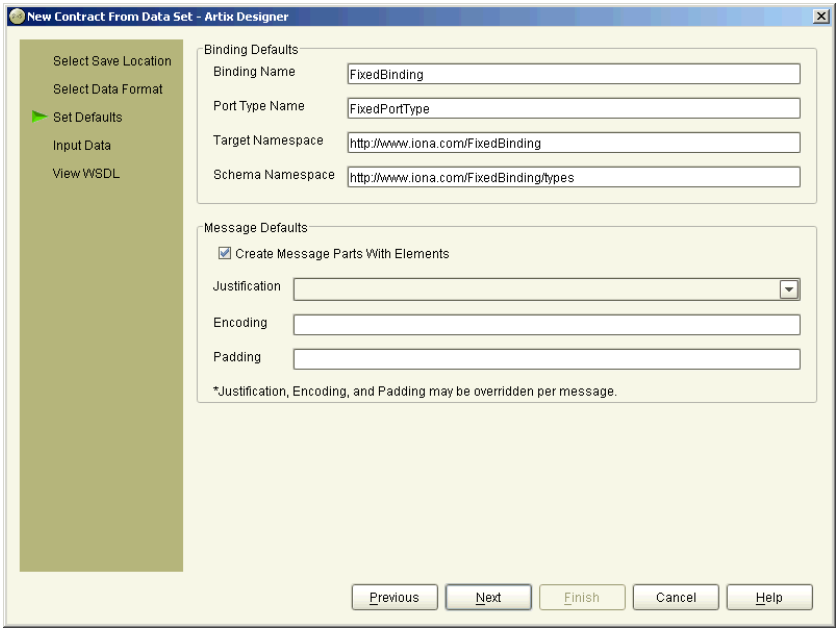

Figure 66: *New Contract from Data Set wizard—Set Fixed Defaults (CCB)*

3. Under the **Binding Defaults**, enter a name for the binding being created in this new contract, or accept the default provided.
4. Enter a name for the new port type, or accept the default provided.

5. The **Target Namespace** and **Schema Namespace** values default to whatever is specified by the platform. Unless absolutely necessary, it is recommended that you do not change these.
6. Under the **Message Defaults,** check the box provided if you want to create your message parts as elements rather than types.
7. Select a justification value from the drop-down list. Values are **Left** and **Right**.
8. Enter an encoding value**.** Valid options are **UTF-8** and **UTF-16**.
9. Enter a value in the **Padding** field, if required. This is any character string to be used to fill unused space in the message field.
10. Click Next to display the Input Data panel, as shown in Figure 64.

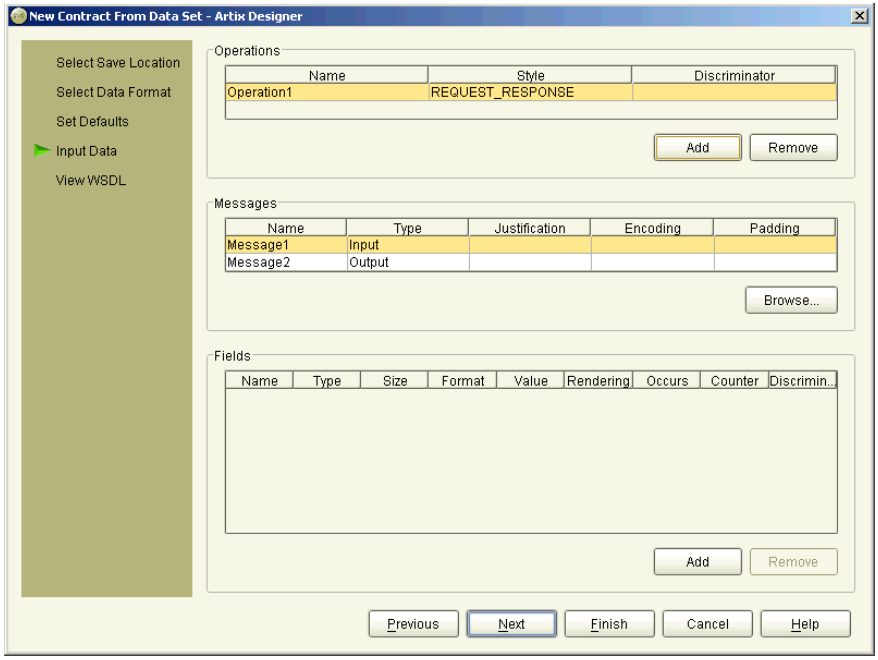

Figure 67: *New Contract from Data Set wizard—Input Data panel (CCB)*

11. Click **Add** to create a new Operation. (In the example shown, this step has already been performed so that the Browse button described in step 15 could be enabled.)
12. Enter a name for the Operation, or accept the one provided.
13. Change the Operation **style** by clicking on the default Style value.
14. You can add a **discriminator** to filter the operations by adding one of the Discriminator cell for the new Operation.
15. Under **Messages** enter values for the attributes for the messages that have been created for the Operation. To use the details from your COBOL Copybook, click the **Browse** button.

    This will invoke a file chooser, from where you can navigate to your COBOL Copybook. When you select one, and click OK, Artix will do one of two things:

    - If the COBOL Copybook contains only one message, the associated fields on the Input Data panel will be populated.
    - If the COBOL Copybook contains more that one message, you will see an intermediary dialog from where you can select which message to associate with the fixed message in the Input Data panel.
16. You can edit any of the fields that are populated for this message from the COBOL Copybook by clicking on the relevant cell.
17. Click **Add** to add extra fields to your messages as required.

    Click on the Type cell to change the field type. You can then add subsequent fields to the each of the field types.

    Each message part can be either a field, an enumeration, a sequence, or a choice.
18. When you have finished adding objects click **Finish** to create the contract with the fixed record binding, as shown in Figure 65.

Creating a Contract Containing a Tagged Binding

Overview

The tagged data format supports applications that use self-describing, or delimited, messages to communicate. Artix can read tagged data and write it out in any supported data format. Similarly, Artix is capable of converting a message from any of its supported data formats into a self-describing or tagged data message.

Procedure

To add a contract containing a Tagged binding:

1. At the Data Format panel, select **Tagged**.
2. Click **Next** to display the Set Defaults panel, as shown in Figure 68.

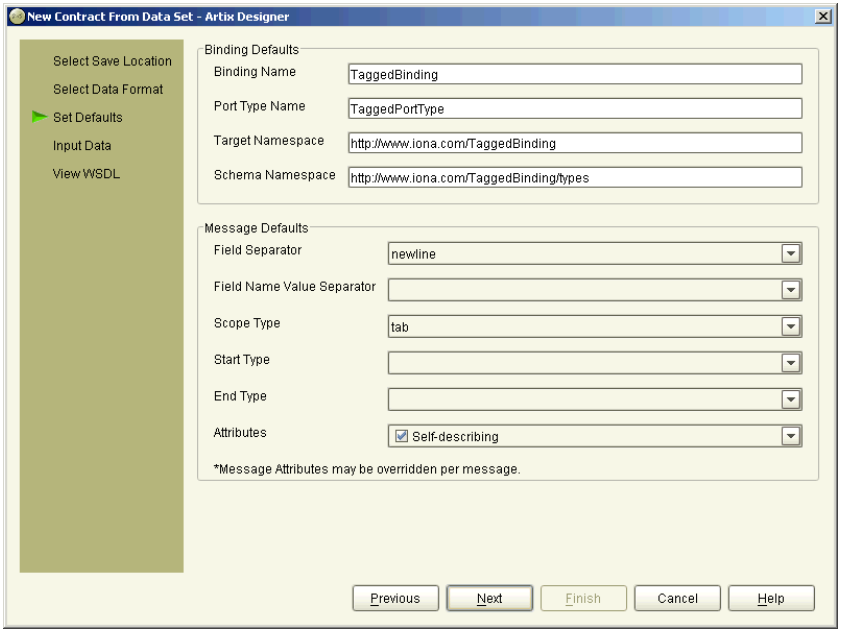

Figure 68: *New Contract from Data Set—Set Tagged Defaults panel*

3. Under the **Binding Defaults**, enter a name for the binding being created in this new contract, or accept the default provided.
4. Enter a name for the new port type, or accept the default provided.

5. The **Target Namespace** and **Schema Namespace** values default to whatever is specified by the platform. Unless absolutely necessary, it is recommended that you do not change these.
6. Under the **Message Defaults**, select a value for the **Field Separator**, or accept the default provided.
7. Select a value for the **Field Name Value Separator**.
8. Select a value for the **Scope Type**, or accept the default provided.
9. Select a value for the **Start Type**.
10. Select a value for the **End Type**.
11. Select the **Attributes** you want to apply as defaults to your messages.

    Note: See the online help for additional information on all the optional settings.
12. Click **Next** to display the Input Data panel, as shown in Figure 69.

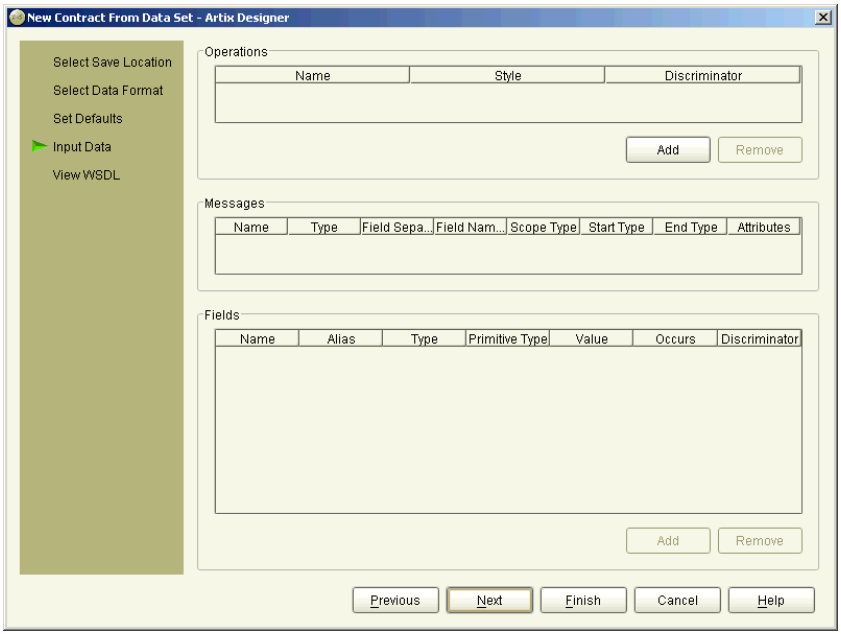

Figure 69: *New Contract from Data Set wizard—Input Data panel (Tagged)*

13. Click **Add** to create a new Operation.
14. Enter a name for the Operation, or accept the one provided.
15. Change the Operation **style** by clicking on the default Style value.
16. You can add a **discriminator** to filter the operations by adding one to the Discriminator cell for the new Operation.
17. Under **Messages** enter values for the attributes for the messages that have been created for the Operation. The values you specified on the Defaults panel are displayed here, but can be over-written at the individual message level if required.
18. Click **Add** to add fields to your messages and select each of the available cells to enter attributes for the fields as required.

    Click on the Type cell to change the field type. You can then add subsequent fields to the each of the field types. Messages can be a field, an enumeration, a sequence, or a choice.
19. When you have finished adding objects click **Finish** to create the contract with the tagged record binding, as shown in Figure 70.

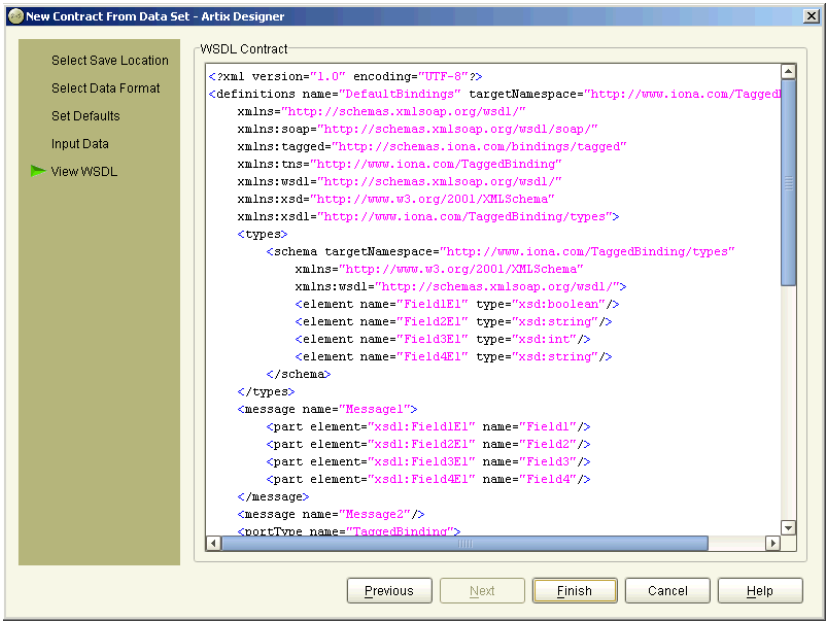

Figure 70: *New Contract from Data Set—Summary panel (Tagged)*

Creating an XSD Schema

Overview

The first thing you need to do is create an XSD contract shell. Depending on which mode of the Designer you are working in (Deployer or Editor), the steps you follow to do this will be slightly different.

If you are working in Editor mode, select **New | XSD File** to display the New Contract dialog, as shown in Figure 72 on page 106.

If you are working in Deployer mode:

1. Select either the Shared Resources folder or a Collection from the Designer Tree.
2. Select **File | New | Resource** from the **File** menu to display the New Resource dialog, as shown in Figure 71.

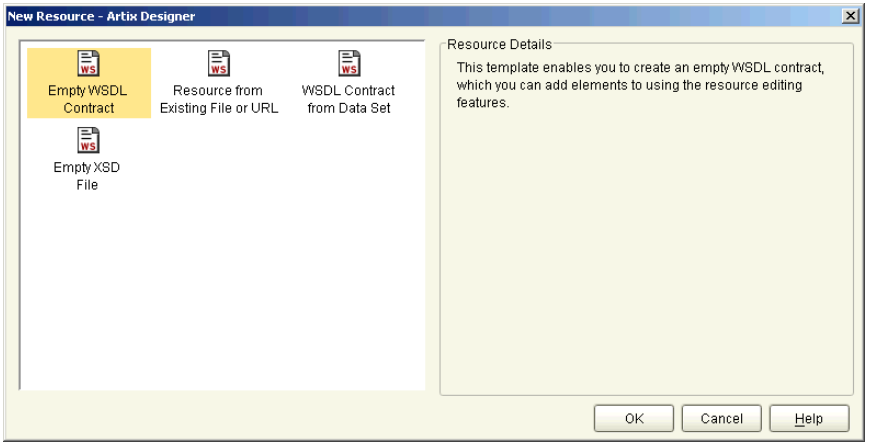

Figure 71: *New Resource dialog*

3. Select the **Empty** XSD File icon and click **OK** to display the New Schema dialog, as shown in Figure 72.

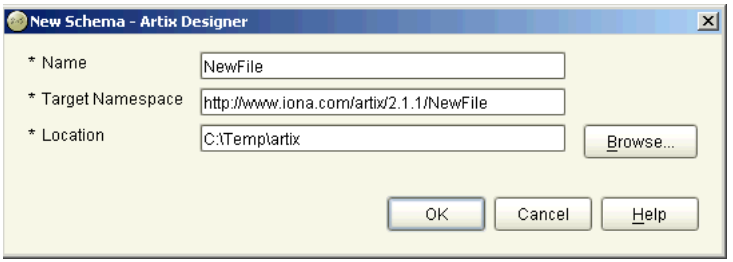

Figure 72: *New Schema dialog*

4. Enter a name in the **Name** field, or accept the default provided.
5. Enter a value in the **Target Namespace** field, or accept the default provided.
6. Enter a value in the Location field, or accept the default provided. Click **Browse** to navigate to a different save location if you like.
7. Click **OK** to close this dialog and return to the Artix Designer. Your new schema will be shown, and you can now add types to it using the procedure documented in "Adding Types" on page 60.

Editing Resources

Overview

The Artix Designer provides edit dialogs for all of the resource components, from which you can edit most of the properties for your resource. This section walks you through that process.

You can access the edit dialog for a resource component either through the menu bar or through the Resource Navigator (diagram view).

To access the edit dialog for one of the components:

1. While in the diagram view of the **Resource Navigator**, select **Resource | Edit | <component>**, where <component> is the name of the element you want to work with.

   The Edit dialog for that component is displayed.

2. Alternatively, you can right-click on the component name and select **Edit**, which will also display the Edit dialog.

Editing in the text view

If you prefer, you can use the Text view of the resource to hand-edit the XML. Be aware however, that any changes you make to the XML could invalidate the contract. If this happens, you will only be able to view the contract in the Text view - the diagram view will be disabled as the model cannot be generated with invalid XML.

The Artix Designer provides you with tools to try to help you avoid invalidating your XML, or to identify and rectify errors. Every time you make a change and click **Apply Edits**, the Designer displays any errors in the error bar at the bottom of the Text view, as shown in Figure 73.

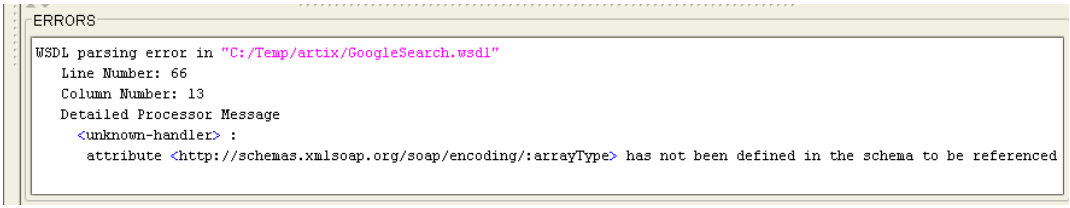

Figure 73: *XML Error Indicator*

Editing Types

This process is the same whether you're working with Types contained in a contract or in a schema.

You can edit a type by selecting **Resource | Edit | Types**, to display the Edit Types dialog as shown in Figure 74.

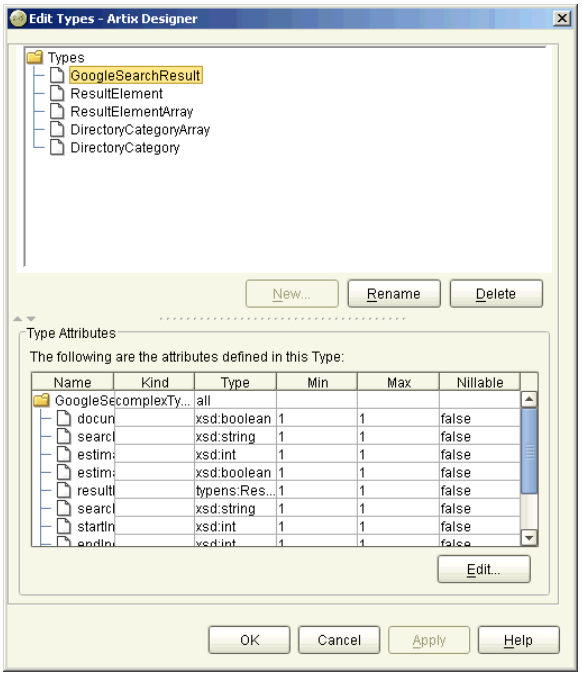

Figure 74: *Edit Types dialog*

At the Edit Types dialog, all of your types and their associated attributes are listed in the top half of the dialog. From here you can:

- Rename a type or an attribute by selecting it and clicking **Rename**
- Delete a type or an attribute by selecting it and clicking **Delete**.
- Add a new type by clicking **Add** to display the New Type wizard, as described in "Adding Types" on page 60.

Editing attribute properties

When you select a type in the top of this dialog, the type attributes are displayed in the panel at the bottom of the dialog.

To edit any of the Type Attributes, click the **Edit** button to display the Edit Type Attributes dialog, as shown in Figure 75.

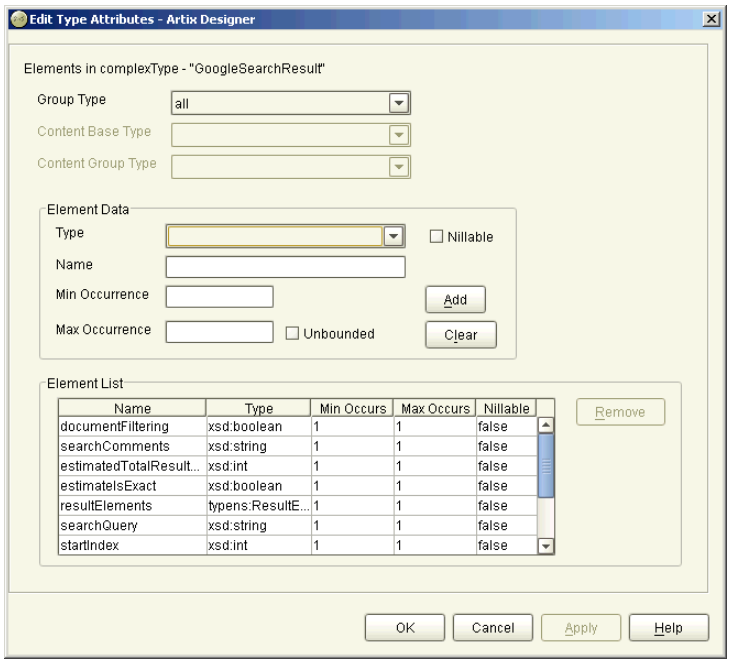

Figure 75: *Edit Type Attributes dialog*

To change values of attributes in this dialog, click on the item you want to change in the Element List - its details will be populated into the Element Data fields. Make your changes and click **Update**.

When you have finished making your changes, click **Apply** to update the attribute, and **OK** to close the wizard and return to the Edit Types dialog, where your changes are displayed in the Type Attributes panel.

Click **OK** to close this dialog and return to the Artix Designer.

Editing Messages

You can edit a type by selecting **Resource | Edit | Messages**, to display the Edit Messages dialog as shown in Figure 76.

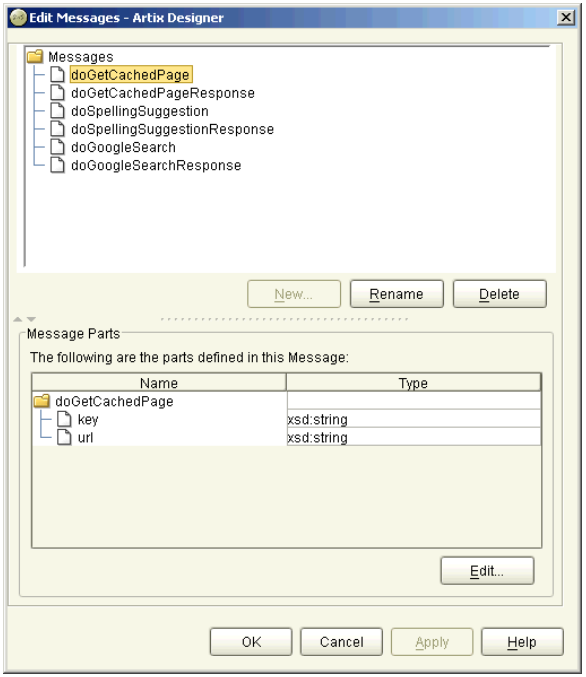

Figure 76: *Edit Messages dialog*

At the Edit Messages dialog, all of your messages and their associated parts are listed in the top half of the dialog. From here you can:

- Rename a message or a part by selecting it and clicking **Rename**
- Delete a message or a part by selecting it and clicking **Delete**.
- Add a new message by clicking **Add** to display the New Message wizard, as described in "Adding Messages" on page 73.

Editing message parts

When you select a message in the top of this dialog, the message parts are displayed in the panel at the bottom of the dialog.

To edit any of the message parts, click the **Edit** button to display the Edit Message Parts dialog, as shown in Figure 75.

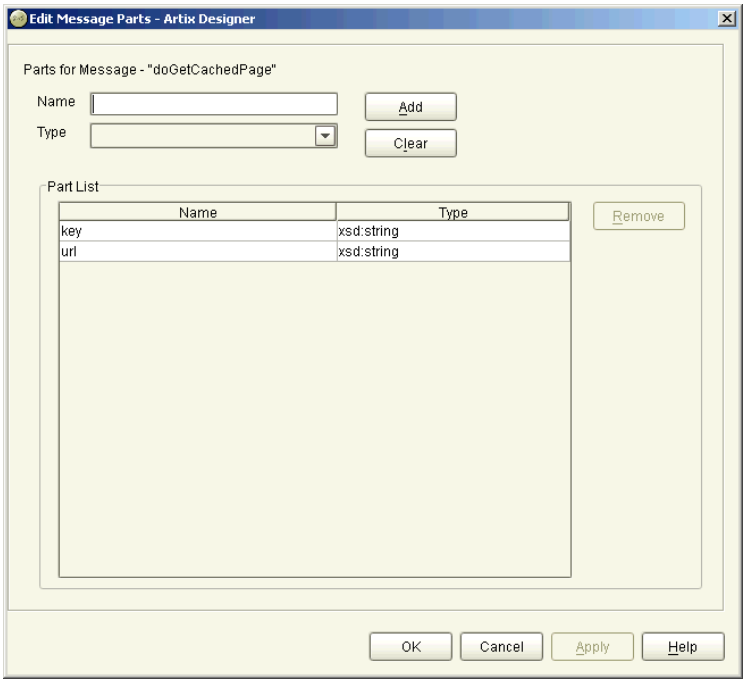

Figure 77: *Edit Message Parts dialog*

To change values of parts in this dialog, click on the item you want to change in the Part List - its details will be populated into the Parts fields. Make your changes and click **Update**.

When you have finished making your changes, click **Apply** to update the part, and **OK** to close the wizard and return to the Edit Messages dialog, where your changes are displayed in the Message Parts panel.

Click **OK** to close this dialog and return to the Artix Designer.

Editing Port Types

You can edit a port type by selecting **Resource | Edit | Port Types**, to display the Edit Port Types dialog as shown in Figure 78.

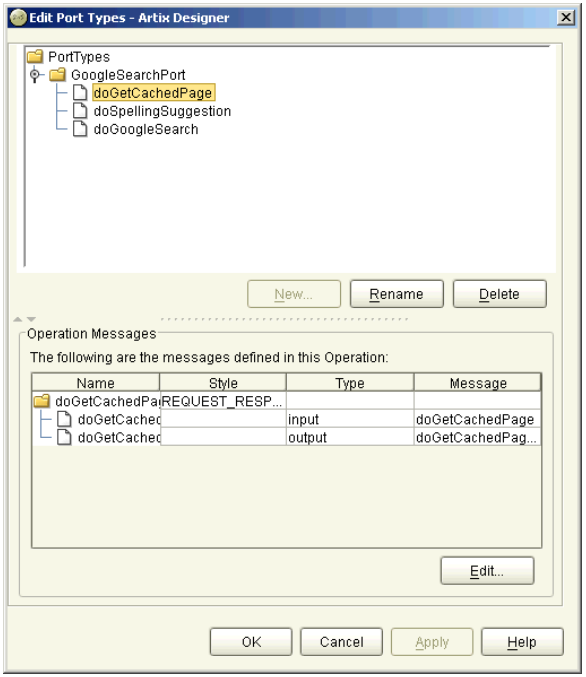

Figure 78: *Edit Port Types dialog*

At the Edit Port Types dialog, all of your port types and their associated operation messages are listed in the top half of the dialog. From here you can:

- Rename a port type or an operation message by selecting it and clicking **Rename**
- Delete a port type or an operation message by selecting it and clicking **Delete**.
- Add a new port type by clicking **Add** to display the New Port Type wizard, as described in "Adding Port Types" on page 77.

Editing operation messages

When you select a port type in the top of this dialog, the operation messages are displayed in the panel at the bottom of the dialog.

To edit any of the operation messages, click the **Edit** button to display the Edit Operation Messages dialog, as shown in Figure 79.

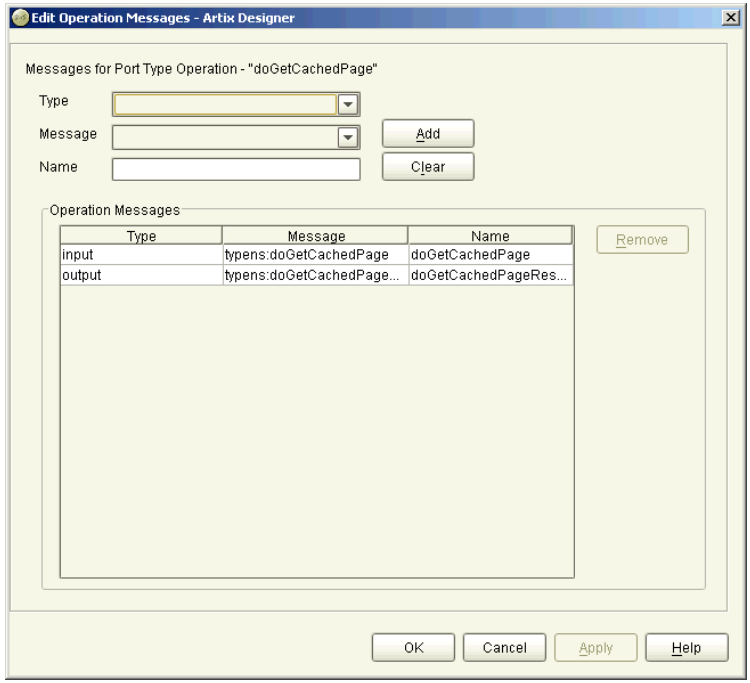

Figure 79: *Edit Type Attributes dialog*

To change values of operation messages in this dialog, click on the item you want to change in the Operation Messages list - its details will be populated into the Messages fields. Make your changes and click **Update**.

When you have finished making your changes, click **Apply** to update the operation message, and **OK** to close the wizard and return to the Edit Port Types dialog, where your changes are displayed in the Operation Messages panel.

Click **OK** to close this dialog and return to the Artix Designer.

CHAPTER 5

Adding Bindings

Bindings contain information used by Artix at runtime to reformat data between endpoints, enabling it to be understood by the target service.

In this chapter

This chapter discusses the following topics:

What is a Binding?

Overview

If you are exposing an existing service using a new transport or payload format, you need to add the mapping of the service's data and operations to the new payload format and transport. To do this, you add one or more bindings to your services. The information you include in the binding is used by Artix at runtime to reformat the data on the wire and thus make it understandable by the target service.

The New Binding wizard walks you through the generation of a binding based on your existing contract. It then adds the binding to the contract.

Artix binding types

Artix provides support for several binding types. They are accessed via two methods:

- From the New Binding wizard, which enables you to create the following binding types:
  - CORBA
  - Fixed
  - SOAP
  - XML
  - Tagged
- From the Contract From Data Set wizard, which enables you to create a new contract that also includes a binding of one of the following types:
  - Fixed
  - Fixed, using data from an existing COBOL Copybook
  - Tagged

  These last three bindings can be created with or without an existing contract, and will create the binding and logical elements from input data.

  For more information about adding contracts with these bindings included, "Creating Contracts from Data Sets" on page 93.

Adding bindings via New Binding wizard

To add a binding to an Artix contract using the New Binding wizard:

1. From the Designer Tree, select the contract to which you want to add the binding.
2. Select **Resource | New | Binding** from the menu bar to display the New Binding wizard, as shown in Figure 80.

   Note that your WSDL needs to contain at least one message and a port type before you can add a binding.

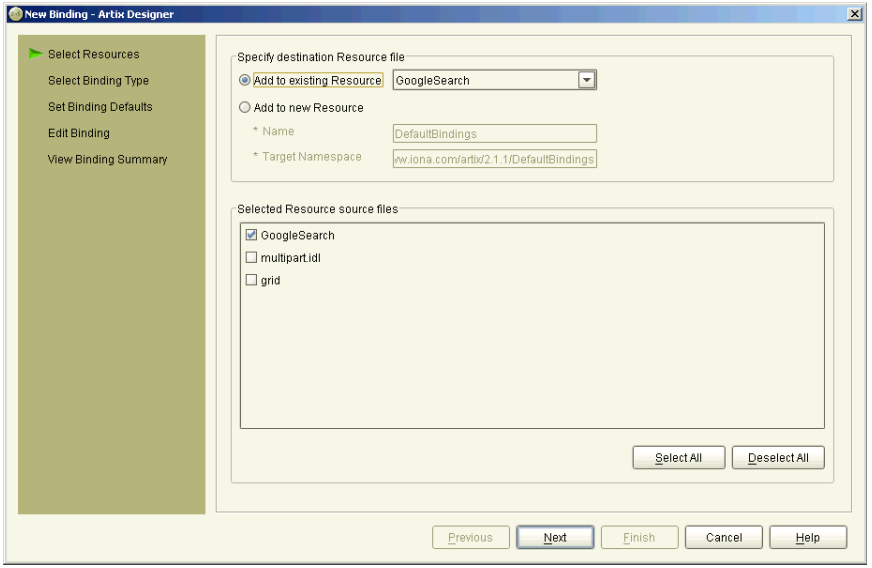

Figure 80: *New Binding wizard*

3. Select where to create the WSDL entry for the new binding.
   - **Add to existing WSDL** adds the binding information to the existing contract.
   - **Add to new WSDL** creates a new WSDL document that contains the binding information plus an import statement in the logical contract in which the binding is being created.

4. Select the resources from this collection that you want to use as the source for this new binding. If you selected a resource before invoking the New Binding wizard, that resource is selected by default. You can also select other resources to use as sources for this binding - this will give you more port types to choose from when setting the binding defaults later in this wizard.
5. Click **Next** to display the Binding Type panel.

Now you need to turn to the appropriate page for the type of binding you are creating:

- For a CORBA binding, see page 119
- For a Fixed binding, see page 123
- For a SOAP binding, see page 126
- For an XML binding, see page 132
- For a Tagged binding, see page 135

Adding a CORBA Binding

Overview

To ensure that messages are converted into a format that a CORBA application can understand, Artix contracts need to describe how data is mapped to CORBA data types.

Procedure

To add a CORBA binding to an Artix contract from the Binding Type panel:

1. Select **CORBA**, and then click **Next** to display the Binding Defaults panel, as shown in Figure 81.

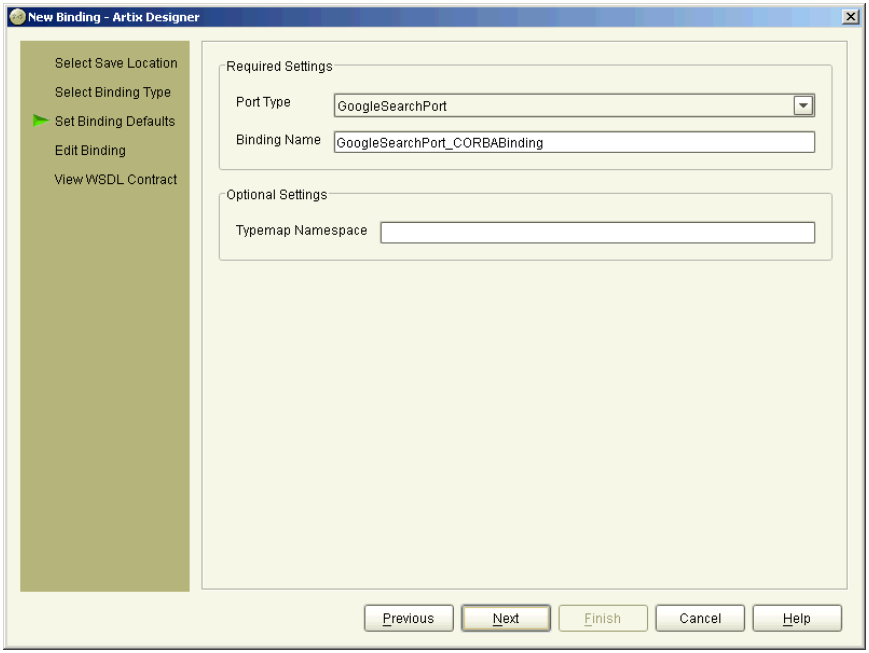

Figure 81: *New Binding wizard—CORBA Binding Defaults panel*

2. From the **Port Type** drop down list select the port type you want to map to the CORBA binding.

3. Enter a name for the new binding, or accept the default provided.
4. Enter a value for the Typemap Namespace if required (optional).
5. Click **Next** to display the Edit Binding panel, as shown in Figure 82, which displays the generated operations and CORBA types.

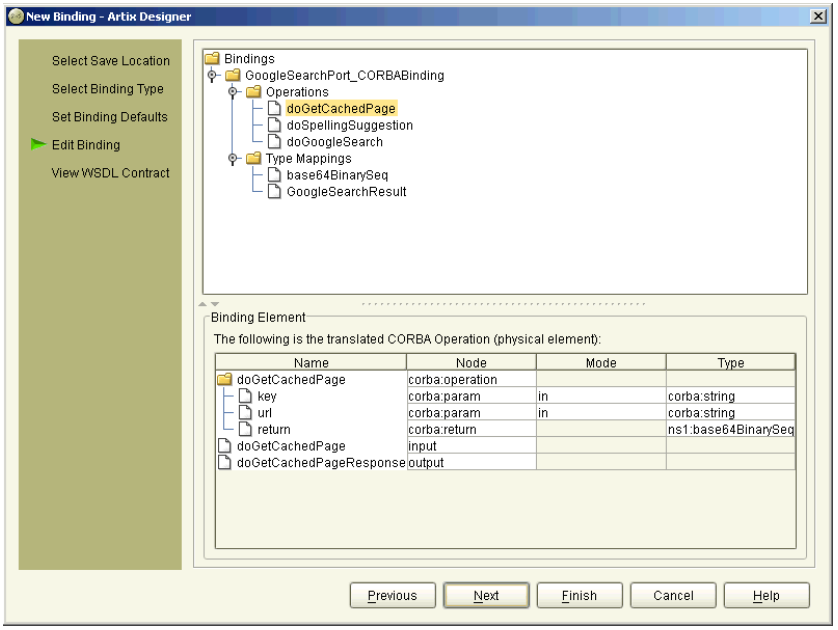

Figure 82: *New Binding wizard—Edit CORBA Binding panel*

6. Examine the different elements of the binding by selecting them from the tree at the top of the dialog.
7. If you like, you can change the name of the Binding. The attribute fields are read-only.
8. Click **Next** to display the Binding Summary panel, as shown in Figure 83.

9. Click **Finish** to close this wizard and return to the Artix Designer.

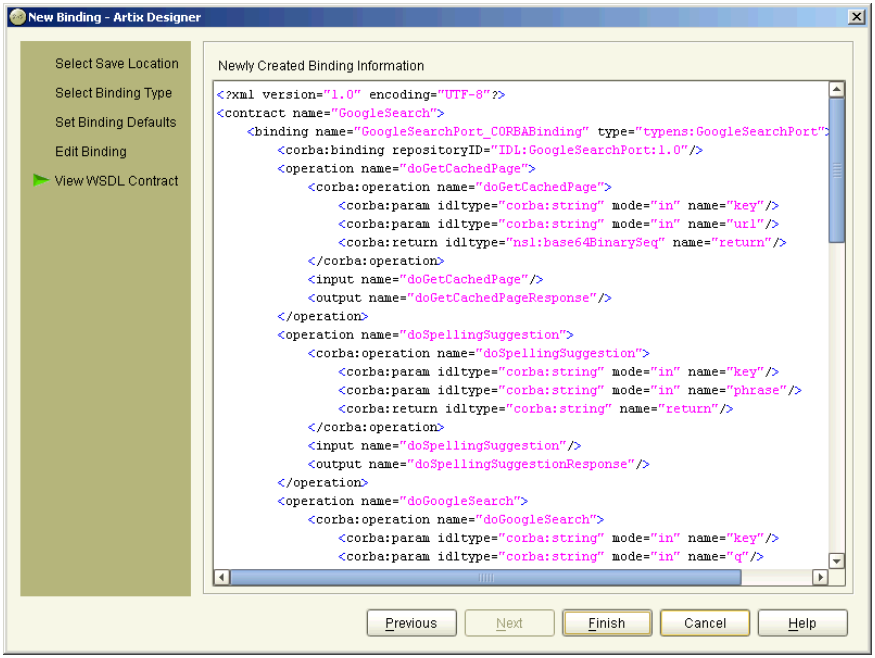

Figure 83: *New Binding wizard—CORBA Binding Summary panel*

When you have created the new CORBA binding, the contract describing the binding and the CORBA type map are added to the Designer Tree under the selected service. Note however, that this new contract **will not** contain a CORBA port description.

For details on adding a CORBA port description see "Adding a CORBA Port" on page 143.

Adding a CORBA Binding, Service, and Port at the Same Time

Overview

There is a smart-menu option you can use if you would like to create a CORBA binding, service, and port for a resource. It is called CORBA enabling, and you can access it either through the Resource menu or via the contextual menu after first selecting a resource in the Designer Tree.

Procedure

To create a CORBA binding, service, and port using the smart-menu option:

1. Select a resource from the Designer Tree, and select **Resource | CORBA Enable**, to display the CORBA Enable dialog as shown in Figure 84.

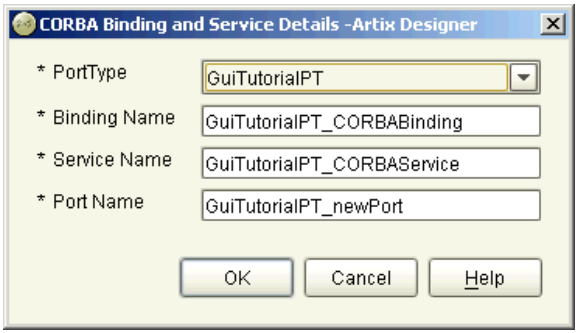

Figure 84: *CORBA Enable dialog*

2. Select the Port Type to use from the drop-down list provided. This list contains all port types that have been defined for this resource.
3. The next three fields, Binding Name, Service Name, and Port Name all contain default names for these elements. Either accept these defaults, or provide alternatives.
4. Click **OK** to close this dialog and return to the Artix Designer. The new binding, service, and port will be displayed in the relevant sections within the WSDL model diagram.

Adding a Fixed Binding

Procedure

To add a Fixed binding to an Artix contract from the Binding Type panel:

1. Select **Fixed**, and then click **Next** to display the Binding Defaults panel, as shown in Figure 85.

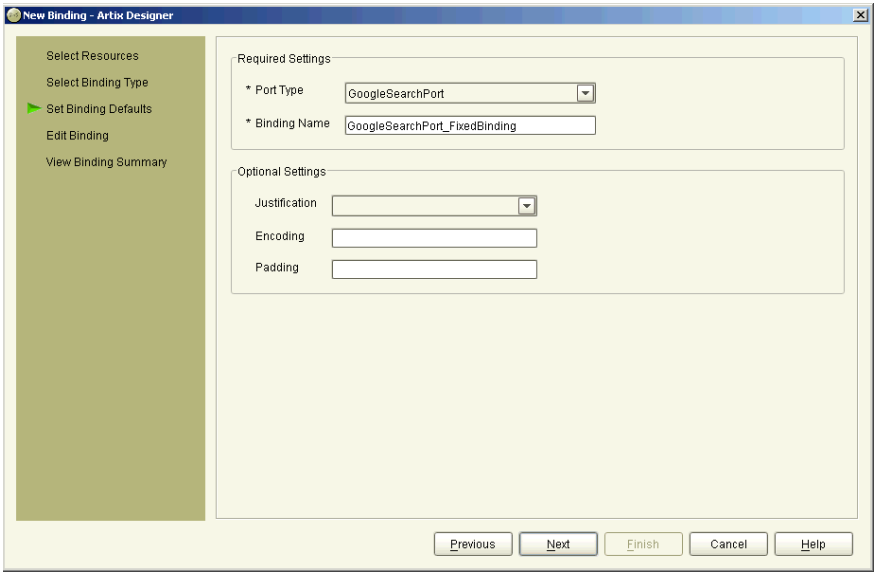

Figure 85: *Binding wizard—Fixed Binding Defaults*

2. From the **Port Type** drop down list select the port type you want to map to the Fixed binding.
3. Enter a name for the new binding, or accept the default provided.
4. Enter a value for the additional settings if required. They are:
   - Justification - the justification of the message data. Options are left and right.
   - Encoding - the encoding style for the message data. Examples are UTF-8 and UTF-16.

- Padding - a character string to be used to fill unused space in the message field.

5. Click **Next** to display the Edit Binding panel, as shown in Figure 82, which displays the generated operations and Fixed types.

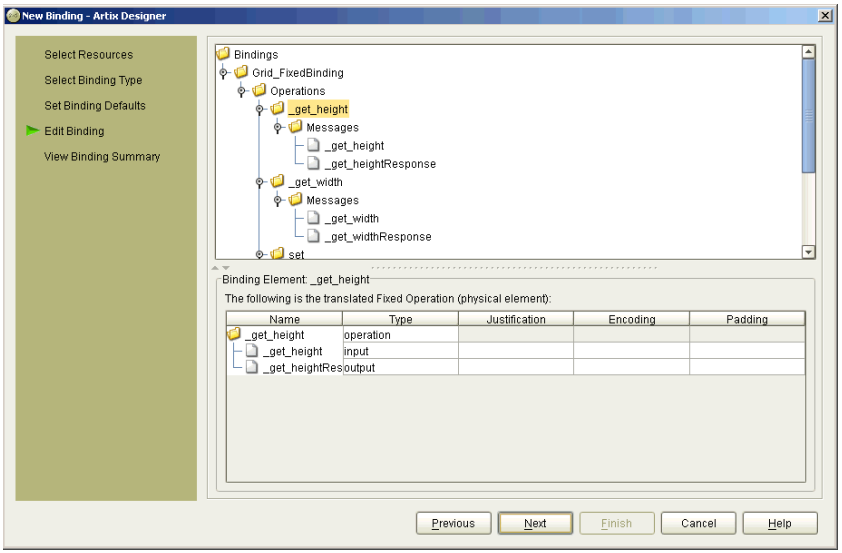

Figure 86: *New Binding wizard—Edit Fixed Binding panel*

6. Examine the different elements of the binding by selecting them from the tree at the top of the dialog.
7. If you like, you can change the name of the Binding. The attribute fields are read-only.

8. Click **Next** to display the Binding Summary panel, as shown in Figure 83.

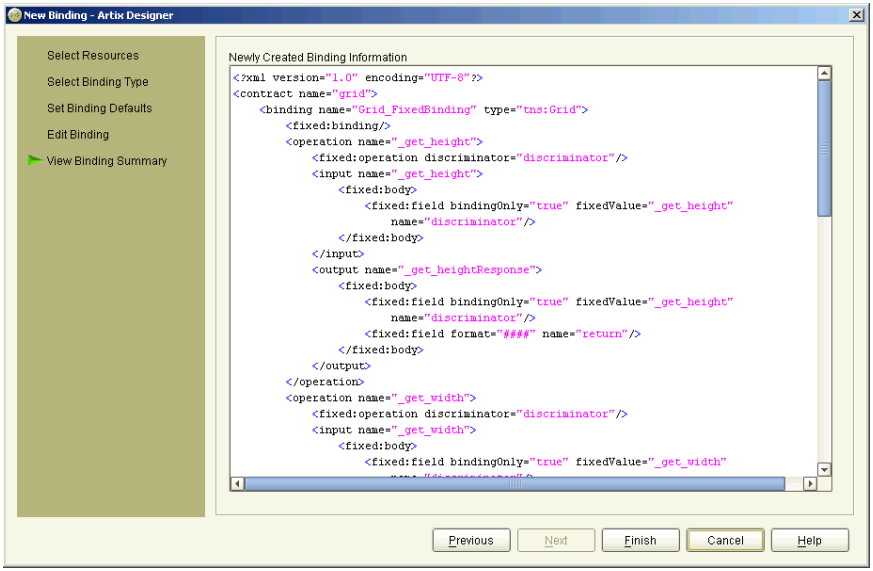

Figure 87: *New Binding wizard—Fixed Binding Summary panel*

9. Click **Finish** to close this wizard and return to the Artix Designer.

Adding a SOAP Binding

Overview

SOAP is termed a *messaging* protocol. It is a framework for transporting client request and server response messages in the form of XML documents over (usually) the HTTP transport.

Procedure

To add a SOAP binding to an Artix contract:

1. At the Binding Type panel, select **SOAP**.
2. Click **Next** to display the Binding Defaults panel, as shown in Figure 88.

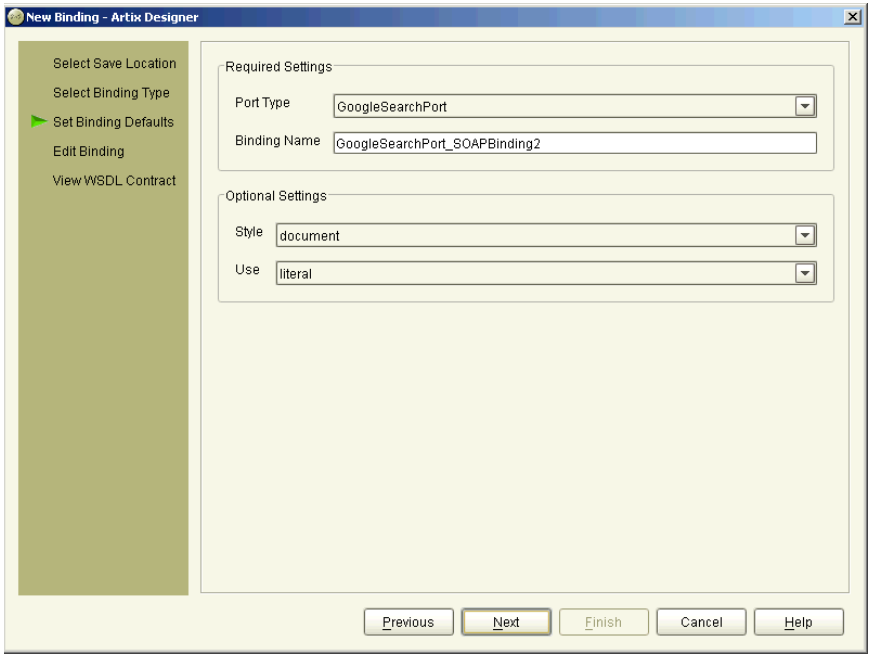

Figure 88: *New Binding wizard—SOAP Binding Defaults panel*

3. From the **Port Type** drop down list, select the port type that the binding relates to.
4. Enter a name for the new binding, or accept the default provided.
5. From the **Style** drop down list, select either **rpc** or **document**, to indicate whether message parts pertaining to each operation are to consist of RPC-based parameters and return values or document-based body entries by default. The value you choose is subsequently populated in the `soap:binding style` attribute in your WSDL contract.
6. From the **Use** drop down list, select either **encoded** or **literal**, to indicate whether message parts are to consist of abstract type definitions or concrete schema definitions. The value you choose is subsequently populated in the `soap:body use` attribute in your WSDL contract.
7. Click **Next** to display the Edit Binding panel, as shown in Figure 89.

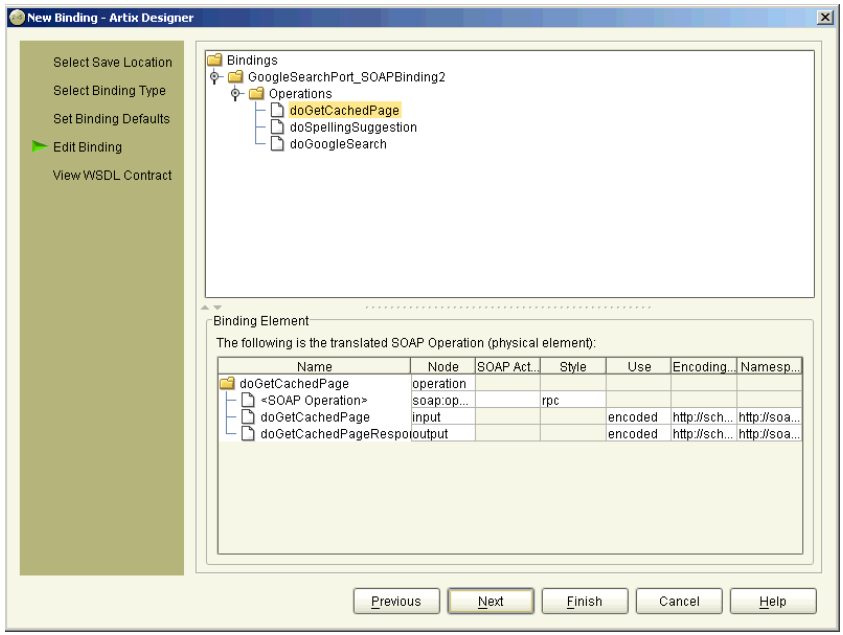

Figure 89: *New Binding wizard—Edit SOAP Binding panel*

8. Click on the name of an operation within your binding.
9. If you want to include a SOAPAction field in the HTTP header of a SOAP message, use the **SOAP Action** cell in the Binding Elements table to specify the URL that represents the resource being requested by the operation.

   Note: This step only relates to the use of SOAP over HTTP, but it is not mandatory for the purposes of Artix. It is available in case some third-party SOAP servers that do use a SOAPAction field in their HTTP headers are to be contacted.

10. If you want to override the default setting for **Style** that you set in **step 5**, click on the **Style** cell and select another value.
11. If you want to override the default setting for **Use** that you set in **step 6**, click on the **Use** cell and select another value.
12. If you want to use other customized encoding styles, add the URL(s) relating to each style to the relevant field(s) in the Encoding Style column. (**Note**: Only possible where Use=Encoded).

    Note: If you want this field to contain more than one URL, ensure that they are separated by spaces, and ordered according to the most restrictive set of rules first and least restrictive set of rules last.

13. Click **Next** to display the Binding Summary panel, as shown in Figure 90.

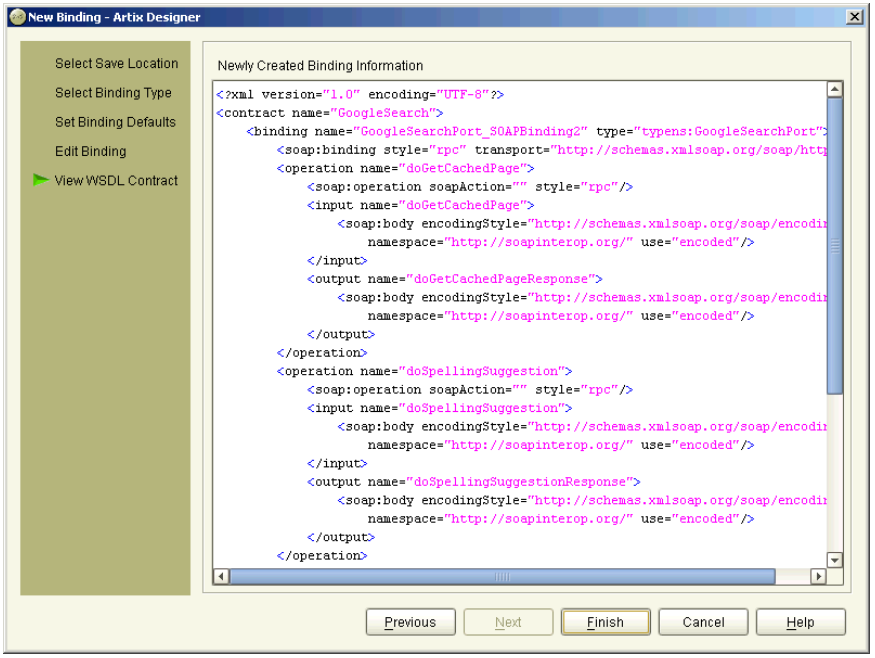

Figure 90: *New Binding wizard—SOAP Binding Summary panel*

14. Click **Finish** to close this wizard and return to the Artix Designer.

Adding a SOAP Binding, Service, and Port at the Same Time

Overview

There is a smart-menu option you can use if you would like to create a SOAP binding, service, and port for a resource. It is called SOAP enabling, and you can access it either through the Resource menu or via the contextual menu after first selecting a resource in the Designer Tree.

Procedure

To create a SOAP binding, service, and port using the smart-menu option:

1. Select a resource from the Designer Tree, and select **Resource | SOAP Enable**, to display the SOAP Enable dialog as shown in Figure 91.

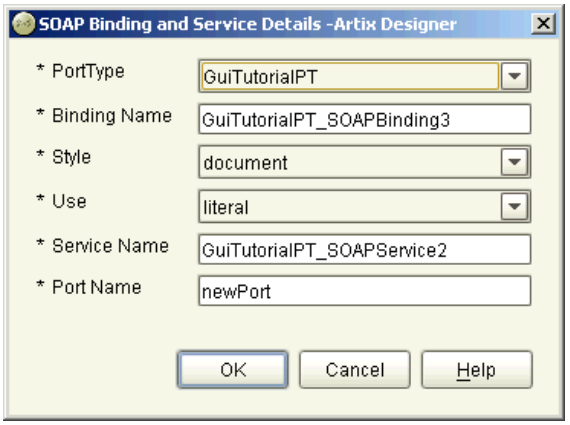

Figure 91: *SOAP Enable dialog*

2. Select the Port Type to use from the drop-down list provided. This list contains all port types that have been defined for this resource.
3. Type a name for the new binding in the Binding Name field, or accept the default provided.
4. Select the Style for this Binding from the drop-down list provided. Valid values are **rpc** or **document**.
5. Select the Use for this Binding from the drop-down list provided. Valid values are **literal** or **encoded**.

6. Type a name for the new service in the Service Name field, or accept the default provided.
7. Type a name for the new port in the Port Name field, or accept the default provided.
8. Click **OK** to close this dialog and return to the Artix Designer. The new binding, service, and port will be displayed in the relevant sections within the WSDL model diagram.

Adding an XML Binding

Overview

The pure XML payload format provides an alternative to the SOAP binding by allowing services to exchange data using straight XML documents without needing the overhead of the SOAP envelope.

Procedure

To add an XML binding to an Artix contract:

1. At the Binding Type panel, select **XML** and click **Next** to display the Binding Defaults panel, as shown in Figure 92.

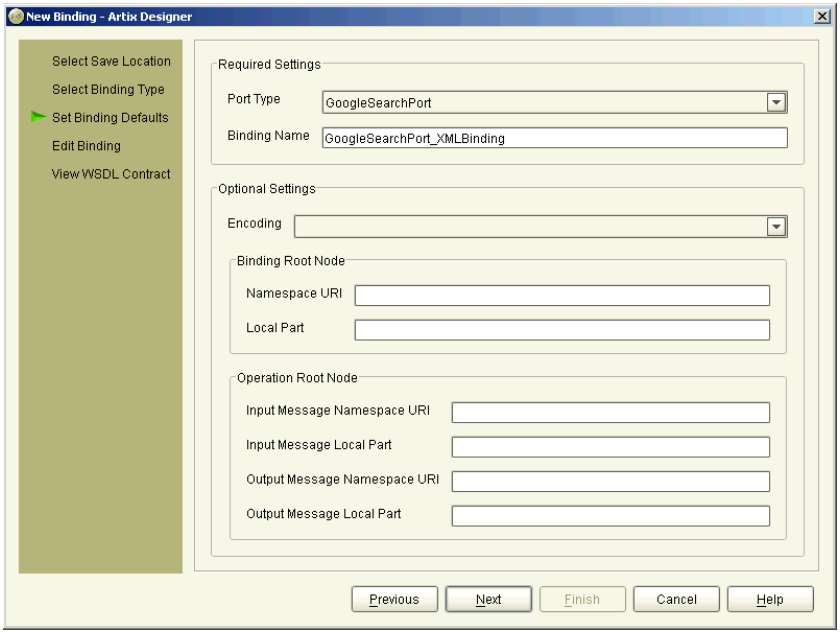

Figure 92: *New Binding wizard—XML Binding Defaults panel*

2. From the **Port Type** drop down list select the Port Type you want to map to the XML binding.
3. Enter a name for the new binding, or accept the default provided.

4. Under the **Additional Settings**, select an **Encoding** value.
5. Enter values in the Binding Route Node section. This is the *Qname* for the binding. This is a unique identifier made up of two parts:
   - Namespace URI - the location of the binding element
   - Local Part - any name you wish to append to the binding element
6. Enter values in the Operation Root Node section. This is the *Qname* at the operation level. This is a unique identifier, again made up of two parts but this time there will be two parts for each message, i.e. input and output, or just input for one-way messages:
   - Namespace URI - the location of the binding element
   - Local Part - any name you wish to append to the binding element

   If you do not specify these values at the Operation level, the Binding Route Node is used by default.
7. Click **Next** to display the Edit Binding panel, as shown in Figure 93.

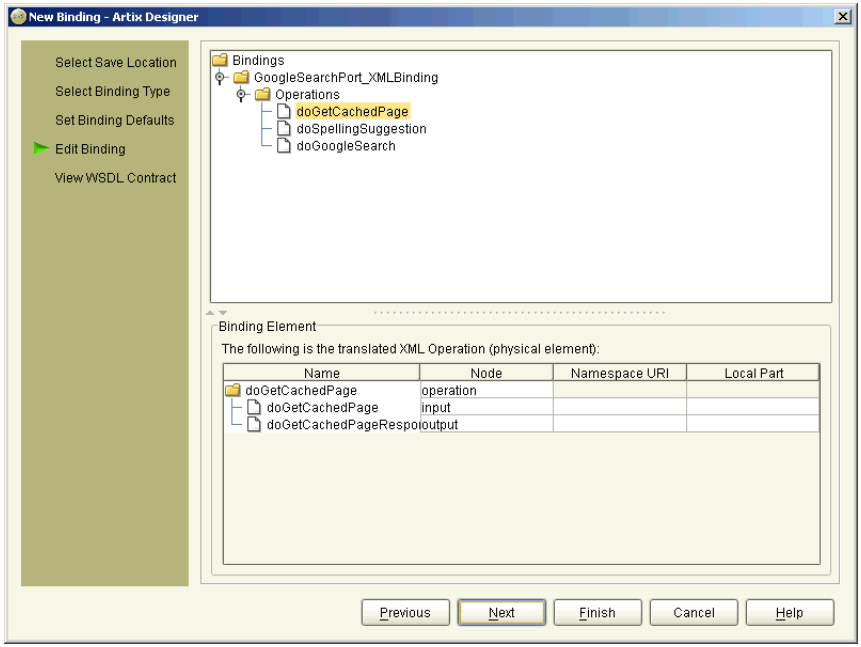

Figure 93: *New Binding wizard—Edit XML Binding panel*

8. Examine the different operations of the binding by selecting them from the tree at the top of the dialog.
9. Edit the **Namespace URI** and **Local Part** values shown in the Binding Element table, or accept the defaults provided.
10. Click **Next** to display the View Binding Summary panel, as shown in Figure 94.

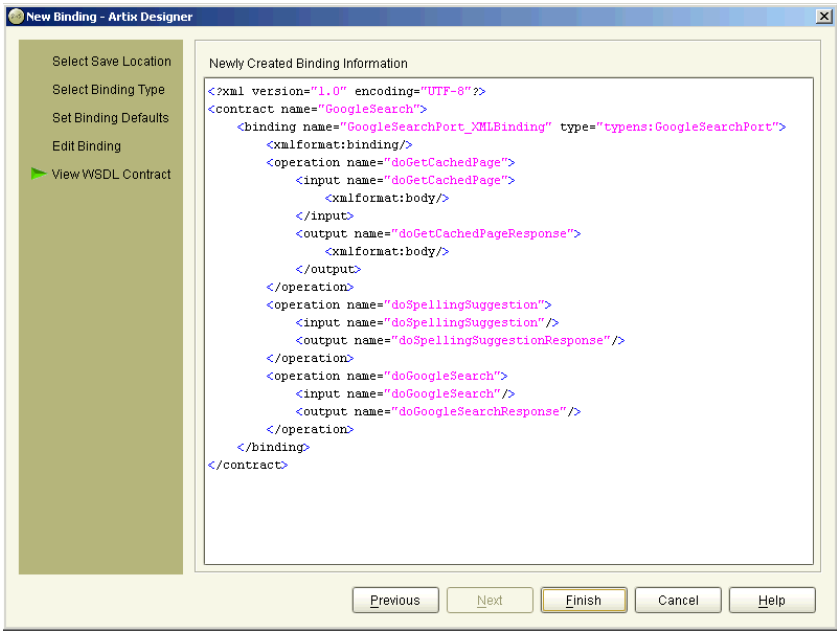

Figure 94: *Binding wizard—XML Binding Summary panel*

11. Click **Finish** to close this wizard and return to the Artix Designer.

Adding a Tagged Binding

Procedure

To add a Tagged binding to an Artix contract from the Binding Type panel:

1. Select **Tagged**, and then click **Next** to display the Binding Defaults panel, as shown in Figure 95.

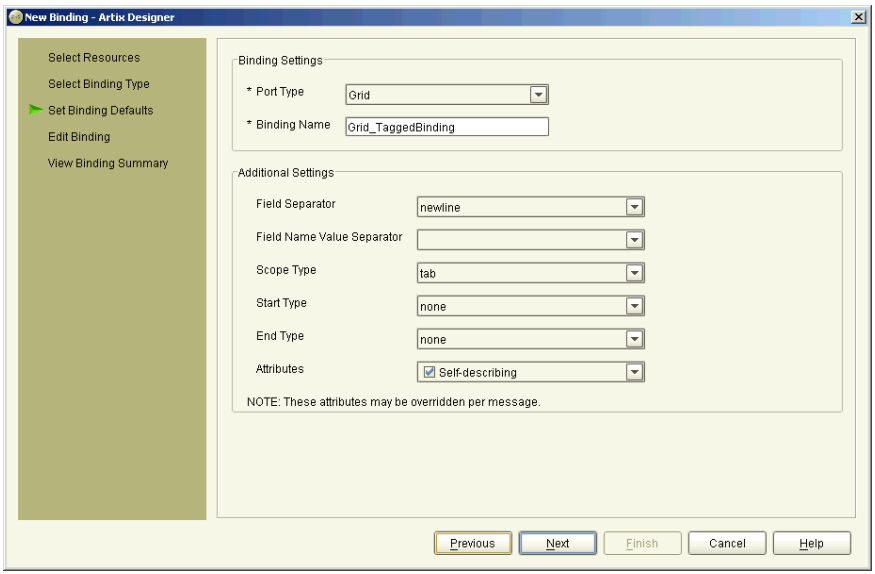

Figure 95: *Binding wizard—Tagged Binding Defaults*

2. From the **Port Type** drop down list select the Port Type you want to map to the XML binding.
3. Enter a name for the new binding, or accept the default provided.
4. Enter values for the additional (optional) settings if required. These settings can also be over-written for each message. The settings are:
   - Field separator - valid values are **newline**, **comma**, **pipe**, or **semicolon**.
   - Field name value separator - valid values are **equals**, **tab**, or **colon**.

- Scope Type - valid values are **tab**, **curlybrace**, or **none**.
- Start type - valid values are **none** or **star**.
- End type - valid values are **newline** or **percent**.
- Attributes - this field contains several settings you can enable by clicking the check box. For more information about each of the attributes, see the Artix online help.

5. Click **Next** to display the Edit Binding panel, as shown in Figure 96, which displays the generated operations and elements.

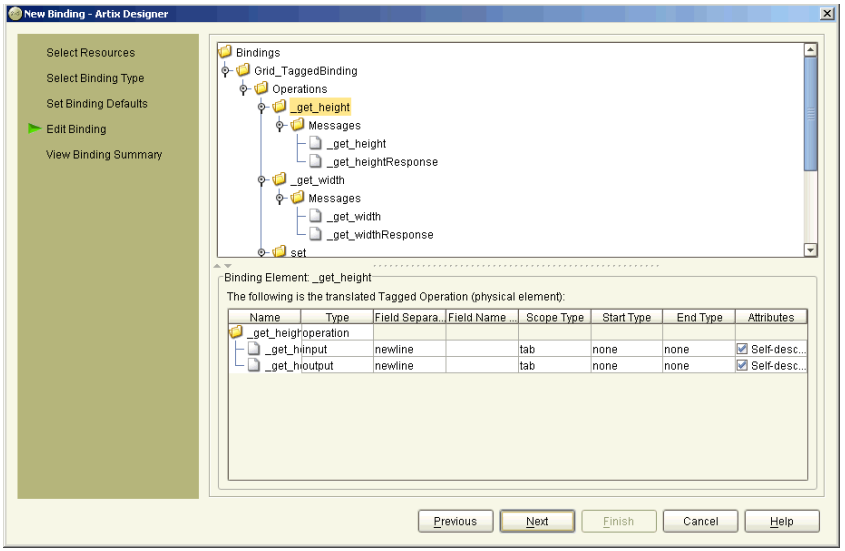

Figure 96: *Binding wizard—Edit Tagged Binding panel*

6. Examine the different operations of the binding by selecting them from the tree at the top of the dialog. You can edit some of the elements in the bottom pane by typing directly in the cells.

7. Click **Next** to display the View Binding Summary panel, as shown in Figure 97.

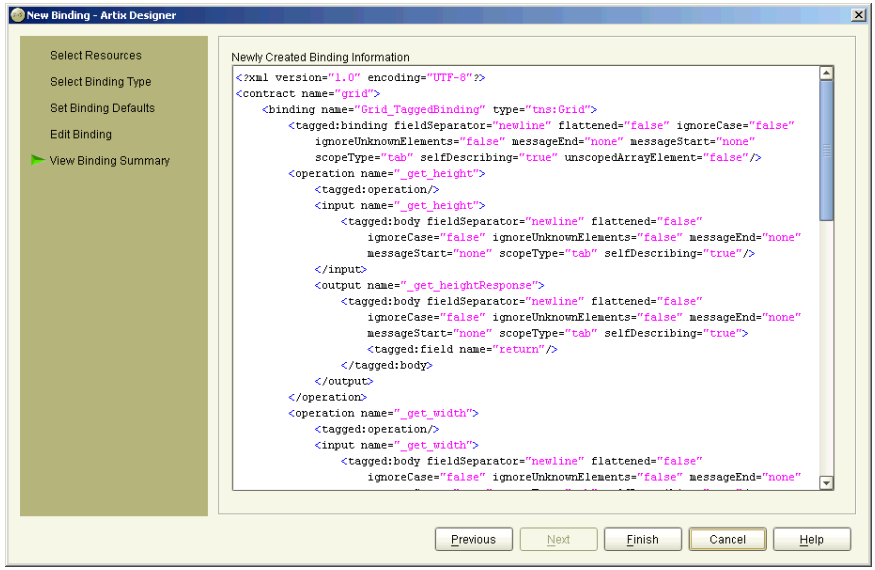

Figure 97: *Binding wizard—Tagged Binding Summary panel*

8. Click **Finish** to close this wizard and return to the Artix Designer.

Editing Bindings

You can edit a binding by selecting it in the Resource Navigator (Diagram view) and selecting Resource | Edit | Binding, to display the Edit Binding panel as shown in Figure 98.

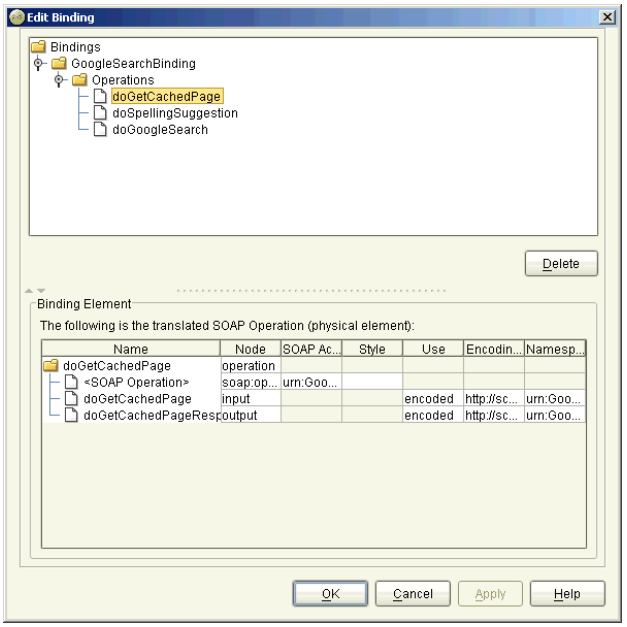

Figure 98: *Edit Binding panel*

At the Edit Binding panel, you can delete operations, by selecting them and clicking the **Delete** button.

You can also change some of the Binding Element attributes by either double-clicking the cell and typing a new value, or by clicking the cell and selecting a new value from the drop-down list provided.

When you have finished making your changes, click **Apply** to update the binding and **OK** to close the wizard and return to the Artix Designer.

CHAPTER 6

Adding Services

A service defines the ports supported by the Web Service.

In this chapter

This chapter discusses the following topics:

Introduction

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.

For each of the supported protocols, there is one `<port>` element. The `<service>` element is a collection of these ports. A service can contain one or many ports.

Typically, 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.

Procedure

To add a Service to your Artix contract:

1. Select **Resource | New | Service** from the menu bar to display the New Service wizard, as shown in Figure 99.

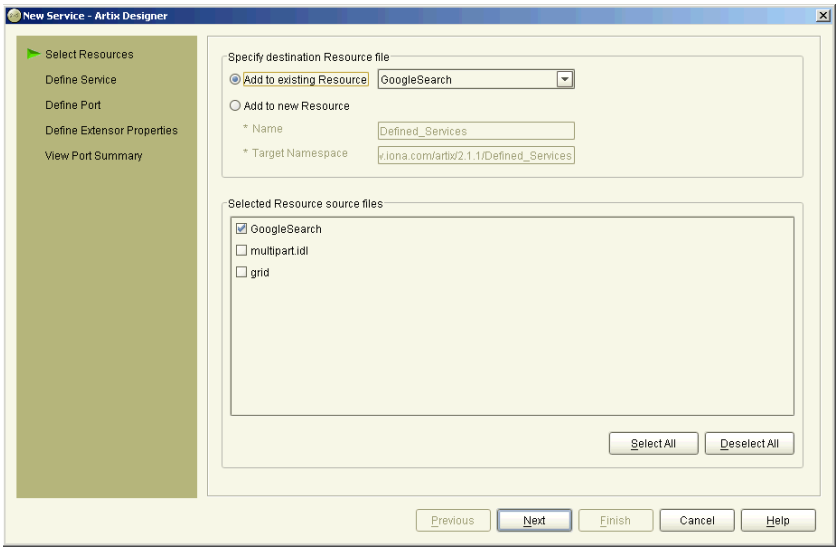

Figure 99: *New Service wizard*

2. Select where to create the WSDL entry for the new service.
   - **Add to existing WSDL** adds the service information to the existing contract.
   - **Add to new WSDL** creates a new WSDL document that contains the service information.
3. Select the resources from this collection that you want to use as the source for this new service. If you selected a resource before invoking the New Service wizard, that resource is selected by default. You can also select other resources to use as sources for this service - this will give you more bindings to choose from when defining the ports later in this wizard.
4. Click **Next** to display the Service Definition panel, as shown in Figure 100.

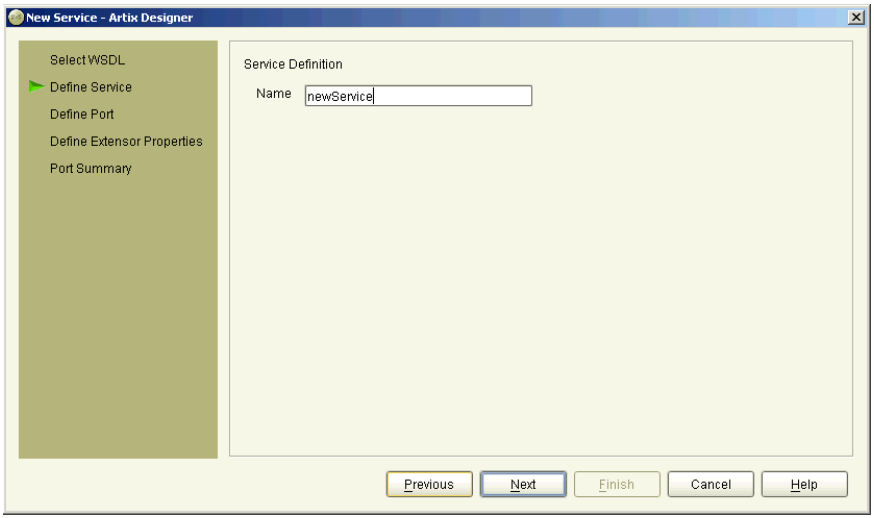

Figure 100:*New Service wizard—Service Definition panel*

5. Enter a name for the new service, or accept the default provided.

6. Click **Next** to display the Port Definition panel, as shown in Figure 101.

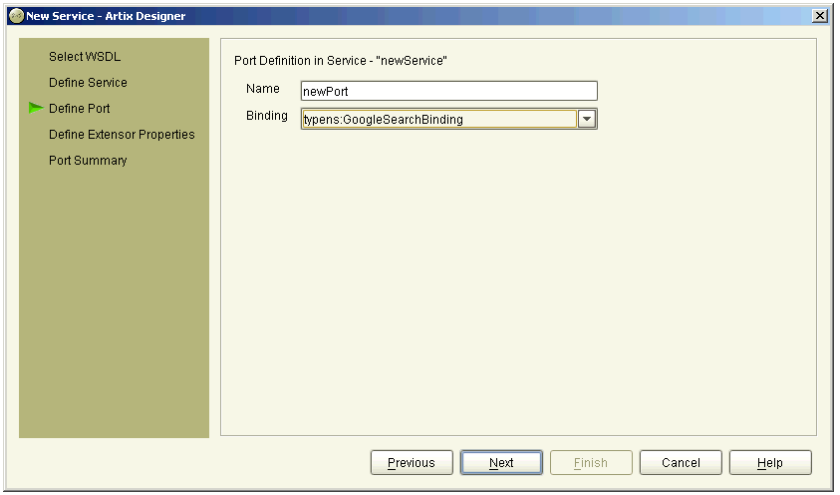

Figure 101:*New Service wizard—Port Definition panel*

7. Enter a name for the new port that is being created as part of this service, or accept the default provided.
8. From the **Binding** drop down list, select the binding that the port is going to expose.
9. Click **Next** to display the Extensor Properties panel.

Turn to the page that is relevant for the type of service you are creating:

- For a CORBA service, see page 143
- For a non-secure HTTP service, see page 146
- For a secure HTTP service, see page 147
- For a WebSphere MQ service, see page 149
- For a Tuxedo service, see page 151
- For a Java Message Service (JMS), see page 154
- For an IIOP Tunnel service, see page 156
- For a non-secure SOAP over HTTP service, see page 159
- For a secure SOAP over HTTP service, see page 162

Adding a CORBA Port

CORBA ports are described using the IONA-specific WSDL elements `<corba:address>` and `<corba:policy>` within the WSDL `<port>` element, to specify how a CORBA object is exposed.

Procedure

1. At the Extensor Properties panel, as shown in Figure 102, select **CORBA** from the **Transport Type** drop down list.

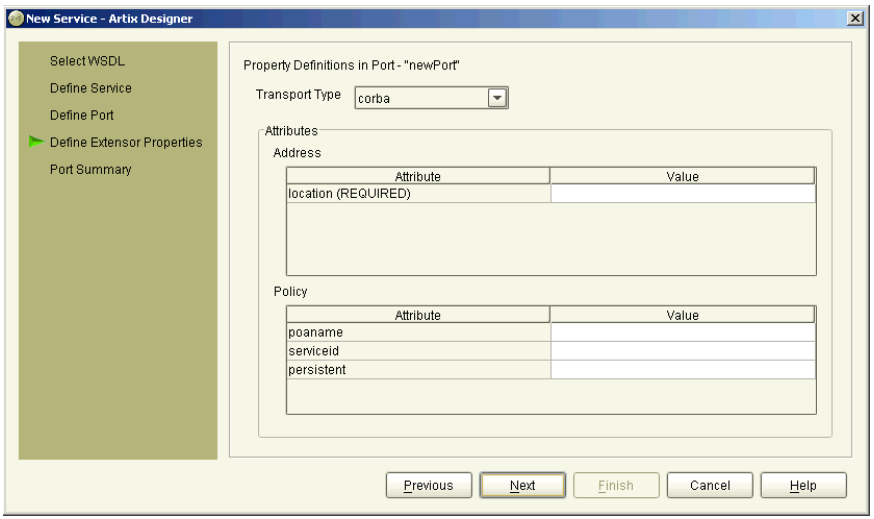

Figure 102: *New Service wizard—Define CORBA Extensor Properties*

2. In the **Address** table, enter the CORBA address in the **Location** field.
3. If you want to set any of the supported Policy Attributes, enter a valid value in the **Policy** table for any or all of the attributes listed.

4. Click **Next** to display the Summary panel, as shown in Figure 103.

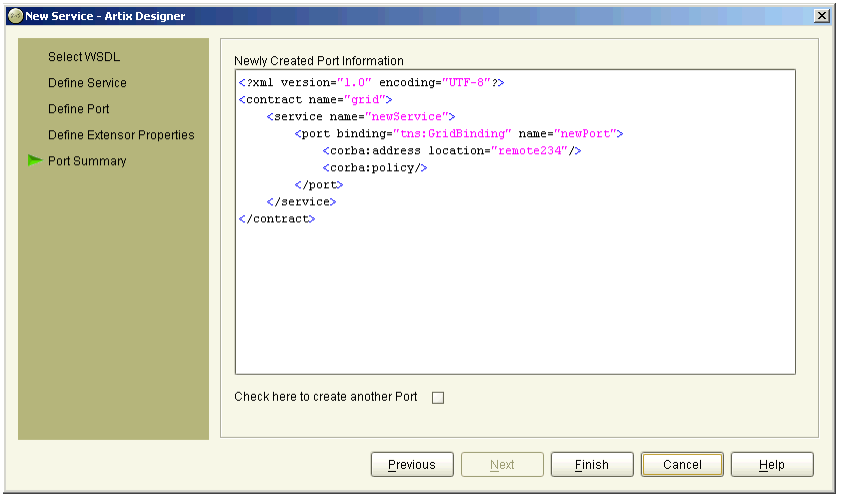

Figure 103:*New Service wizard—Summary panel (CORBA)*

5. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
6. Click **Finish** to close this wizard and return to the Designer.

Artix expects the IOR for the CORBA object to be located in a file called `objref.ior`, and creates a persistent POA with an object id of `personalInfo` to connect the CORBA application.

Adding a CORBA Binding, Service, and Port at the Same Time

Overview

There is a smart-menu option you can use if you would like to create a CORBA binding, service, and port for a resource. It is called CORBA enabling, and you can access it either through the Resource menu or via the contextual menu after first selecting a resource in the Designer Tree.

Procedure

To create a CORBA binding, service, and port using the smart-menu option:

1. Select a resource from the Designer Tree, and select **Resource | CORBA Enable**, to display the CORBA Enable dialog as shown in Figure 104.

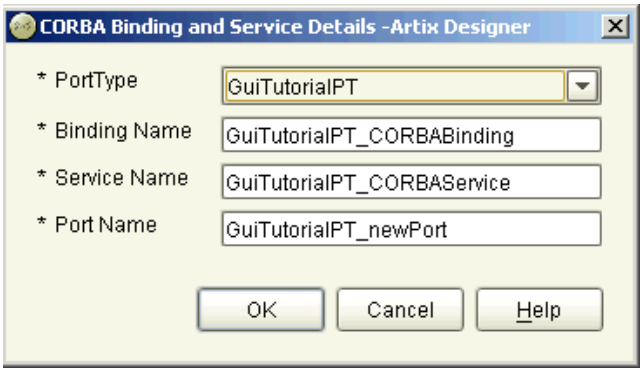

Figure 104:*CORBA Enable dialog*

2. Select the Port Type to use from the drop-down list provided. This list contains all port types that have been defined for this resource.
3. Type a name for the binding in the Name field, or accept the default.
4. Type a name for the service in the Name field, or accept the default.
5. Type a name for the port in the Port Name field, or accept the default.
6. Click **OK** to close this dialog and return to the Artix Designer. The new binding, service, and port will be displayed in the relevant sections within the WSDL model diagram.

Adding an HTTP Port

When adding an HTTP port, you have the option of making it either secure or non-secure. A secure port means that the connections with that port, and information moving in and out of it, are secure.

Non-Secure Connections

This section describes how to add an HTTP port that does not enable secure connections.

Before you begin

To add a port, you must have already created a binding within the `<binding>` component of the contract. See "Adding Bindings" on page 115 for more information.

Procedure

To add an HTTP port to your service contract:

1. At the Extensor Properties panel, as shown in Figure 105, select **http** from the **Transport Type** drop-down list.

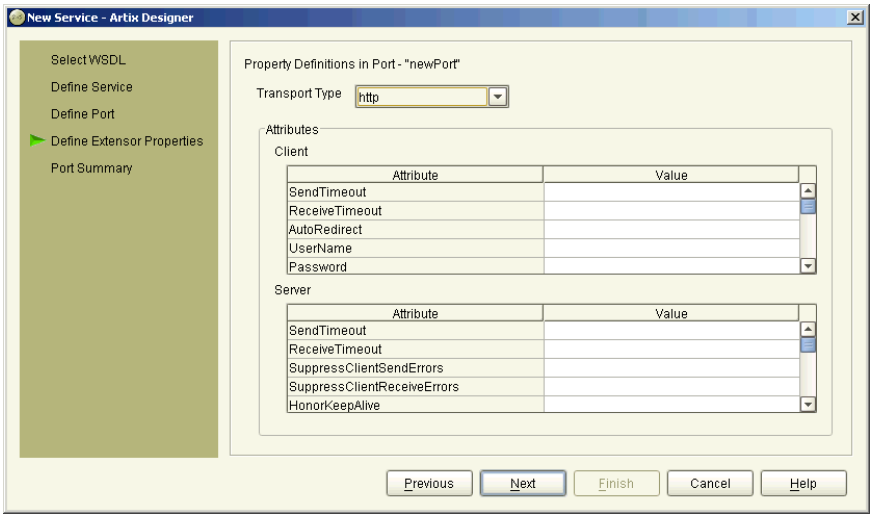

Figure 105: *New Service wizard—Define HTTP Extensor Properties*

2. To specify a value for a one of the client or server attribute, type (or in the case of certain true or false attributes select) the value you want.
3. Click **Next** to display the Summary panel, as shown in Figure 106.

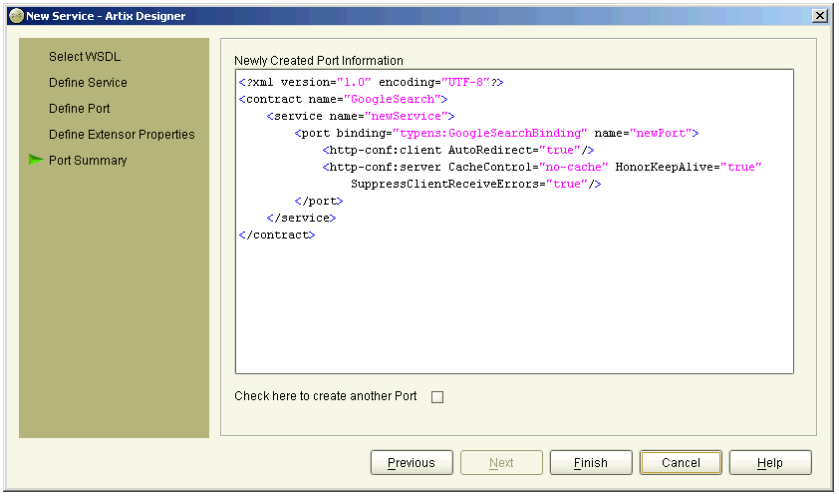

Figure 106: *New Service wizard—Summary panel (HTTP)*

4. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
5. Click **Finish** to close this wizard and return to the Artix Designer.

Secure Connections

This section describes how to add an HTTP port that enables secure connections.

Before you begin

To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See "Adding Bindings" on page 115 for more information.

SSL-related attributes

The SSL-related attributes that can be configured to be included in the <http-conf:client> and <http-conf:server> elements of an HTTP port binding are as follows:

Client SSL Attributes	Server SSL Attributes
UseSecureSockets	UseSecureSockets
ClientCertificate	ServerCertificate
ClientCertificateChain	ServerCertificateChain
ClientPrivateKey	ServerPrivateKey
ClientPrivateKeyPassword	ServerPrivateKeyPassword
TrustedRootCertificate	TrustedRootCertificate

Procedure

Follow the steps described in "Procedure" on page 146, with the following minor changes:

- Specify https:// rather than http:// as the prefix for the value of the **URL** attribute in the **Client** configuration table.
- Enter values for the various SSL-related attributes in the **Client** and **Server** configuration tables. See "SSL-related attributes" above for a listing of these attributes.

Note: When you specify https:// as the prefix for the value of the **URL** attribute in the **Client** configuration table, a secure HTTP connection is automatically enabled, even if **UseSecureSockets** is not set to true.

Adding a WebSphere MQ Port

The description for an Artix WebSphere MQ port is entered in a `<port>` element of the Artix contract containing the interface to be exposed over WebSphere MQ. Artix defines two elements to describe WebSphere MQ ports and their attributes:

- <mq:client> describes the port Artix client applications use to connect to an WebSphere MQ server application.
- <mq:server> describes the port WebSphere MQ client applications use to connect to Artix.

You can use one or both of the WebSphere MQ elements to describe the Artix WebSphere MQ port. Each can have different configurations depending on the attributes you choose to set.

Procedure

To add a WebSphere MQ port to an Artix contract:

1. At the Extensor Properties panel, as shown in Figure 107, select **mq** from the **Transport Type** drop-down list.

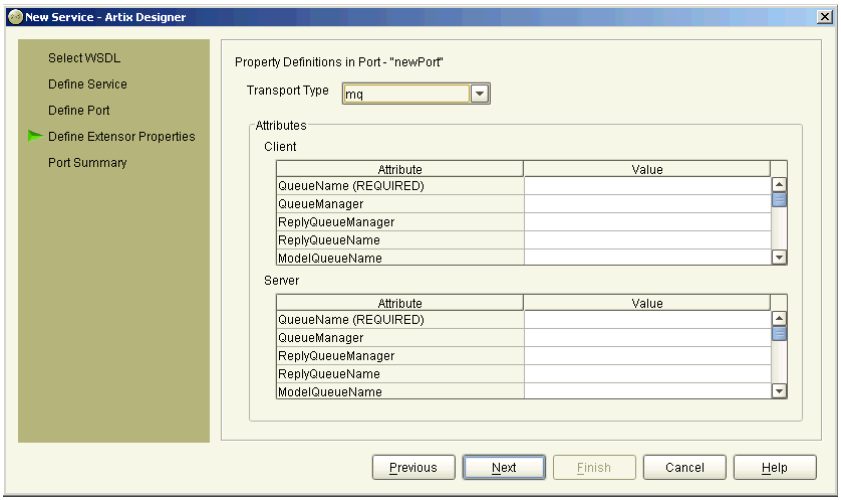

Figure 107: *New Service Wizard—Define WebSphere MQ Port Properties*

2. Enter values for the desired attributes. You must supply `QueueName` values at a minimum.
3. Click **Next** to display the Port Summary panel as shown in Figure 108.

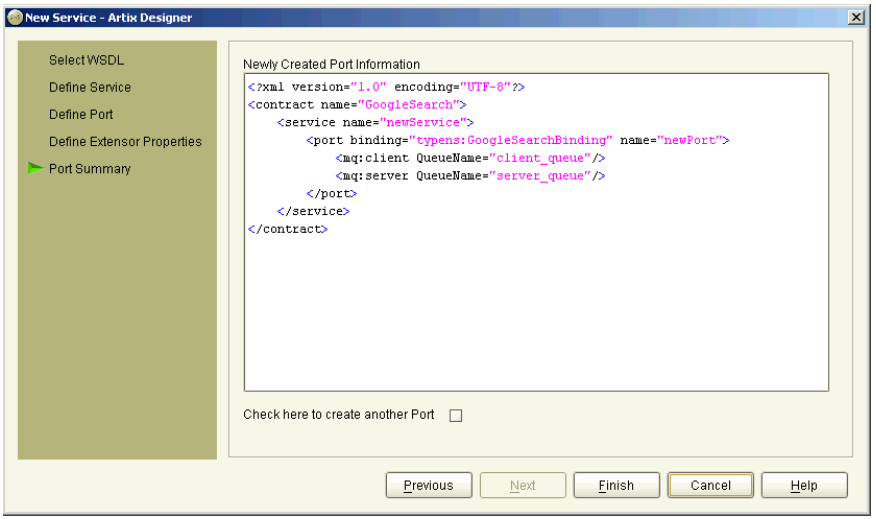

Figure 108:*New Service wizard—Summary panel (MQ)*

4. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
5. Click **Finish** to close the wizard and return to the Artix Designer.

Adding a Tuxedo Port

Artix allows services to connect using Tuxedo's transport mechanism. This provides them with all of the qualities of service associated with Tuxedo.

To use the Tuxedo transport, you need to describe the port using Tuxedo in the physical part of an Artix contract. The extensions used to describe a Tuxedo port are defined in the namespace:

```
xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
```

This namespace will need to be included in your Artix contract's `<definition>` element.

As with other transports, the Tuxedo transport description is contained within a `<port>` element. Artix uses `<tuxedo:server>` to describe the attributes of a Tuxedo port. `<tuxedo:server>` takes a single mandatory attribute, `serviceName`, which specifies the bulletin board name of the Tuxedo port being exposed.

Before you begin

Note that your Artix contract must have an existing SOAP binding before you can add a Tuxedo port. For more information, see "Adding a Fixed Binding" on page 123.

Procedure

To add a Tuxedo port to an Artix contract:

1. At the **Define Port** panel (as shown in Figure 101 on page 142), select the SOAP binding which this port will expose to the network from the Binding drop-down list.
2. Click **Next** to display the Extensor Properties panel.

3. Select **Tuxedo** from the **Transport Type** drop-down list to display the Tuxedo attributes as shown in Figure 109.

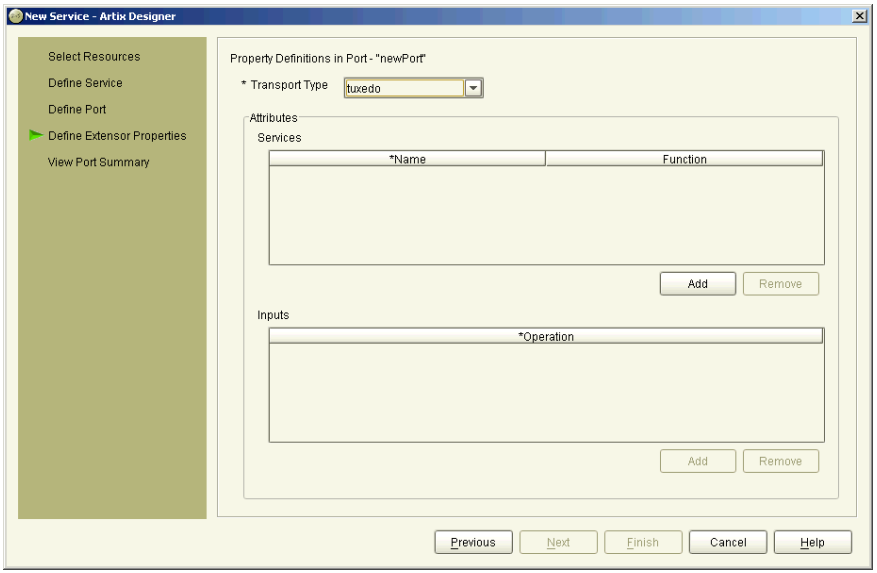

Figure 109:*New Service wizard—Define Tuxedo Port Properties panel*

4. To add a Service, click the **Add** button. You can change the name of the service or accept the one provided. You can also provide some information about the function of this service in the field provided if you like.
5. If you do add a service, you can also add an Input Operation for that service. To do this, select the Service and then click the **Add** button under the Input table.

   You can add multiple operations for each service, and you can change the operation name by selecting other available ones from the drop-down provided. This list of available operations is populated by your WSDL file.

6. Click **Next** to display the Summary panel, as shown in Figure 110.

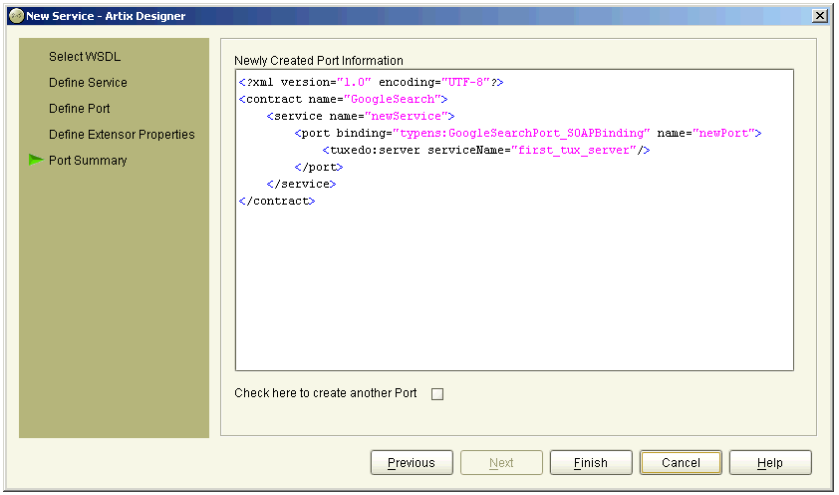

Figure 110:*New Service wizard—Summary panel (Tuxedo)*

7. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
8. Click **Finish** to close this wizard and return to the Artix Designer.

Adding a Java Message Service Port

The Java Messaging System (JMS) provides a standardized means for Java applications to send messages. Artix provides a transport plug-in that enables systems to place and receive messages from JMS implementations. One advantage of this is that Artix allows C++ applications to interact directly with Java applications over JMS.

Procedure

To add a Java Message Service (JMS) port to an Artix contract:

1. At the Extensor Properties panel, as shown in Figure 111, select **jms** from the **Transport Type** drop-down list.

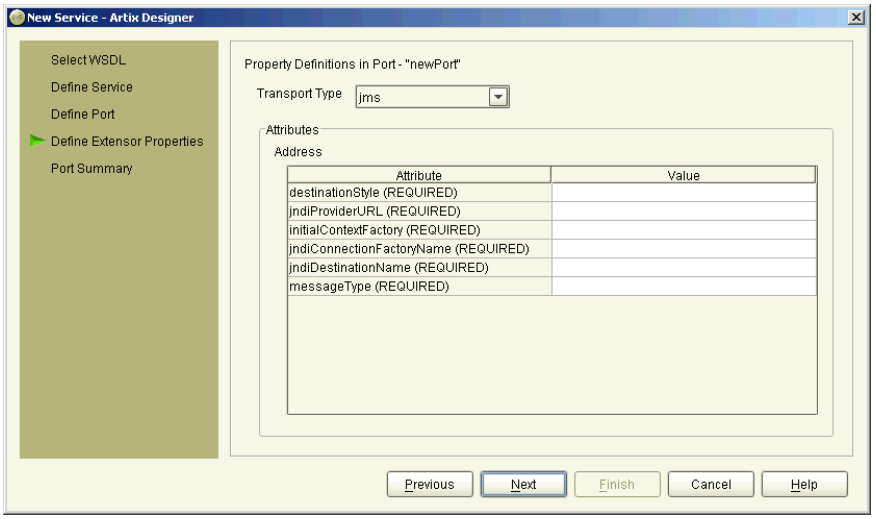

Figure 111:*New Service Wizard—Define WebSphere MQ Port Properties*

2. Enter values for the desired attributes. All attributes are required.
   - **destinationStyle** - Specifies the type of jms messaging object you're connecting to; options are topic (one-way only) or queue.
   - **jndiProviderURL** - Specifies the URL of the JNDI service where the connection information for the JMS destination is stored.

- **initialContextFactory** - Specifies the name of the `InitialContextFactory` class or a list of package prefixes used to construct URL context factory classnames.
- **jndiConnectionFactoryName** - Specifies the JNDI name bound to the JMS connection factory to use to connect to the JMS destination.
- **jndiDestinationName** - Specifies the JNDI name bound to the JMS destination to which Artix connects.
- **messageType** - Specifies how the message data will be packaged as a JMS message. `text` specifies that the data will be packaged as a `TextMessage`. `binary` specifies that the data will be packaged as an `ObjectMessage`.

3. Click **Next** to display the Summary panel as shown in Figure 108.

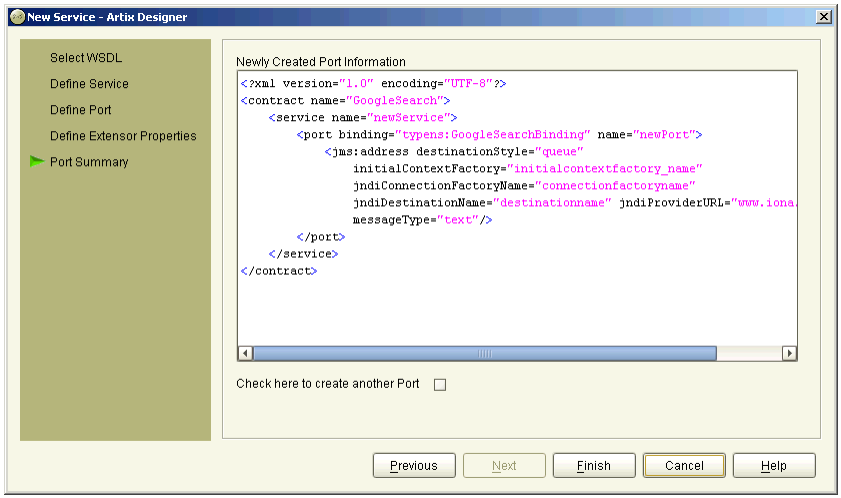

Figure 112:*New Service wizard—Summary panel (JMS)*

4. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
5. Click **Finish** to close the wizard and return to the Artix Designer.

Adding an IIOP Tunnel Port

An IIOP tunnel provides a means for taking advantage of existing CORBA services while transmitting messages using a payload format other than CORBA. For example, you could use an IIOP tunnel to send fixed format messages to an endpoint whose address is published in a CORBA naming service.

Supported payload formats

IIOP tunnels can transport messages using the following payload formats:

- SOAP
- Fixed format
- Fixed record length
- G2++
- Octet streams

Procedure

To add an IIOP tunnel port to your service contract:

1. At Extensor Properties panel, as shown in Figure 113, select **tunnel** from the **Transport Type** drop-down list.

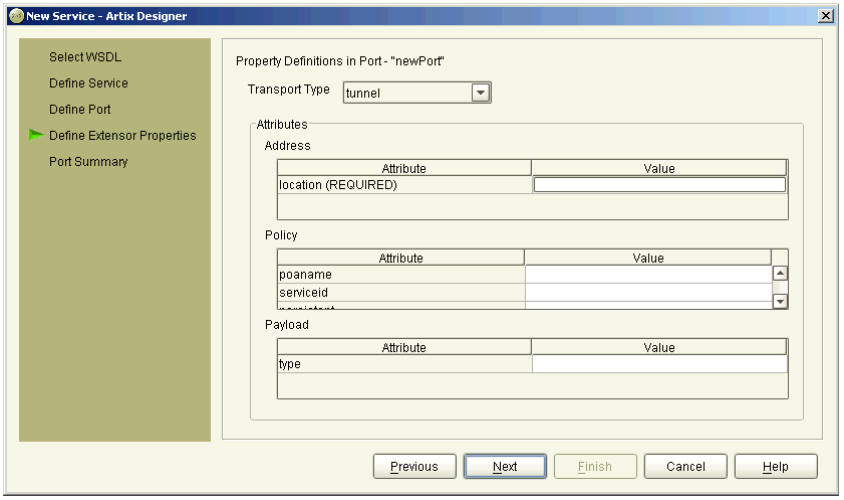

Figure 113:*New Service wizard—Define IIOP Port Properties panel*

2. From the drop down list in the **Transport** box, select **tunnel**.
3. In the **Address** table, enter the address in the line for **Location**.
4. If you want to set any of the supported POA policies, place a check in the **Specified** box on the appropriate line in the **Policy** table and enter a valid value.

5. Click **Next** to display the Port Summary panel, as shown in Figure 114.

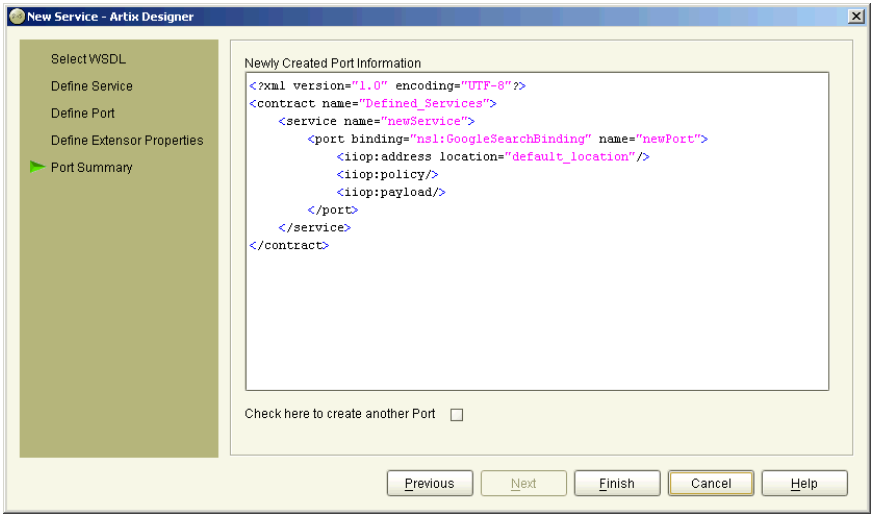

Figure 114:*New Service wizard—Summary panel (IIOP)*

6. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
7. Click **Finish** to close this wizard and return to the Artix Designer.

Artix expects the IOR for the IIOP tunnel to be located in a file called `objref.ior`, and creates a persistent POA with an object id of `personalInfo` to configure the IIOP tunnel.

Adding a SOAP Port

Non-Secure Connections

This section describes how to add a port for SOAP over HTTP that does not enable secure connections.

Before you begin

To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See "Adding Bindings" on page 115 for more information.

Procedure

To enable the use of SOAP over HTTP:

1. At the Extensor Properties panel, as shown in Figure 115, select **SOAP** from the **Transport Type** drop-down list.

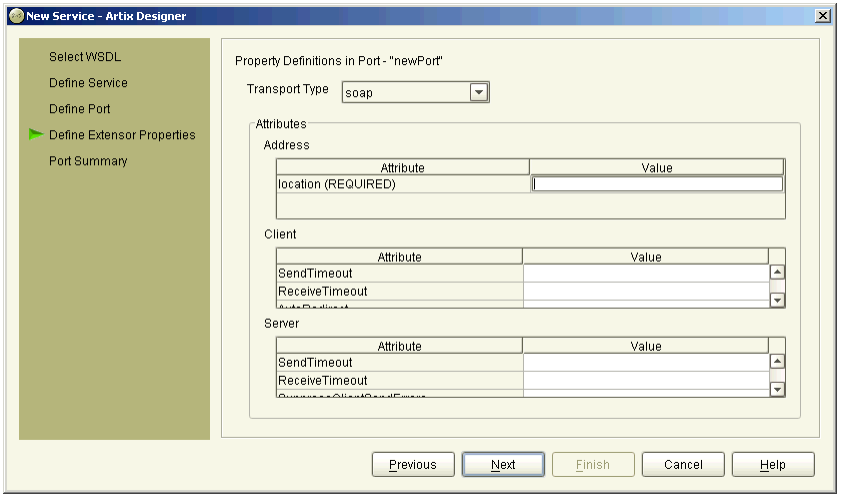

Figure 115: *New Service wizard—Define SOAP Properties panel*

2. In the **Value** field corresponding to the **location** line of the **Address** configuration table, type the URL that represents the resource being requested.

   Note: The **Address** configuration table relates to the `soap:address` element within the port component of the WSDL contract. You must specify a value for the **location** attribute.

3. To specify a value for another attribute, place a check in the **Specified** box on the appropriate line in the appropriate configuration table, and type or (in the case of certain true or false attributes) select the value you want.

   Note: All attributes are optional in the **Client** and **Server** configuration tables. These relate to the `http-conf:client` and `http-conf:server` elements that can be specified as peers of the `soap:address` element under the same port binding. See "SSL-related attributes" below for details of each attribute relating to `http-conf:client` and `http-conf:server`.

4. Click **Next** to display the Summary panel, as shown in Figure 116.

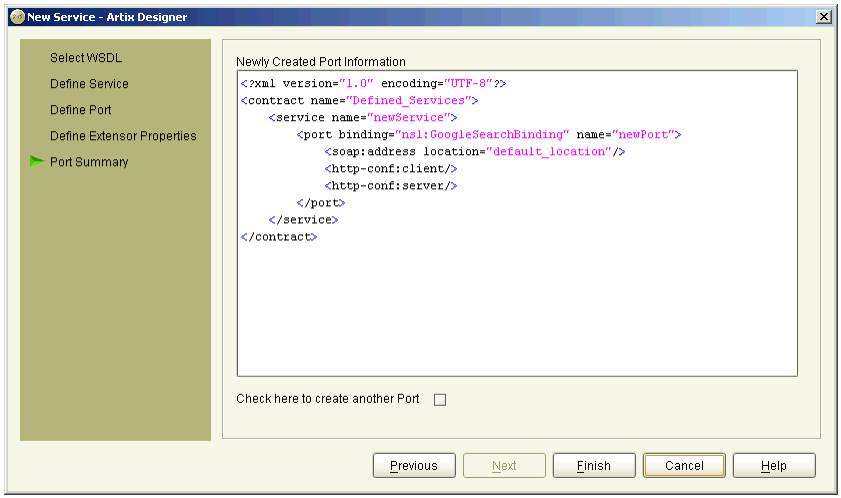

Figure 116:*New Service wizard—Summary panel (SOAP)*

5. To add another port to this service, check the box provided under the summary panel and click **Next**. This will take you back to the Define Port panel (as shown in Figure 101 on page 142), where you can enter details for the new port.
6. Click **Finish** to close this wizard and return to the Artix Designer.

Secure Connections

This section describes how to add a port for SOAP over HTTP that enables secure connections.

Before you begin

To add a port, you must have already created a payload format binding within the `<binding>` component of the contract. See "Adding Bindings" on page 115 for more information.

SSL-related attributes

The SSL-related attributes that can be configured to be included in the `<http-conf:client>` and `<http-conf:server>` elements of an HTTP port binding are as follows:

Client SSL Attributes	Server SSL Attributes
`UseSecureSockets`	`UseSecureSockets`
`ClientCertificate`	`ServerCertificate`
`ClientCertificateChain`	`ServerCertificateChain`
`ClientPrivateKey`	`ServerPrivateKey`
`ClientPrivateKeyPassword`	`ServerPrivateKeyPassword`
`TrustedRootCertificate`	`TrustedRootCertificate`

Procedure

Follow the steps in "Procedure" on page 159, with the following minor changes:

- Specify `https://` rather than `http://` as the prefix for the value of the **location** attribute in the **Address** configuration table.
- Enter values for the various SSL-related attributes in the **Client** and **Server** configuration tables. See "SSL-related attributes" above for a listing of these attributes.

Note: When you specify `https://` as the prefix for the value of the **location** attribute in the **Address** configuration table, a secure HTTP connection is automatically enabled, even if **UseSecureSockets** is not set to `true`.

Adding a SOAP Binding, Service, and Port at the Same Time

Overview

There is a smart-menu option you can use if you would like to create a SOAP binding, service, and port for a resource. It is called SOAP enabling, and you can access it either through the Resource menu or via the contextual menu after first selecting a resource in the Designer Tree.

Procedure

To create a SOAP binding, service, and port using the smart-menu option:

1. Select a resource from the Designer Tree, and select **Resource | SOAP Enable**, to display the SOAP Enable dialog as shown in Figure 117.

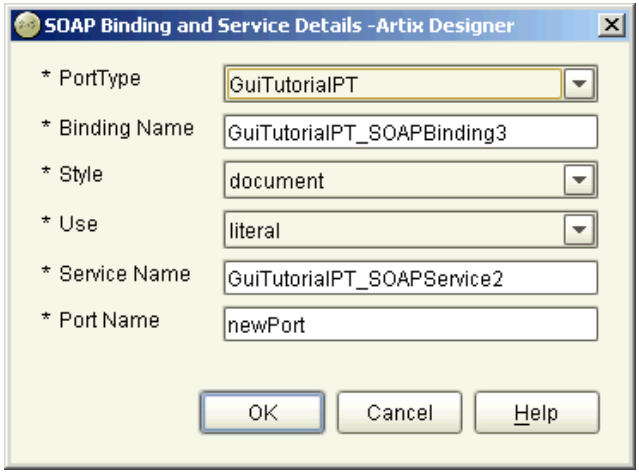

Figure 117: *SOAP Enable dialog*

2. Select the Port Type to use from the drop-down list provided. This list contains all port types that have been defined for this resource.
3. Type a name for the new binding in the Binding Name field, or accept the default provided.
4. Select the Style for this Binding from the drop-down list provided. Valid values are **rpc** or **document**.

5. Select the Use for this Binding from the drop-down list provided. Valid values are **literal** or **encoded**.
6. Type a name for the new service in the Service Name field, or accept the default provided.
7. Type a name for the new port in the Port Name field, or accept the default provided.
8. Click **OK** to close this dialog and return to the Artix Designer. The new binding, service, and port will be displayed in the relevant sections within the WSDL model diagram.

Editing Services

You can edit a service by selecting **Resource | Edit | Services**, to display the Edit Services panel as shown in Figure 118.

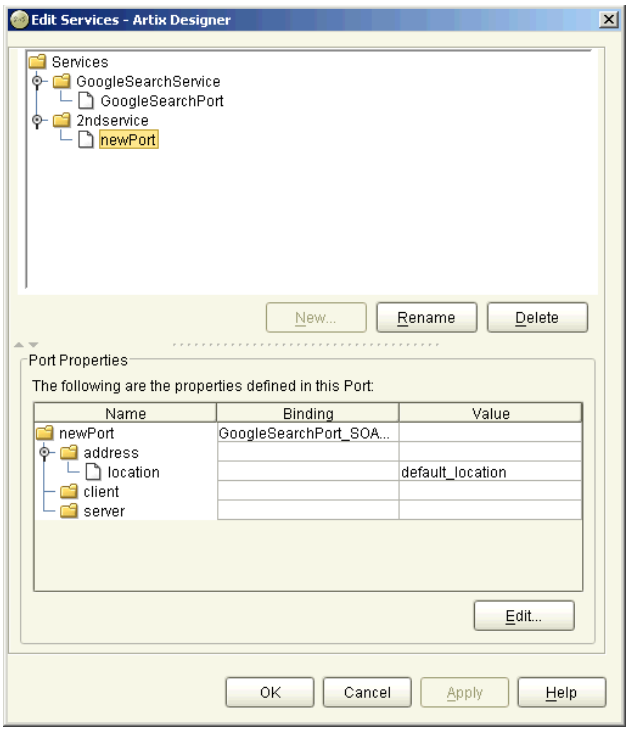

Figure 118:*Edit Services panel*

At the Edit Services panel, all of your services and their associated ports are listed in the top half of the dialog. From here you can:

- Rename a service or a port by selecting it and clicking **Rename**
- Delete a service or a port by selecting it and clicking **Delete**.
- Add a new service by clicking **Add** to display the New Service wizard.

Editing port properties

When you select a port in the top of this dialog, the port properties are displayed in the Port Properties panel at the bottom of the dialog.

To change any of the Port Properties, click the **Edit** button to display the Edit Port Properties dialog, as shown in Figure 119.

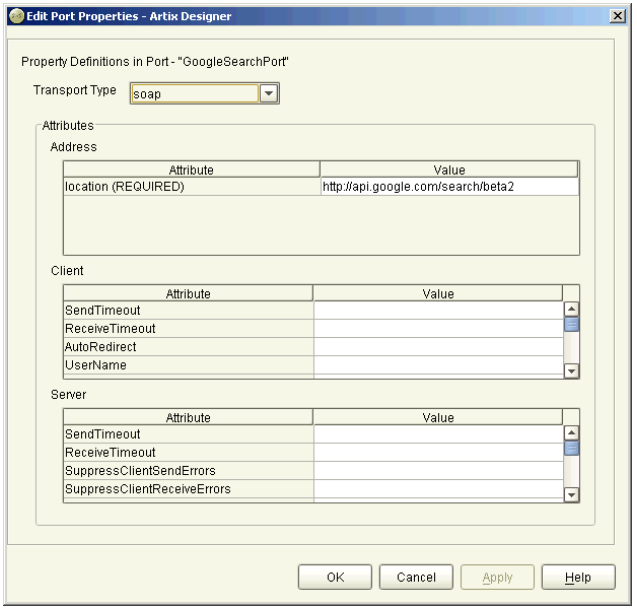

Figure 119:*Edit Port Properties dialog*

To change values of attributes in this dialog, click on the value field to either select or type the new value.

When you have finished making your changes, click **Apply** to update the port, and **OK** to close the wizard and return to the Edit Services dialog, where your changes are displayed in the Port Properties panel.

Click **OK** to close this dialog and return to the Artix Designer.

CHAPTER 7

Routing Messages

Artix provides messages routing based on operations, ports, or message attributes.

In this chapter

This chapter discusses the following topics:

What is a Route?

Overview

Artix routing is implemented within Artix collections and is controlled by rules specified in the collection's contract. Artix collections that include routing rules can be deployed into an Artix service.

Artix supports the following types of routing:

- "Port-based"
- "Operation-based"

A router's contract must include definitions for the source services and destination services. The contract also defines the routes that connect source and destination ports, according to some specified criteria. This routing information is all that is required to implement port-based or operation-based routing. Content-based routing requires that application code be written to implement the routing logic.

Port-based

Port-based routing acts on the port or transport-level identifier, specified by a `<port>` element in an Artix contract. This is the most efficient form of routing. Port-based routing can also make a routing decision based on port properties, such as the message header or message identifier. Thus Artix can route messages based on the origin of a message or service request, or based on the message header or identifier.

Operation-based

Operation-based routing lets you route messages based on the logical operations described in an Artix contract. Messages can be routed between operations whose arguments are equivalent. Operation-based routing can be specified on the interface, `<portType>`, level or the finer grained operation level.

Creating a Route

Overview

The Artix Designer includes a routing wizard that assists you in creating routes from the services available in your contract. It walks you through the steps of creating a route and provides you with the valid options for the services available. It performs all of the compatibility testing for you and will never allow you to create an invalid route.

Procedure

To create a route:

1. From the Designer Tree, select a contract with multiple service definitions that have operations that can be routed.
2. Select **Contracts | New | Route** from the menu bar to display the New Route wizard, as shown in Figure 120.

> **Note:** If the Route option is not available, your contract does not have any compatible operations for routing. For a contract to be able to be routed, it needs to contain two or more services with compatible port types. See "Adding Services" on page 139 for more information.

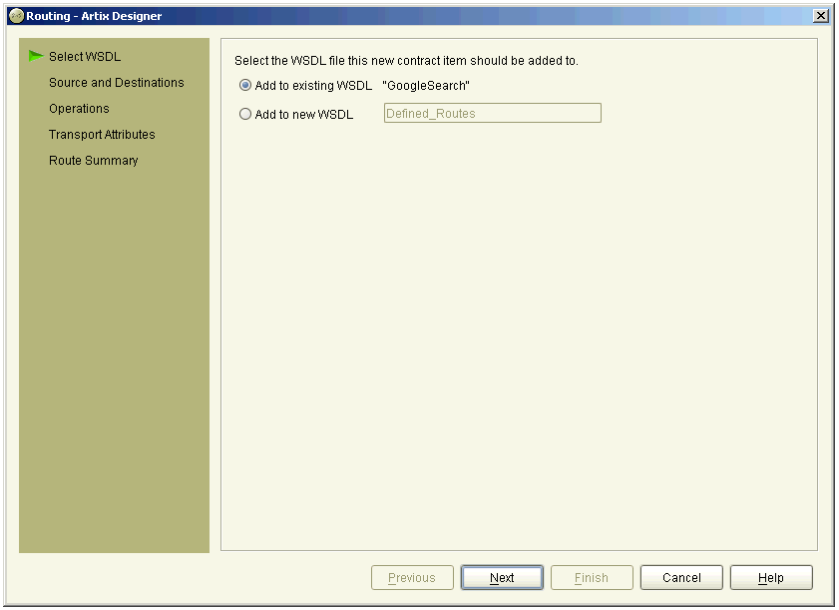

Figure 120:*New Route wizard*

3. Select where you want to add the routing information.
   - **Add to existing WSDL** adds the route information to the existing contract.
   - **Add to new WSDL** creates a new WSDL document that contains the route information.

4. Click **Next** to display the Source and Destination panel, as shown in Figure 121.

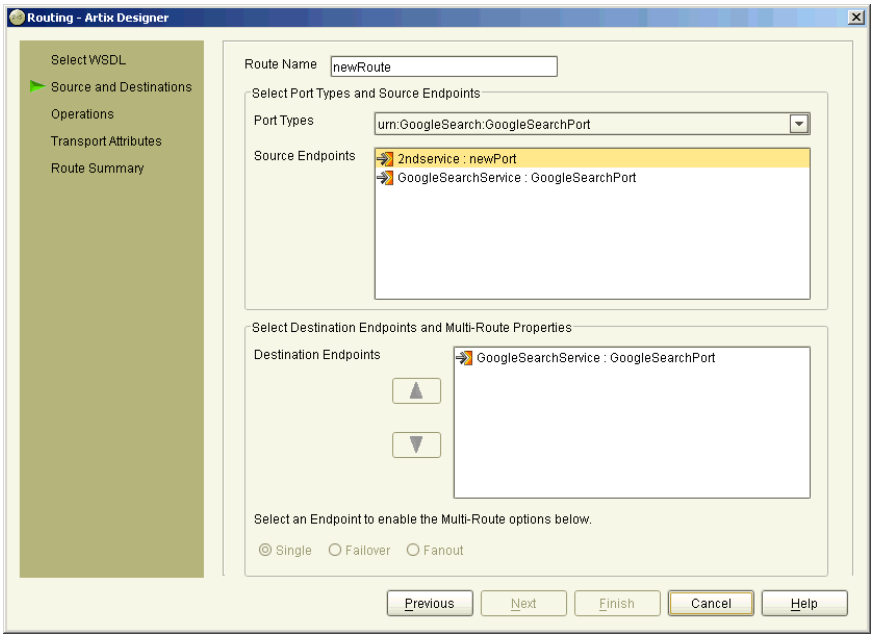

Figure 121:*New Route wizard—Source and Destination panel*

5. Enter a name for the route, or accept the default provided
6. Select the source `portType` for the route from the **PortType** pull-down list.
7. Select the source endpoint from the available options in the **Source Endpoints** list.
8. Select the destination endpoint from the available options in the **Destination Endpoints** list.
9. If you selected multiple destination endpoints on the previous screen, select either **Failover** or **Fanout** under **Multiple Route Destination Preference**.

10. Click **Next** to display the Operation Routing panel, as shown in Figure 122.

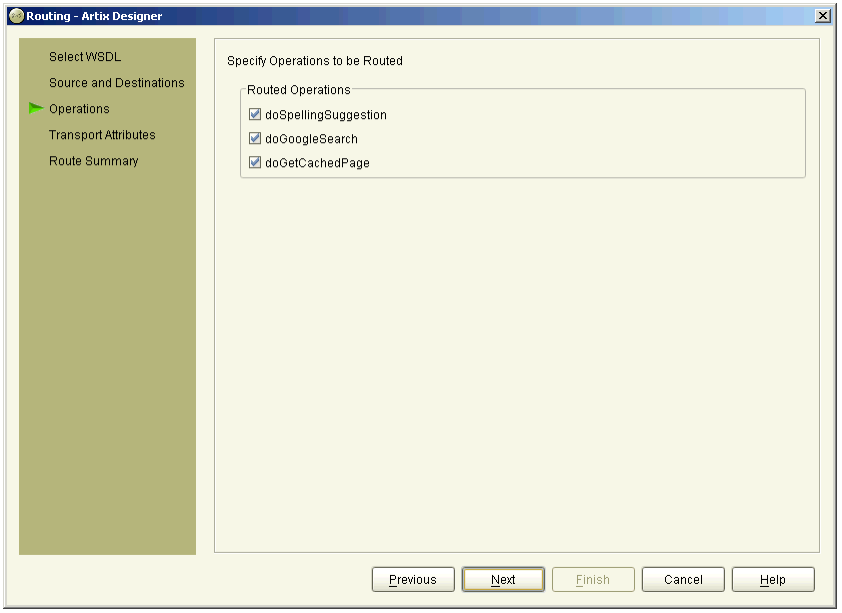

Figure 122:*New Route wizard—Operation Routing panel*

11. Select the operations you want to route from the list provided. By default, all operations are pre-selected.

12. Click **Next** to display the Transport Attributes panel, as shown in Figure 123.

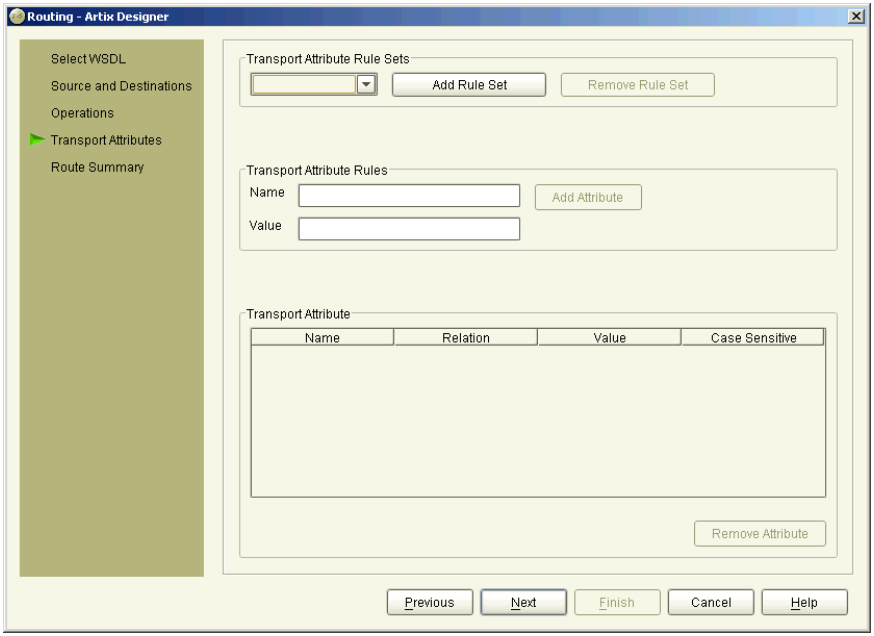

Figure 123:*New Route wizard—Transport Attributes panel*

13. Click **Add Rule Set** to add transport attribute based routing rules. The counter will automatically start at **0**.
14. Enter the name of the transport attribute.
15. Enter the value to be used for the attribute.
16. Click **Add Attribute** to add the attribute to the Transport Attribute table. When the attribute is in the table you can edit it to determine how matching attributes are compared to the value.
17. Repeat **steps 13-16** for all the attributes you want to use in your route.

Note: The editor has no knowledge of the valid attribute names and will allow you to enter any names and values.

18. Click **Next** to display the Summary panel, as shown in Figure 124.

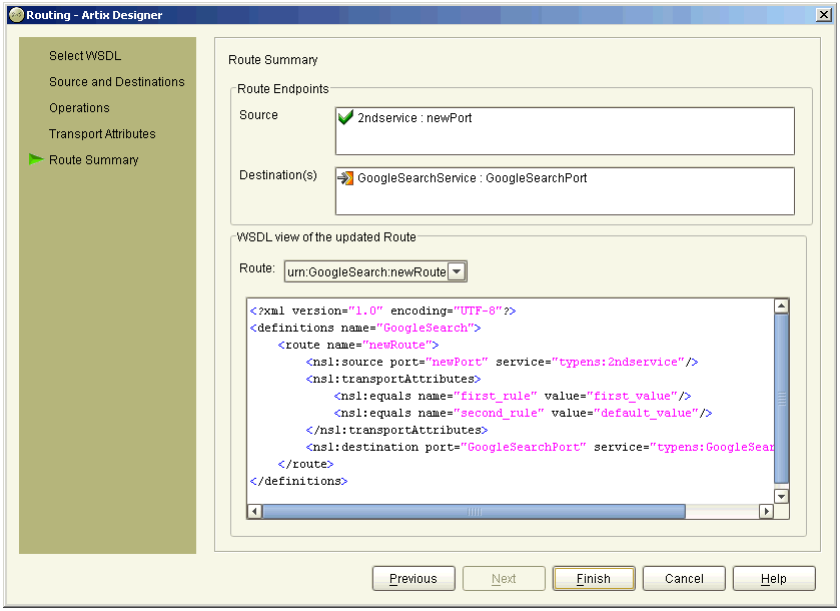

Figure 124:*New Route wizard—Summary panel*

19. Click **Finish** to close this wizard and return to the Artix Designer.

Editing a Route

The only things you can edit in a route are the transport attributes. When you choose to edit a route, the Transport Attributes panel for the New Route wizard is displayed for that route, enabling you to change any transport attributes that you previously created, or to add new transport attributes.

Procedure

To edit a route:

1. Select the Route in the Resource Navigator (diagram view) and select Resource | Edit | Route to display the Transport Attributes panel, as shown in Figure 125.

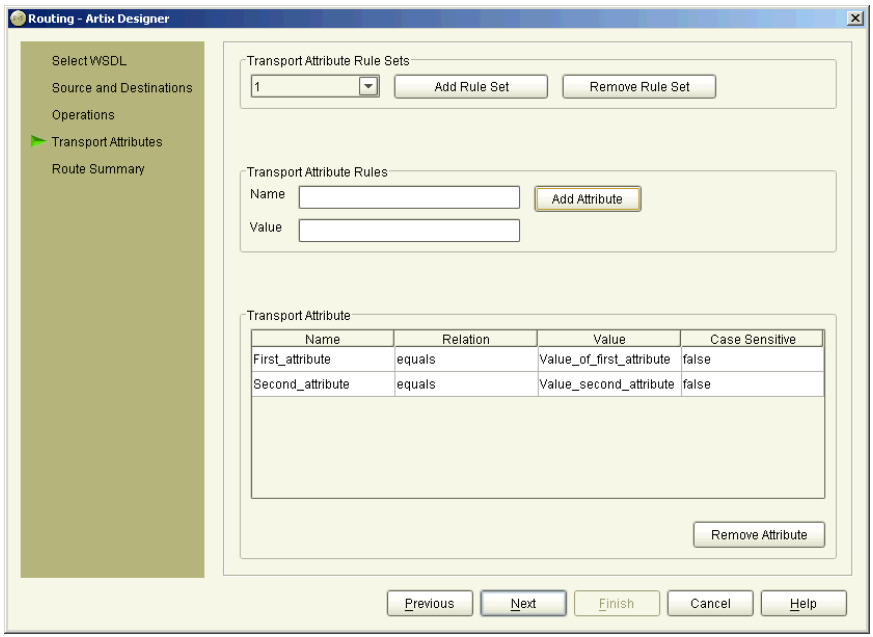

Figure 125: *Transport Attributes panel—Editing a Route*

2. You can change the values for any of the existing transport attributes, or add new transport attributes.

3. When you have finished your changes, click OK to display the Summary panel, as shown in Figure 126. This panel will display the route with your changes included.

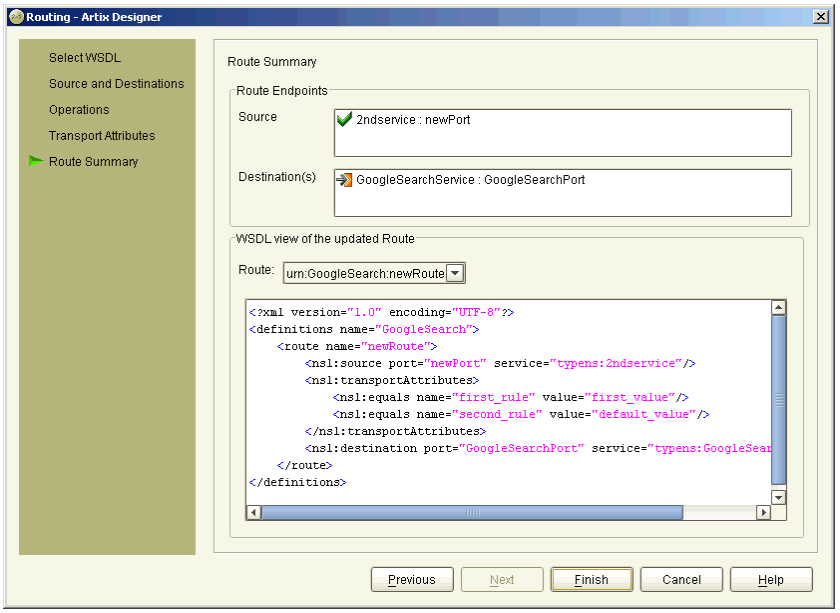

Figure 126:*Summary panel—Editing a Route*

CHAPTER 8

Deployment

You can generate code for your Artix collections as often as you like using different configurations to satisfy your solution requirements.

In this chapter

This chapter discusses the following topics:

Deployment Explained

Overview

Deployment is only available in the Artix Designer when it is in Deployer mode. If you are working with the Designer in Editor mode, there is no way to access the deployment functionality described in this chapter other than switching to Deployer mode.

Artix Collections can be deployed as Java, C++, or CORBA-based applications. As part of the deployment process, you can use a collection to create a client, a server, or a switch, or any combination of all three options.

Deployment involves three steps:

1. Creating a Deployment Profile - see page 179
2. Creating the Deployment Bundle - see page 183
3. Generating the Code - see page 192

You do not have to perform all the steps in one go - you can perform one or more and then complete the rest later. You must, however, perform them in the order shown here. That is, you need to create a Deployment Profile before you can create a Deployment Bundle, and you cannot generate the code until you have created a bundle.

One other thing you need to consider is that you can't create a deployment bundle for a collection if it doesn't contain a contract that has had a service defined. For more information on adding services to a contract, see page 139.

As part of the code generation process, Artix generates four directories in your specified save location:

- `src` - contains the generated source code in the language you specified (C++, Java, or IDL)
- `etc` - contains the configuration information required for the application to run successfully
- `wsdl` - contains the locally defined WSDL contracts
- `bin` - contains the environment scripts and the start and stop (UNIX only) scripts

Creating a Deployment Profile

Overview

The Deployment Profile defines machine level-information such as the Artix save location, the compiler location, and the operating system being used. This profile can be used multiple times as it is not specific to any particular collection defined within the workspace.

If you create your workspace using one of the workspace templates, Artix creates a default local profile for you automatically, which you can use for deploying to your local machine. The details of this profile are displayed on the Workspace Details panel. For deploying to other machines, you need to create your own profiles.

The next step after creating a Deployment Profile, is to create a Deployment Bundle to capture specific information about the deployment of a collection. Deployment Bundles are explained further in the next section of this chapter.

You can have as many Deployment Profiles as you like within your workspace, but each Deployment Bundle can reference only one Deployment Profile.

Procedure

To create a Deployment Profile:

1. Select **File | New | Deployment Profile** from the menu bar to display the Deployment Profile wizard, as shown in Figure 127.

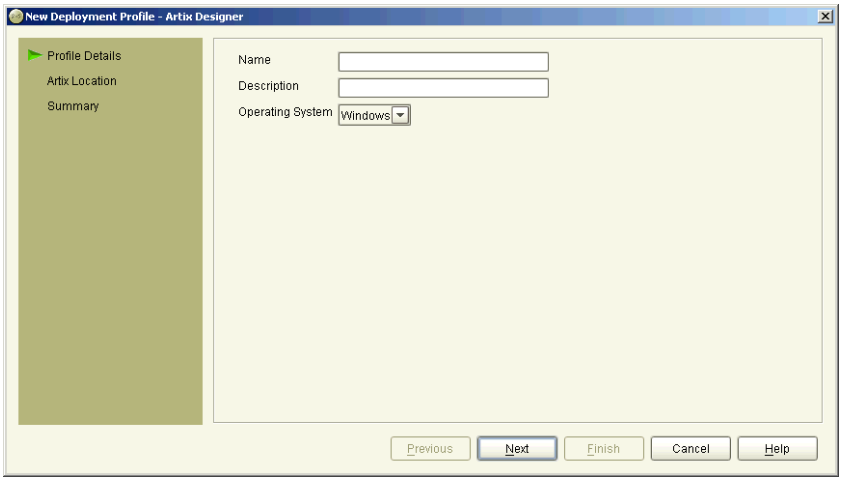

Figure 127:*Deployment Profile wizard*

2. Enter a name for this profile.
3. Enter a description for this profile to help distinguish it from other profiles you may create.
4. Select the operating system for this profile from the list provided. Artix currently supports Windows or UNIX.

5. Click **Next** to display the Artix Location panel, as shown in Figure 128.

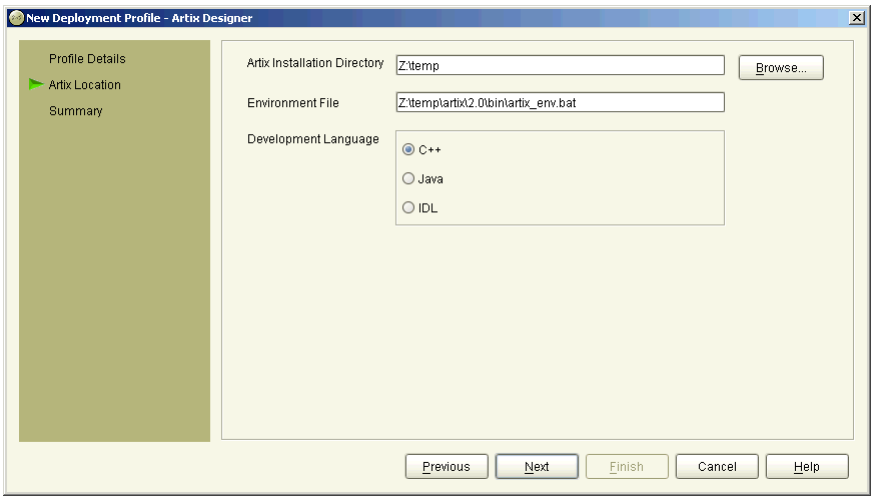

Figure 128:*Deployment Profile wizard—Artix Location panel*

6. Enter values for each of the fields provided on the panel, or accept the defaults provided. Changes you make to the **Location** field will be reflected in the **Environment File** field.
7. Select a **Development Language** for this profile from the options provided (**C++, Java, IDL**).

8. Click **Next** to display the Summary panel, as shown in Figure 129.

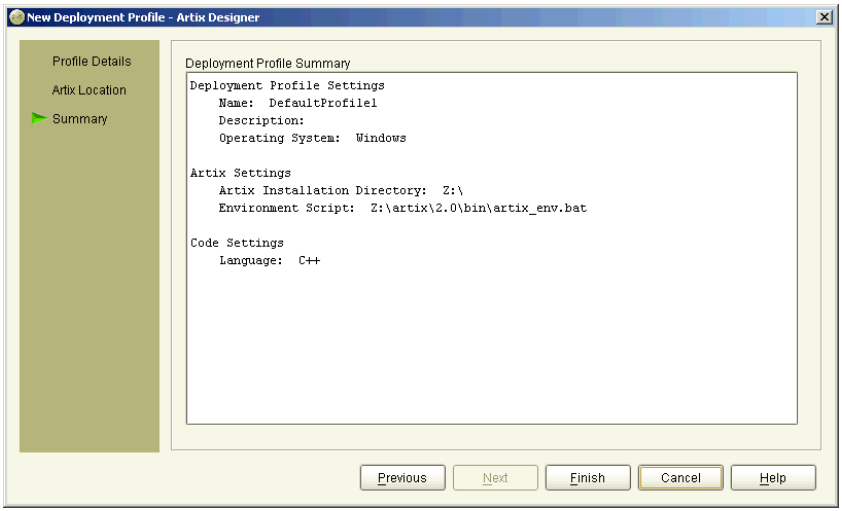

Figure 129:*Deployment Profile wizard—Summary panel*

9. Click **Finish** to close this wizard and return to the Artix Designer. Your new Deployment Profile is listed in the Designer Tree as well as on the Workspace Details panel.

Editing a Deployment Profile

After you have created a deployment profile, you can view its details by selecting it in the Designer Tree to display the Deployment Profile Details panel, as shown in Figure 130.

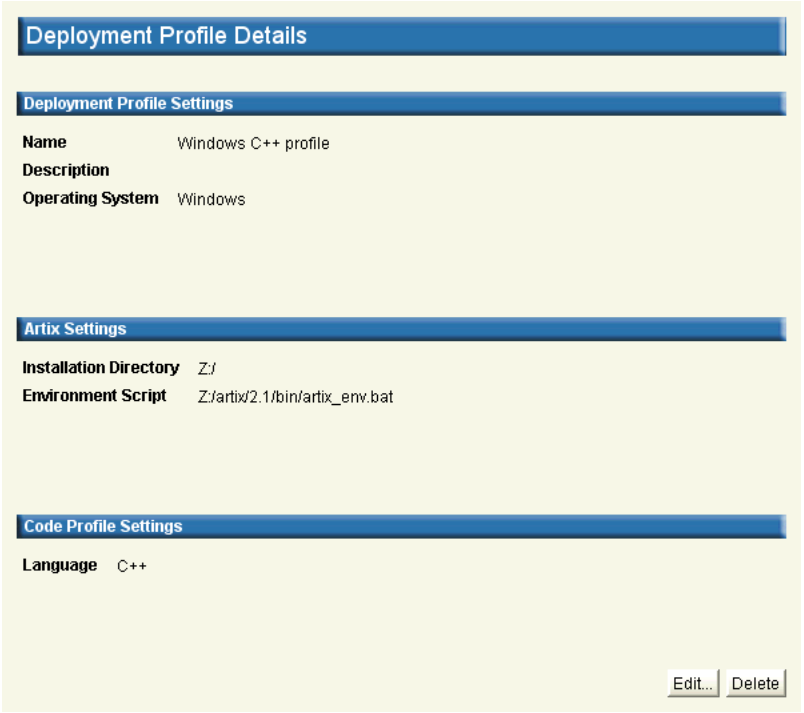

Figure 130:*Deployment Profile Details*

This panel displays, in read-only format, the settings you defined for the profile during the New Profile wizard.

To edit a profile click on the Edit button to display the Edit Deployment Profile dialog, as shown in Figure 131.

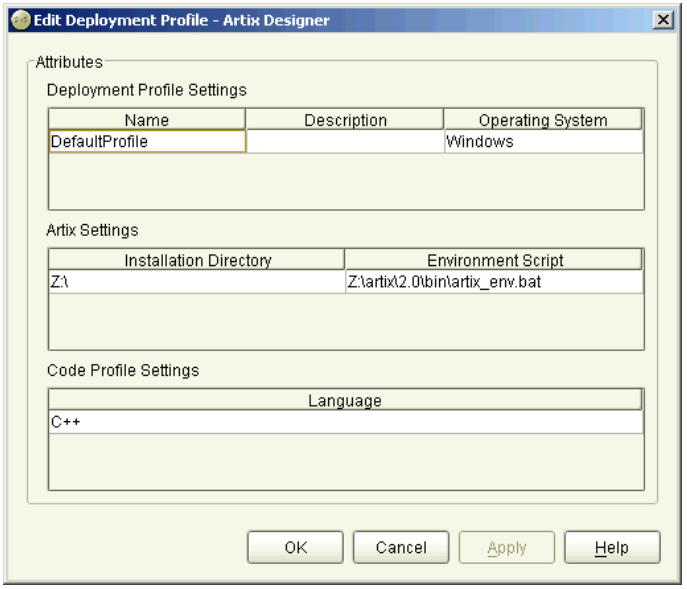

Figure 131:*Edit Deployment Profile dialog*

This dialog enables you to change the settings for your Deployment Profile.

To change the values for any of the settings, either:

- Type a new value in the cell, or
- Select a valid option from the drop-down within the cell

Click **Apply** to apply your changes, and **OK** to close this dialog.

Note that if you make changes to a Deployment Profile for a Bundle that has already had code generated, you will need to regenerate the code as it will most likely be rendered invalid. This is indicated by a warning icon to the left of the bundle name in the Designer Tree.

To regenerate the code for a bundle, select it in the Designer Tree and select Tools | Generate Code. See page 192 for more information.

Creating a Deployment Bundle

Overview

The Deployment Bundle defines the deployment characteristics for a collection, such as the deployment type (client, server, or switch), code generation options, and configuration details. You can also modify the service WSDL for each deployment bundle, if necessary.

You can have as many Deployment Bundles per collection as you like, but you must have at least one Deployment Profile created before you can create a Bundle. Typically you would create a new profile for each different operating system you intended using for deployment.

Procedure

To create a Deployment Bundle:

1. Select the Collection for which you want to create a bundle in the Designer Tree.
2. Select **File | New | Deployment Bundle** from the menu bar to display the New Deployment Bundle wizard, as shown in Figure 132.

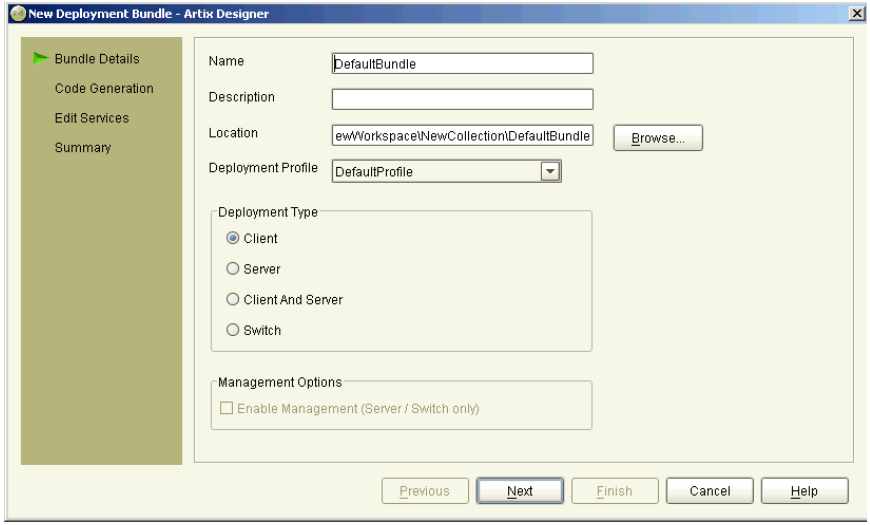

Figure 132: *New Deployment Bundle wizard*

3. Enter a name for this bundle, or accept the default provided.
4. Enter a description for this bundle to help distinguish it from other bundles you may create.
5. Enter a save location for this bundle, or accept the default provided.
6. Select a **Deployment Profile** to reference for this bundle. If there are no profiles listed, you need to create one before you can continue. See "Creating a Deployment Profile" on page 179 for more information.
7. Select the **Deployment Type** from the list provided - options are **client, server, client and server, or switch**.
8. Click **Next** to display the Code Generation panel, as shown in Figure 133.

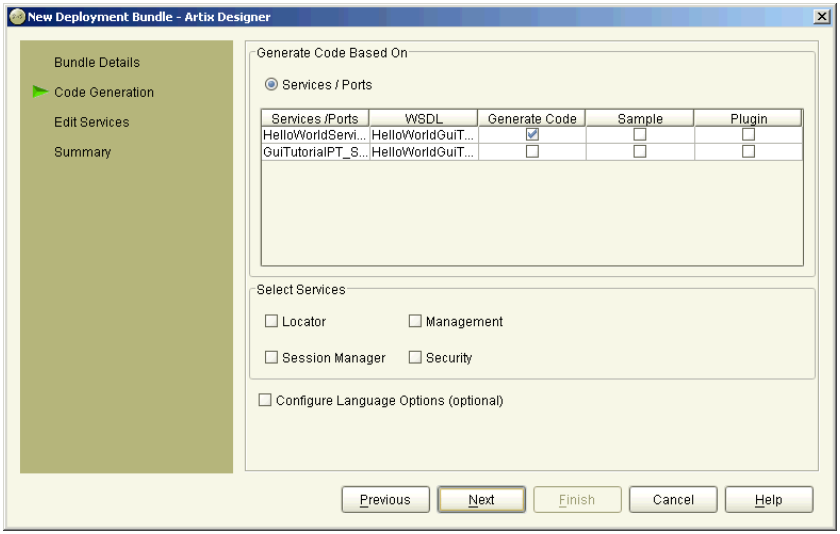

Figure 133:*Deployment Bundle wizard—Code Generation panel*

9. Select how you'd like to display the items for which you can generate code. The available options are:
   - Services/ports
   - Bindings
   - Port Types

10. Select the check boxes to indicate what type of code you wish to generate. Options are:
    - Code - generates proxy and stub code that is not user-editable - doesn't contain any starting point code.
    - Sample - generates starting point code with sample data that you can edit or add processing logic to. (Only available if Code is also checked.)
    - Plug-in - generates the code so that it compiles as an Artix plug-in. (Only available if Code is also checked.)
11. Select the Services you would like to enable for this deployment bundle from the list provided:
    - **Locator** - clients use the location service to detect other deployed Artix applications. Typically deployed into a standalone switch.
    - **Session Manager** - provides control over the number of clients that can connect to a service, and the duration of their connection. Typically deployed into a standalone switch.
    - **Management** - generates scripts required to manage your deloyment through a management console. (Only valid for server or switch deployments.)
    - **Security** - enables secure communication between the client and the web service using SSL.

    Note: These services are only available for selection if they have been enabled at the workspace level on the Workspace Services Details panel.
12. Click the **Configure Language Options** check box if you want to specify settings for the code generation in the language you have selected. This will add an extra panel to the wizard where you can specify these options.

    Depending on which code you are working with, the options displayed will differ. Possible options are:
    - Namespace (C++) - The namespace you want to use in the C++ code.

- Declaration Specification (C++) - If this collection is being built as a DLL on Windows, this specifier is required for the symbols exported from the library.
- Package Name (Java) - The name you want to use for the Java package. Enter a name or accept the default provided.

13. Click **Next** to display the Edit Services panel, as shown in Figure 134.

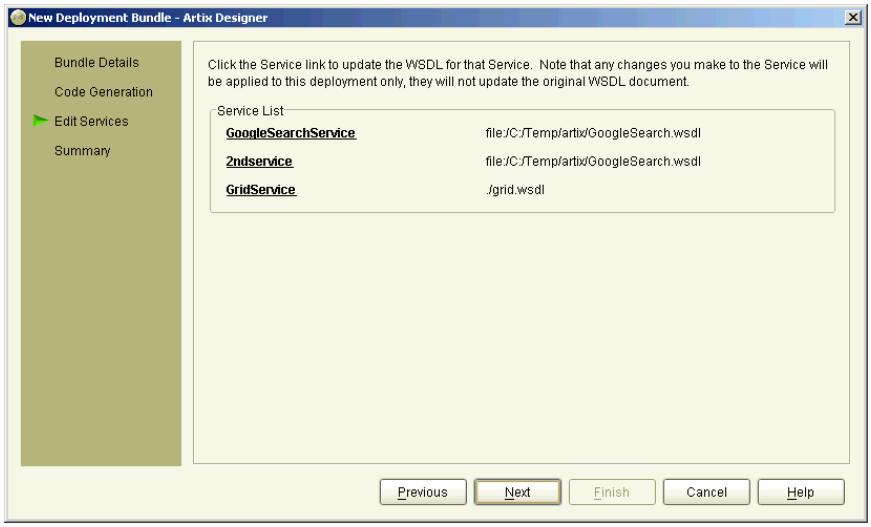

Figure 134:*Deployment Bundle wizard—Update Service panel*

14. Click any of the Services links to update them for this deployment.

    Note that any changes made to the Service details from within this wizard will only apply to that bundle; they will not be applied to the WSDL document itself.

15. Click **Next** to display the Summary panel, as shown in Figure 135.

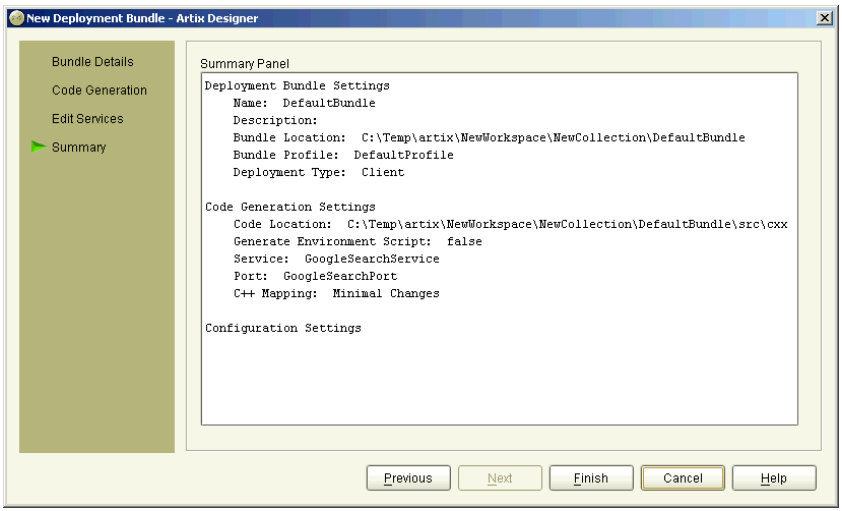

Figure 135:*Deployment bundle wizard—Summary panel*

16. Click **Finish** to close this wizard and return to the Artix Designer. The new Deployment Bundle is listed in the Designer Tree as well as on the Collection Details panel.

Deployment bundle status

The status of your deployment bundle can change over its lifetime. After you have generated code for your bundle, its status is indicated on the Collection and Deployment Bundle Details panels. The date and time that the code was generated is stated.

If you subsequently make changes to any of the collection entitites, such as the Deployment Profile, or the Bundle, or even the resources, the code will most likely no longer be valid. You will need to regenerate it. This is indicated on the Details panels, as before, and also by a warning icon next to the bundle name in the Designer Tree.

Editing a Deployment Bundle

After you have created a deployment bundle, you can view its details by selecting it in the Designer Tree to display the Deployment Bundle Details panel, as shown in Figure 130.

Figure 136:*Deployment Bundle Details*

This panel displays, in read-only format, the settings you defined for the profile during the New Bundle wizard.

To edit the bundle by clicking on the Edit button to display the Edit Deployment Bundle dialog, as shown in Figure 131.

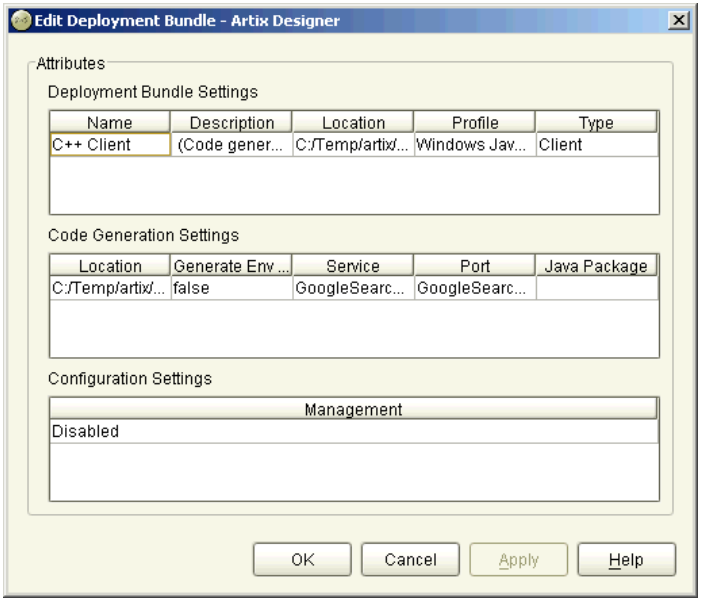

Figure 137:*Edit Deployment Bundle dialog*

To change the values for any of the settings, either:

- Type a new value in the cell, or
- Select a valid option from the drop-down within the cell

Click **Apply** to apply your changes, and **OK** to close this dialog.

Note that if you make changes to a Deployment Bundle that has already had code generated, you will need to regenerate the code as it will most likely be rendered invalid. This is indicated by a warning icon to the left of the bundle name in the Designer Tree.

To regenerate the code for a bundle, select it in the Designer Tree and select Tools | Generate Code. See the next section for more information.

Generating Code

Overview

Once you have created your Deployment Bundle, code generation for the collection is very simple and quick. Artix generates the code based on the information you provided in the bundle, and creates the code, environment scripts, and configuration files in the locations you provided.

Note that if you make any changes to any of the contents of a collection after code generation, it could invalidate that code. For this reason, it is recommended that you regenerate the code for any collection that has been modified. Bundles requiring their code to be regenerated are indicated in the Designer Tree by a small warning icon next to the Bundle name.

Procedure

To generate code for a collection:

1. Select the collection in the Designer Tree.
2. Select **Tools | Generate Code** from the menu bar to display the Generate Code dialog, as shown in Figure 138.

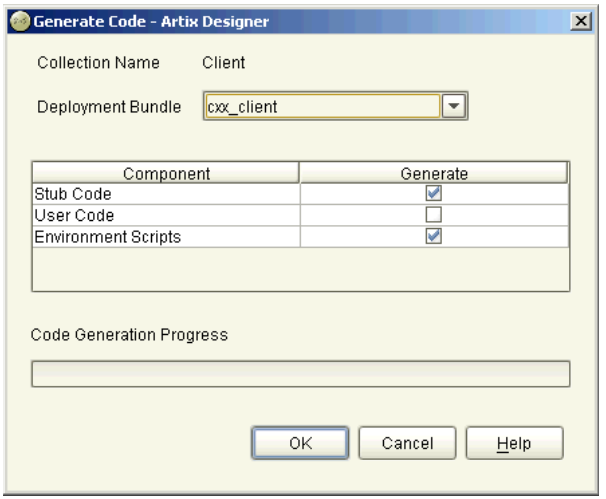

Figure 138:*Generate Code dialog*

3. Select a Deployment Bundle from the **Deployment Bundle** drop-down list.
4. There are several components already pre-selected in the **Generate** column - if you want to change any of these settings you can do so by selecting and deselecting check boxes. The options provided are:
   - Stub Code - code that marshals and de-marshals the request. This is Required, and is always pre-selected.
   - User Code - generates a template for the implementation code. You will need to complete this code by hand.
   - Environment Scripts - scripts required to set up the Artix development and runtime on the deployment machine.
   - Start/Stop Scripts - scripts to start and stop the server. Only valid for server and switch implementations. For Windows operating systems there will be no stop scripts, as Artix is unable to determine which process to stop from the command line API.
   - Management Scripts - scripts required for integration into the BMC console. Only valid for server and switch implementations.
5. Click **OK** to generate the code for this bundle.

   You will see a progress indicator, and messages stating things like "Generating Code", "Generating Configuration File". When the process is complete (usually only 3-4 seconds), you will receive a message stating that it is "Finished".

Testing the solution

Now that you have generated the code necessary for your application, you need to do some hand coding before you can test the solution.

For help with editing your code, see either:

- *Developing Artix Solutions with C++*; or
- *Developing Artix Solutions with Java*

These books will guide you through the steps required to get your code to a state where it is ready to run.

Part II

Using Artix Command Line Tools

In this part

This part contains the following chapters:

CHAPTER 9

Designing Artix Solutions from the Command Line

Using a combination of Artix command line utilities and a text editor you can create complex Artix solutions.

In this chapter

This chapter discusses the following topics:

Artix and WSDL

Overview

When designing Artix solutions from the command line, you will be working directly with the WSDL and XMLSchema that makes up the Artix contract. In many instances the Artix designer automates many of the details of creating a well-formed and valid WSDL document. When hand-editing Artix contracts you will need to ensure that the contract is valid, as well as correct. To do that you must have some familiarity with WSDL. You can find the standard on the W3C website, www.w3.org.

Structure of a WSDL document

A WSDL document is, at its simplest, a collection of elements contained within the root `<definition>` element. These elements describe a service and how that service can is accessed.

The `<types>`, `<message>`, and `<portType>` elements describe the service's interface and make up the *logical* section of a contract. Within the `<types>` element, XMLSchema is used to define complex data types. A number of `<message>` elements are used to define the structure of the messages used by the service. The `<portType>` element contains one or more `<operation>` elements that define the operations provided by the service.

The `<binding>` and `<service>` elements describe how the service connects to the outside world and make up the *physical* section of the contract. `<binding>` elements describe how the data defined in the `<message>` elements are mapped into a concrete on-the-wire data format, such as SOAP. `<service>` elements contain one or more `<port>` elements which define the network interface for the service.

WSDL elements

A WSDL document is made up of the following elements:

- `<definitions>` - the root element of a WSDL document. The attributes of this element specfiy the name of the WSDL document, the document's target namespace, and the shorthand definitions for the namespaces referenced by the WSDL.
- `<types>` – the definition of complex data types based on in-line type descriptions and/or external definitions such as those in an XMLSchema document (XSD).

- <message> – the abstract definition of the data being communicated.
- <portType> – a collection of <operation> elements representing an abstract endpoint.
- <operation>– the abstract description of an action.
- <binding> – the concrete data format specification for a port type.
- <service> – a collection of <port> elements.
- <port> – the endpoint defined by a binding and a physical address.

Artix extensions

Artix extends the original concept of WSDL by expanding it to describe services that use transports and bindings beyond SOAP over HTTP. Artix also extends WSDL to allow it to describe complex systems of services and how they are integrated. To do this IONA has extended WSDL according to the procedures outlined by W3C.

The majority of the IONA WSDL extension elements are used in the physical section of the contract because they relate to how data is mapped into an on the wire format and how different transports are configured. In addition, Artix defines extensions for creating routes between services, CORBA data type mapping, and working with service references.

Each extension is defined in a separate namespace and IONA provides the XMLSchema definitions for each extension so that any XML editor can validate an Artix contract.

Creating an Artix Contract

Overview

The process of designing an Artix solution using command line tools is similar to the process of designing an Artix solution using Artix Designer. You still need to define all of the information that defines the logical and physical characteristics of the services in your system and how they interact.

Design process

To design an Artix solution from the command line you must perform the following steps:

1. Define the data types used in your solution.
2. Define the messages used in your solution.
3. Define the interfaces for each of the services in your solution.
4. Define the bindings between the messages used by each interface and the concrete representation of the data on the wire.
5. Define the transport details for each of the services in your solution.
6. Define any routing rules used in your solution.
7. Define any Artix services used to provide added functionality to your solution.

Using composite contracts

When using the command line tools you can choose to design your project using a single contract that contains all of the type definitions, interface definitions, bindings, ports, and other Artix-specific information defining your solution. You can also choose to work with a number of smaller contracts that you import into a composite contract that represents the solution.

This approach allows you to reuse parts of your contract, such as the data type definitions, in multiple projects. It can also make working with large contracts more manageable.

Beyond the Contract

Overview

After you have created the contract defining your Artix solution, you still have work to do before your solution is ready to go. There are two remaining steps in developing a solution using Artix:

1. Develop any application-level code needed to complete the solution.
2. Configure the Artix components.

Develop application code

Often, you will need to develop new application logic as a part of your solution. Artix provides tools that allow you to develop this new functionality using familiar programming paradigms. For example, if you are a CORBA developer integrating a CORBA system with a Tuxedo application, Artix will generate the IDL representing the interface used in the service integration. You can then implement the interface using CORBA.

Artix also provides code generators to create stub and skeleton code in C++ and Java. The APIs used by Artix make it easy to develop transport-independent, Web services-based applications using standard programming techniques. For more information on developing Artix applications, see *Developing Artix Applications in C++* or *Developing Artix Applications in Java*.

Configure the Artix components

Before deploying your Artix solution you need to configure the run time environment for your Artix components and services. For a detailed discussion of Artix configuration, see the *Deploying and Managing Artix Solutions*.

CHAPTER 10

Defining Data Types

In Artix, complex data types are defined using XMLSchema.

Overview

When defining an interface in an Artix contract, the first thing you need to consider is the types of data that are used by the operation parameters of the interface. Artix uses XMLSchema as its native type system. XMLSchema supports a number of simple types that do not require you to describe them in the contract. XMLSchema also supports the definition of complex data types that are either a collection of typed elements or a derivative of a simple type. In an Artix contract, complex type definitions are entered in the `<type>` element.

Defining the types used in an Artix contract involves seven steps:

1. Determine all of the data types used in the interface described by the contract.
2. Create a `<type>` element in your contract.
3. Create a `<schema>` element, as a child of the `<type>` element, that specifies the type system used in the contract. See "Specifying a Type System in a Contract" on page 205.
4. For each complex type that is a collection of elements, define the data type using a `<complexType>` element. See "Defining Data Structures" on page 209.

5. For each array, define the data type using a `<complexType>` element. See "Defining Arrays" on page 212.
6. For each complex type that is derived from a simple type, define the data type using a `<simpleType>` element. See "Defining Types by Restriction" on page 214.
7. For each enumerated type, define the data type using a `<simpleType>` element. See "Defining Enumerated Types" on page 216.

In this chapter

This chapter discusses the following topics:

Specifying a Type System in a Contract

Overview

According to the WSDL specification, you can use any type system you like to define data types in WSDL. However, the W3C specification states XMLSchema (XSD) is the preferred canonical type system for a WSDL document. Therefore, XSD is the intrinsic type system in Artix.

Specifying the type system

The first child element of the `<types>` element in a contract is the `<schema>` element. This element specifies the namespace for the types defined by the WSDL. It also defines the type system used to define the new types and any namespaces that are referenced in the type definitions.

Example 1 shows the standard `<schema>` element for an Artix contract. The attribute `targetNamespace` is where you specify the namespace under which your new data types are defined. The remaining entries are required. The first specifies that the types are defined using XMLSchema. The second references a few special XMLSchema types defined specifically for WSDL.

Example 1: *Artix Schema Element*

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

General guidelines

The W3C also provides guidelines on using XMLSchema to represent data types in WSDL documents:

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

XMLSchema Simple Types

Overview

If a message part is going to be of a simple type you do not need to create a type definition for it. However, the complex types used by the interfaces defined in the contract are defined using simple types.

Entering simple types

XSD simple types are mainly placed in the `type` attribute of `<element>` elements used in defining sequences in the types section of your contract. They are also used in the `base` attribute of `<restriction>` elements and `<extension>` elements.

Simple types are always entered using the `xsd` prefix. For example, to specify that an element is of type `int`, you would enter `xsd:int` in its `type` attribute.

Supported XSD simple types

Artix supports the following XMLSchema simple types:

- `xsd:anyType`
- `xsd:base64Binary`
- `xsd:boolean`
- `xsd:byte`
- `xsd:dateTime`
- `xsd:decimal`
- `xsd:double`
- `xsd:float`
- `xsd:hexBinary`
- `xsd:int`
- `xsd:integer`
- `xsd:long`
- `xsd:QName`
- `xsd:short`
- `xsd:string`
- `xsd:unsignedByte`
- `xsd:unsignedInt`
- `xsd:unsignedLong`

- xsd:unsignedShort
- xsd:integer
- xsd:positiveInteger
- xsd:negativeInteger
- xsd:nonPositiveInteger
- xsd:nonNegativeInteger
- xsd:ID
- xsd:anyURI
- xsd:gDay
- xsd:gMonth
- xsd:gYear
- xsd:gMonthDay
- xsd:gYearMonth

Defining Complex Data Types

Overview

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

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

In this section

This section discusses the following topics:

Defining Data Structures

Overview

In XMLSchema data structures that are a collection of data fields are defined using `<complexType>` elements. The definition of a `<complexType>` has three parts:

1. The name of the defined type is specified in the `name` attribute of the `<complexType>` element.
2. The first child element of the `<complexType>` describes the behavior of the structure's fields when it is put on the wire. See "complexType varieties" on page 210.
3. Each of the fields of the defined structure are defined in `<element>` elements that are grandchildren of the `<complexType>`. See "Defining the parts of a structure" on page 210.

For example the structure shown in Example 2 would be defined in XMLSchema as a `<complexType>` with two elements.

Example 2: *Simple Structure*

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

Example 3 shows one possible XMLSchema mapping for `personalInfo`.

Example 3: *A Complex Type*

```
<complexType name="personalInfo>
  <sequence>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </sequence>
</complexType>
```

complexType varieties

XMLSchema has three ways of describing how the fields of a complex type are organized when represented as an XML document and when passed on the wire. The first child element of the `<complexType>` determines which variety of complex type is being used. Table 1 shows the elements used to define complex type behavior.

Table 1: *complexType Descriptor Elements*

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

If neither `<sequence>`, `<all>`, nor `<choice>` is specified, the default is `<sequence>`.

For example, the structure defined in Example 3 would generate a message containing two elements: `name` and `age`. If the structure was defined as a `<choice>`, as shown in Example 4, it would generate a message with either a `name` element or an `age` element.

Example 4: *Simple Complex Choice Type*

```
<complexType name="personalInfo>
  <choice>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" />
  </choice>
</complexType>
```

Defining the parts of a structure

You define the data fields that make up a structure using `<element>` elements. Every `<complexType>` should have at least one `<element>` defined inside of it. Each `<element>` in the `<complexType>` represents a field in the defined data structure.

To fully describe a field in a data structure, `<element>` elements have two required attributes:

- `name` specifies the name of the data field and must be unique within the defined complex type.
- `type` specifies the type of the data stored in the field. The type can be either one of the XMLSchema simple types or any named complex type that is defined in the contract.

In addition to `name` and `type`, `<element>` elements have two other commonly used optional attributes: `minOcurrs` and `maxOccurs`. These attributes place bounds on the number of times the field occurs in the structure. By default, each field occurs only once in a complex type. Using these attributes, you can change how many times a field must or can appear in a structure. For example, you could define a field, `previousJobs`, that must occur at least three times and no more than seven times as shown in Example 5.

Example 5: *Simple Complex Type with Occurrence Constraints*

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

You could also use `minOccurs` to make a date field optional by setting it to zero as shown in Example 6. In this case `age` can be omitted and the data will still be valid.

Example 6: *Simple Complex Type with minOccurs*

```
<complexType name="personalInfo>
  <choice>
    <element name="name" type="xsd:string" />
    <element name="age" type="xsd:int" minOccurs="0" />
  </choice>
</complexType>
```

Defining Arrays

Overview

Artix supports two methods for defining arrays in a contract. The first is define a complex type with a single element with occurrence constraint placed on it. The second is to use SOAP arrays. SOAP arrays provide added functionality such as the ability to easily define multi-dimensional arrays and transmit sparsely populated arrays.

Complex type arrays

Complex type arrays are nothing more than a special case of a `<sequence>` complex type. You simply define a complex type with a single element and specify a value for the `maxOccurs` attribute. For example to define an array of twenty `float`s you would use a complex type similar to the one shown in Example 7.

Example 7: *Complex Type Array*

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

You could also specify a value for `minOccurs`.

SOAP arrays

SOAP arrays are defined by deriving from the `SOAP-ENC:Array` base type using the `wsdl:arrayType`. The syntax for this is shown in Example 8.

Example 8: *Syntax for a SOAP Array derived using wsdl:arrayType*

```
<complexType name="TypeName">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="ElementType<ArrayBounds>"/>
    </restriction>
  </complexContent>
</complexType>
```

Using this syntax, *TypeName* specifies the name of the newly-defined array type. *ElementType* specifies the type of the elements in the array. *<ArrayBounds>* specifies the number of dimensions in the array. To specify a single dimension array you would use `[]`; to specify a two-dimensional array you would use either `[][]` or `[,]`.

For example, the SOAP Array, `SOAPStrings`, shown in Example 9, defines a one-dimensional array of strings. The `wsdl:arrayType` attribute specifies the type of the array elements, `xsd:string`, and the number of dimensions, `[]` implying one dimension.

Example 9: *Definition of a SOAP Array*

```
<complexType name="SOAPStrings">
  <complexContent>
    <restriction base="SOAP-ENC:Array">
      <attribute ref="SOAP-ENC:arrayType"
                 wsdl:arrayType="xsd:string[]"/>
    </restriction>
  </complexContent>
</complexType>
```

You can also describe a SOAP Array using a simple element as described in the SOAP 1.1 specification. The syntax for this is shown in Example 10.

Example 10: *Syntax for a SOAP Array derived using an Element*

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

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

Defining Types by Restriction

Overview

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

The definition of a `<simpleType>` has three parts:

1. The name of the new type is specified by the `name` attribute of the `<simpleType>` element.
2. The simple type from which the new type is derived, called the *base type*, is specified in the `<restriction>` element. See "Specifying the base type" on page 214.
3. The rules, called *facets*, defining the restrictions placed on the base type are defined as children of the `<restriction>` element. See "Defining the restrictions" on page 215.

Specifying the base type

The base type is the type that is being restricted to define the new type. It is specified using a `<restriction>` element. The `<restriction>` element is the only child of a `<simpleType>` element and has one attribute, base, that specifies the base type. The base type can be any of the XMLSchema simple types.

For example, to define a new type by restricting the values of an `xsd:int` you would use a definition like Example 11.

Example 11: *int as Base Type*

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

Defining the restrictions

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

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

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

Example

Example 12 shows an example of a simple type, `SSN`, which represents a social security number. The resulting type will be a string of the form `xxx-xx-xxxx`. `<SSN>032-43-9876<SSN>` is a valid value for an element of this type, but `<SSN>032439876</SSN>` is not.

Example 12: *SSN Simple Type Description*

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

Defining Enumerated Types

Overview

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

Defining an enumeration

The syntax for defining an enumeration is shown in Example 13.

Example 13: *Syntax for an Enumeration*

```
<simpleType name="EnumName">
  <restriction base="EnumType">
    <enumeration value="Case1Value" />
    <enumeration value="Case2Value" />
    ...
    <enumeration value="CaseNValue" />
  </restriction>
</simpleType>
```

EnumName specifies the name of the enumeration type. *EnumType* specifies the type of the case values. *CaseNValue*, where *N* is any number one or greater, specifies the value for each specific case of the enumeration. An enumerated type can have any number of case values, but because it is derived from a simple type, only one of the case values is valid at a time.

Example

For example, an XML document with an element defined by the enumeration `widgetSize`, shown in Example 14, would be valid if it contained `<widgetSize>big</widgetSize>`, but not if it contained `<widgetSize>big,mungo</widgetSize>`.

Example 14: *widgetSize Enumeration*

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

CHAPTER 11

Defining Messages

You can define complex messages to pass between your services.

Overview

WSDL is designed to describe how data is passed over a network and because of this 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 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 15.

Example 15: *personalInfo lookup Method*

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

This method signature could be mapped to the WSDL fragment shown in Example 16.

Example 16: *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 that represent a service request or output messages that represent a response.

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 datatype of the part is defined by a `simpleType` or `complexType` called `type_name`
`element="elem_name"`	The datatype 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 17.

Example 17: *Reused Part*

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

CHAPTER 12

Defining Your Interfaces

In WSDL documents interfaces are defined using the <portType> element.

Overview

Interfaces are defined using the WSDL `<portType>` element. Like an interface, the `<portType>` is a collection of operations that define the input, output, and fault messages used by the service implementing the interface to complete the transaction the operation describes. The difference is that the operations in a port type are built up using messages that are defined outside of the port type instead of parameter lists defined as part of the operation itself.

To define an interface, port type, in an Artix contract do the following:

1. Create a `<portType>` element to contain the interface definition and give it a unique name. See "Port types" on page 224.
2. Create an `<operation>` element for each operation defined in the interface. See "Operations" on page 224.
3. For each operation, specify the messages used represent the operation's parameter list, return type, and exceptions. See "Operation messages" on page 224.

Port types

A <portType> element is the root element in an interface definition and many Web service implementations, including Artix, map port types directly to generated implementation objects. In addition, the <portType> element is 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.

Each <portType> element in a WSDL document must have a unique name, specified using the name attribute, and is made up of a collection of operations, described in <operation> elements. A WSDL document can describe any number of port types.

Operations

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 defined within a <portType> element must have a unique name, specified using the name attribute. The name attribute is required to define an operation.

Operation messages

Operations are 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 the message the client endpoint sends to the service provider when a request is made. The parts of this message correspond to the input parameters of the operation.
<output>	Specifies the message that the service provider sends to the client endpoint in response to a request. The parts of this message correspond to any operation parameters that can be changed by the service provider, such as values passed by reference. This includes the return value of the operation.

Table 3: *Operation Message Elements*

Element	Description
<fault>	Specifies a message used to communicate an error condition between the endpoints.

An operation is required to have at least one input or one output element. An operation can have both input and output elements, but it can only have one of each. Operations are not required to have any fault messages, but can have any number of fault messages needed.

The elements are defined by two attributes listed inTable 4.

Table 4: *Attributes of the Input and Output Elements*

Attribute	Description
name	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
message	Specifies the abstract message that describes the data being sent or received. The value of the message attribute must correspond to the name attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the name attribute for all input and output elements; WSDL provides a default naming scheme based on the enclosing operation's name. If only one element is used in the operation, the element name defaults to the name of the operation. If both an input and an output element are used, the element name defaults to the name of the operation with Request or Response respectively appended to the name.

Return values

Because the <operation> element is an abstract definition of the data passed during 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 "Binding Interfaces to a Payload Format" on page 227.

Example

For example, you might have an interface similar to the one shown in Example 18.

Example 18: *personalInfo lookup interface*

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

This interface could be mapped to the port type in Example 19.

Example 19: *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>
```

Note that the return value of `lookup()` is mapped to the message used in the `<output>` element of the WSDL definition. Because the operation does not have any other parameters that can be returned, such as `out` or `inout` parameters in CORBA, the return parameter is the only part of the message used for the `<output>` element.

CHAPTER 13

Binding Interfaces to a Payload Format

You can bind your interfaces to a number of payload formats in Artix.

Overview

To define an endpoint that corresponds to a running service, port types are mapped to bindings that describe how the abstract messages used by the interface's operations map to the data format used on the wire. These 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.

In this Chapter

This chapter discusses the following topics:

Adding a SOAP Binding

Overview

Artix provides a tool to generate a default SOAP binding which does not use any SOAP headers. However, you can add SOAP header s to your binding using any text or XML editor. In addition, you can define a SOAP binding that uses MIME multipart attachments.

For more information

For more detailed information on the SOAP binding and the specifics of the elements used in defining it see "SOAP Binding Extensions" on page 427.

In this section

This section discusses the following topics:

Adding a Default SOAP Binding

Overview

Artix provides a command line tool, `wsdltosoap`, that will generate a default SOAP binding for an interface defined in a WSDL `<portType>`. The tool will generate a new contract which includes the generated SOAP binding.

Using the tool

To generate a SOAP binding using `wsdltosoap` use the following command:

```
wsdltosoap -i portType -n namespace wsdl_file
           [-b binding][-d dir][-o file]
           [-style {document|rpc}][-use {literal|encoded}]
```

The command has the following options:

`-i` *portType*	Specifies the name of the port type being mapped to a SOAP binding.
`-n` *namespace*	Specifies the namespace to use for the SOAP binding.
`-b` *binding*	Specifies the name for the generated SOAP binding. Defaults to *portType*`Binding`.
`-d` *dir*	Specifies the directory into which the new WSDL file is written.
`-o` *file*	Specifies the name of the generated WSDL file. Defaults to *wsdl_file*`-soap.wsdl`.
`-style`	Specifies the encoding style to use in the SOAP binding. Defaults to `document`.
`-use`	Specifies how the data is encoded. Default is `literal`.

`wsdltosoap` does not support the the generatoin of `document`/`encoded` SOAP bindings.

Example

If your system had an interface that took orders and offered a single operation to process the orders it would be defined in an Artix contract similar to the one shown in Example 20.

Example 20: *Ordering System Interface*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int" />
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
...
</definitions>
```

The SOAP binding generated for `orderWidgets` is shown in Example 21.

Example 21: *SOAP Binding for orderWidgets*

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="rpc"/>
      <input name="order">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      </input>
      <output name="bill">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      </output>
      <fault name="sizeFault">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      </fault>
  </operation>
</binding>
```

This binding specifies that messages are sent using the `rpc/encoded` message style. The value of the `namespace` attribute is, in this example, the same as the contract's target namespace.

Adding SOAP Headers to a SOAP Binding

Overview

SOAP headers are defined by adding `<soap:header>` elements into your default SOAP binding. The `<soap:header>` element is an optional child of the `<input>`, `<output>`, and `<fault>` elements of the binding. The SOAP header becomes part of the parent message. A SOAP header is defined by specifying a message and a message part. Each SOAP header can only contain one message part, but you can insert as many SOAP headers as needed.

Syntax

The syntax for defining a SOAP header is shown in Example 22. The `message` attribute of `<soap:header>` is the qualified name of the message from which the part being inserted into the header is taken. The `part` attribute is the name of the message part inserted into the SOAP header. Because SOAP headers are always doc style, the WSDL message part inserted into the SOAP header must be defined using an element. Together the `message` and the `part` attributes fully describe the data to insert into the SOAP header.

Example 22: *SOAP Header Syntax*

```
<binding name="headwig">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="weave">
      <soap:operation soapAction="" style="rpc"/>
      <input name="grain">
        <soap:body ... />
        <soap:header message="QName" part="partName" />
      </input>
...
</binding>
```

In addition to the mandatory `message` and `part` attributes, `<soap:header>` also supports the `namespace`, the `use`, and the `encodingStyle` attributes. These optional attributes function the same for `<soap:header>` as they do for `<soap:body>`.

Development considerations

When you are using SOAP headers in your Artix applications, you are responsible for creating and populating the SOAP headers in your application logic. For details on Artix application development, see either *Developing Artix Applications in C++* or *Developing Artix Applications in Java*.

Splitting messages between body and header

The message part inserted into the SOAP header can be any valid message part from the contract. It can even be a part from the parent message which is being used as the SOAP body. Because it is unlikely that you would want to send information twice in the same message, the SOAP binding provides a means for specifying the message parts that are inserted into the SOAP body.

The `<soap:body>` element has an optional attribute, `parts`, that takes a space delimited list of part names. When `parts` is defined, only the message parts listed are inserted into the SOAP body. You can then insert the remaining parts into the SOAP header.

Note: When you define a SOAP headers using parts of the parent message, Artix automatically fills in the SOAP headers for you.

Example

Example 23 shows a modified version of the orderWidgets service shown in Example 20. This version has been modified so that each order has an `xsd:base64binary` value placed in the SOAP header of the request and response. The SOAP header is defined as being the `keyVal` part from the `widgetKey` message. In this case you would be responsible for adding the SOAP header in your application logic because it is not part of the input or output message.

Example 23: *SOAP Binding for orderWidgets with a SOAP Header*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
```

Example 23: *SOAP Binding for orderWidgets with a SOAP Header*

```
<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary" />
  </schema>
</types>
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int" />
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int" />
</message>
<message name="widgetKey">
  <part name="keyVal" element="xsd1:keyElem" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
```

Example 23: *SOAP Binding for orderWidgets with a SOAP Header*

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="rpc"/>
      <input name="order">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
        <soap:header message="tns:widgetKey" part="keyVal" />
      </input>
      <output name="bill">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
        <soap:header message="tns:widgetKey" part="keyVal" />
      </output>
      <fault name="sizeFault">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

You could modify Example 23 so that the header value was a part of the input and output messages as shown in Example 24. In this case `keyVal` is a part of the input and output messages. In the `<soap:body>` elements the `parts` attribute specifies that `keyVal` is not to be inserted into the body. However, it is inserted into the SOAP header.

Example 24: *SOAP Binding for orderWidgets with a SOAP Header*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
    targetNamespace="http://widgetVendor.com/widgetOrderForm"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://widgetVendor.com/widgetOrderForm"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://widgetVendor.com/types/widgetTypes"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
```

Example 24: *SOAP Binding for orderWidgets with a SOAP Header*

```
<types>
  <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <element name="keyElem" type="xsd:base64Binary" />
  </schema>
</types>
<message name="widgetOrder">
  <part name="numOrdered" type="xsd:int" />
  <part name="keyVal" element="xsd1:keyElem" />
</message>
<message name="widgetOrderBill">
  <part name="price" type="xsd:float"/>
  <part name="keyVal" element="xsd1:keyElem" />
</message>
<message name="badSize">
  <part name="numInventory" type="xsd:int" />
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
    <fault message="tns:badSize" name="sizeFault"/>
  </operation>
</portType>
```

Example 24: *SOAP Binding for orderWidgets with a SOAP Header*

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="placeWidgetOrder">
      <soap:operation soapAction="" style="rpc"/>
      <input name="order">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"
                   parts="numOrdered" />
        <soap:header message="tns:widgetOrder" part="keyVal" />
      </input>
      <output name="bill">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"
                   parts="bill" />
        <soap:header message="tns:widgetOrderBill" part="keyVal" />
      </output>
      <fault name="sizeFault">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="http://widgetVendor.com/widgetOrderForm" use="encoded"/>
      </fault>
  </operation>
</binding>
...
</definitions>
```

Sending Data Using SOAP with Attachments

Overview

SOAP messages generally do not carry binary data. However, the W3C SOAP specification allows for using MIME multipart/related messages to send binary data in SOAP messages. This technique is called using SOAP with attachments. SOAP attachments are defined in the W3C's *SOAP Messages with Attachments Note* (http://www.w3.org/TR/SOAP-attachments).

Namespace

The WSDL extensions used to define the MIME multipart/related messages are defined in the namespace `http://schemas.xmlsoap.org/wsdl/mime/`.

In the discussion that follows, it is assumed that this namespace is prefixed with `mime`. The entry in the WSDL `<defintion>` element to set this up is shown in Example 25.

Example 25: *MIM Namespace Specification in a Contract*

```
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
```

Changing the message binding

In a default SOAP binding the first child element of the `<input>`, `<output>`, and `<fault>` elements is a `<soap:body>` element describing the body of the SOAP message representing the data. When using SOAP with attachments, the `<soap:body>` element is replaced with a `<mime:multipartRelated>` element.

Note: WSDL does not support using `<mime:multipartRelated>` for `<fault>` messages.

The `<mime:multipartReleated>` element tells Artix that the message body is going to be a multipart message that potentially contains binary data. The contents of the element define the parts of the message and their contents. `<mime:multipartReleated>` elements in Artix contain one or more `<mime:part>` elements that describe the individual parts of the message.

The first <mime:part> element must contain the <soap:body> element that would normally appear in a default SOAP binding. The remaining <mime:part> elements define the attachments that are being sent in the message.

Describing a MIME multipart message

MIME multipart messages are described using a <mime:multipartRelated> element that contains a number of <mime:part> elements. To fully describe a MIME multipart message in an Artix contract do the following:

1. Inside the <input> or <output> message you want to send as a MIME multipart message, add a <mime:mulipartRelated> element as the first child element of the enclosing message.
2. Add a <mime:part> child element to the <mime:multipartRelated> element and set its name attribute to a unique string.
3. Add a <soap:body> element as the child of the <mime:part> element and set its attributes appropriately.

   If the contract had a default SOAP binding, you can copy the <soap:body> element from the corresponding message from the default binding into the MIME multipart message.
4. Add another <mime:part> child element to the <mime:multipartReleated> element and set its name attribute to a unique string.
5. Add a <mime:content> child element to the <mime:part> element to describe the contents of this part of the message.

   To fully describe the contents of a MIME message part the <mime:content> element has the following attributes:

part	Specifies the name of the WSDL message part, from the parent message definition, that is used as the content of this part of the MIME multipart message being placed on the wire.

`type` — The MIME type of the data in this message part. MIME types are defined as a type and a subtype using the syntax *`type/subtype`*.

There are a number of predefined MIME types such as `image/jpeg` and `text/plain`. The MIME types are maintained by IANA and described in detail in *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies* (ftp://ftp.isi.edu/in-notes/rfc2045.txt) and *Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types* (ftp://ftp.isi.edu/in-notes/rfc2046.txt).

6. For each additional MIME part, repeat steps **4** and **5**.

Example

Example 26 shows an Artix contract for a service that stores X-rays in JPEG format. The image data, `xRay`, is stored as an `xsd:base64binary` and is packed into the MIME multipart message's second part, `imageData`. The remaining two parts of the input message, `patientName` and `patientNumber`, are sent in the first part of the MIME multipart image as part of the SOAP body.

Example 26: *Contract using SOAP with Attachments*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="XrayStorage"
    targetNamespace="http://mediStor.org/x-rays"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://mediStor.org/x-rays"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <message name="storRequest">
    <part name="patientName" type="xsd:string" />
    <part name="patientNumber" type="xsd:int" />
    <part name="xRay" type="xsd:base64Binary"/>
  </message>
  <message name="storResponse">
    <part name="success" type="xsd:boolean"/>
  </message>
```

Example 26: *Contract using SOAP with Attachments*

```
  <portType name="xRayStorage">
    <operation name="store">
      <input message="tns:storRequest" name="storRequest"/>
      <output message="tns:storResponse" name="storResponse"/>
    </operation>
  </portType>
  <binding name="xRayStorageBinding" type="tns:xRayStorage">
    <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="store">
      <soap:operation soapAction="" style="rpc"/>
      <input name="storRequest">
        <mime:multipartRelated>
          <mime:part name="bodyPart">
            <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://mediStor.org/x-rays" use="encoded"/>
          </mime:part>
          <mime:part name="imageData">
            <mime:content part="xRay" type="image/jpeg"/>
          </mime:part>
        </mime:multipartRelated>
      </input>
      <output name="storResponse">
        <soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                   namespace="urn:AttachmentService" use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="xRayStorageService">
    <port binding="tns:xRayStorageBinding" name="xRayStoragePort">
      <soap:address location="http://localhost:9000"/>
    </port>
  </service>
</definitions>
```

Adding a CORBA Binding

Overview

CORBA applications use a specific payload format when making and responding to requests. The CORBA binding, described using an IONA extension to WSDL, specifies the repository ID of the IDL interface represented by the port type, resolves parameter order and mode ambiguity in the operations' messages, and maps the XMLSchema data types to CORBA data types.

In addition to the binding information, Artix also uses a `<corba:typemap>` extension to unambiguously describe how data is mapped to CORBA data types. For primitive types, the mapping is straightforward. However, complex types such as structures, arrays, and exceptions require more detailed descriptions. For a detailed description of the CORBA type mappings see "CORBA Type Mapping" on page 441.

Options

To add a CORBA binding to an Artix contract you can choose one of two methods. The first option is to use the `wsdltocorba` command line tool. The command line tool automatically generates the binding and type map information for a specified port type.

The second option is to enter the binding and typemap information by hand using a text editor or XML editor. This option provides you the flexibility to customize the binding. However, hand editing Artix contracts can be a time consuming process and provides no error checking mechanisms. For information on the WSDL extensions used to specify a CORBA binding see "Mapping to the binding" on page 244.

Command line tool

The `wsdltocorba` tool adds CORBA binding information to an existing Artix contract. To generate a CORBA binding using `wsdltocorba` use the following command:

```
wsdltocorba -corba -i portType [-d dir][-b binding][-o file]
            [-n namespace] wsdl_file
```

The command has the following options:

`-corba`	Instructs the tool to generate a CORBA binding for the specified port type.
`-i` *`portType`*	Specifies the name of the port type being mapped to a CORBA binding.
`-d` *`dir`*	Specifies the directory into which the new WSDL file is written.
`-b` *`binding`*	Specifies the name for the generated CORBA binding. Defaults to *`portType`*`Binding`.
`-o` *`file`*	Specifies the name of the generated WSDL file. Defaults to *`wsdl_file`*`-corba.wsdl`.
`-n` *`namespace`*	Specifies the namespace to use for the generated CORBA typemap

The generated WSDL file will also contain a CORBA port with no address specified. To complete the port specification you can do so manually or use the Artix Designer.

WSDL Namespace

The WSDL extensions used to describe CORBA data mappings and CORBA transport details are defined in the WSDL namespace `http://schemas.iona.com/bindings/corba`. To use the CORBA extensions you will need to include the following in the `<definitions>` tag of your contract:

```
xmlns:corba="http://schemas.iona.com/bindings/corba"
```

Mapping to the binding

The extensions used to map a logical operation to a CORBA binding are described in detail below:

corba:binding indicates that the binding is a CORBA binding. This element has one required attribute: `repositoryID`. `repositoryID` specifies the full type ID of the interface. The type ID is embedded in the object's IOR and therefore must conform to the IDs that are generated from an IDL compiler. These are of the form:

```
IDL:module/interface:1.0
```

The corba:binding element also has an optional attribute, bases, that specifies that the interface being bound inherits from another interface. The value for bases is the type ID of the interface from which the bound interface inherits. For example, the following IDL:

```
//IDL
interface clash{};
interface bad : clash{};
```

would produce the following corba:binding:

```
<corba:binding repositoryID="IDL:bad:1.0"
               bases="IDL:clash:1.0"/>
```

corba:operation is an IONA-specific element of <operation> and describes the parts of the operation's messages. <corba:operation> takes a single attribute, name, which duplicates the name given in <operation>.

corba:param is a member of <corba:operation>. Each <part> of the input and output messages specified in the logical operation, except for the part representing the return value of the operation, must have a corresponding <corba:param>. The parameter order defined in the binding must match the order specified in the IDL definition of the operation. <corba:param> has the following required attributes:

mode	Specifies the direction of the parameter. The values directly correspond to the IDL directions: in, inout, out. Parameters set to in must be included in the input message of the logical operation. Parameters set to out must be included in the output message of the logical operation. Parameters set to inout must appear in both the input and output messages of the logical operation.
idltype	Specifies the IDL type of the parameter. The type names are prefaced with corba: for primitive IDL types, and corbatm: for complex data types, which are mapped out in the corba:typeMapping portion of the contract.
name	Specifies the name of the parameter as given in the logical portion of the contract.

corba:return is a member of `<corba:operation>` and specifies the return type, if any, of the operation. It only has two attributes:

`name`	Specifies the name of the parameter as given in the logical portion of the contract.
`idltype`	Specifies the IDL type of the parameter. The type names are prefaced with `corba:` for primitive IDL types and `corbatm:` for complex data types which are mapped out in the `corba:typeMapping` portion of the contract.

corba:raises is a member of `<corba:operation>` and describes any exceptions the operation can raise. The exceptions are defined as fault messages in the logical definition of the operation. Each fault message must have a corresponding `<corba:raises>` element. <corba:raises> has one required attribute, `exception`, which specifies the type of data returned in the exception.

In addition to operations specified in `<corba:operation>` tags, within the `<operation>` block, each `<operation>` in the binding must also specify empty `<input>` and `<output>` elements as required by the WSDL specification. The CORBA binding specification, however, does not use them.

For each fault message defined in the logical description of the operation, a corresponding `<fault>` element must be provided in the `<operation>`, as required by the WSDL specification. The `name` attribute of the `<fault>` element specifies the name of the schema type representing the data passed in the fault message.

Example

For example, a logical interface for a system to retrieve employee information might look similar to `personalInfoLookup`, shown in Example 27.

Example 27: *personalInfo lookup port type*

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

Example 27: *personalInfo lookup port type*

```
<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>
```

The CORBA binding for `personalInfoLookup` is shown in Example 28.

Example 28: *personalInfoLookup CORBA Binding*

```
<binding name="personalInfoLookupBinding" type="tns:personalInfoLookup">
  <corba:binding repositoryID="IDL:personalInfoLookup:1.0"/>
  <operation name="lookup">
    <corba:operation name="lookup">
      <corba:param name="empId" mode="in" idltype="corba:long"/>
      <corba:return name="return" idltype="corbatm:personalInfo"/>
      <corba:raises exception="corbatm:idNotFound"/>
    </corba:operation>
    <input/>
    <output/>
    <fault name="personalInfoLookup.idNotFound"/>
  </operation>
</binding>
```

Adding an FML Binding

Overview

FML buffers used by Tuxedo applications are described in one of two ways:

- A field table file that is loaded at run time.
- A C header file that is compiled into the application.

A field table file is a detailed and user readable text file describing the contents of a buffer. It clearly describes each field's name, id number, data type, and a comment. Using the FML library calls, Tuxedo applications map the field table description to usable `fldid`s at run time.

The C header file description of an FML buffer simply maps field names to their `fldid`. The `fldid` is an integer value that represents both the type of data stored in a field and a unique identifying number for that field.

Mapping from a field table to an Artix contract

Creating an Artix contract to represent an FML buffer is a two-step process. First, you must create the logical data representation of the FML buffer in the Artix contract as described in "Mapping to logical type descriptions" on page 248. Then, you must enter the FML binding information using Artix WSDL extensions as described in "Mapping to the physical FML binding" on page 250.

Mapping to logical type descriptions

To create a logical data type to represent data in an FML buffer do the following:

1. If the C header file for the FML buffer does not exist, generate it from the field table using the Tuxedo `mkfldhdr` or `mkfldhdr32` utility program.
2. In the `<types>` section of your Artix contract, create a `<complexType>` to represent the FML buffer.
3. Specify that all of the elements must be present and in the order specified by adding a `<sequence>` child element to the `<complexType>` element. See "Defining Data Structures" on page 209.
4. For each field in the FML buffer, create an `<element>` with the following attribute settings:
   - `name` is set to the name specified in the field table.

- `type` is set to the appropriate XMLSchema type for the type specified in the field table. See "XMLSchema Simple Types" on page 206.
- `maxOccurs` is set to `unbounded`.
- `minOccurs` is set to `0`.

Note: The elements of the `<complexType>` must be ordered in increasing order by the `fldid` specified in the C header.

For example, the `personalInfo` structure, defined in Example 27 on page 246, could be described by the field table file shown in Example 29.

Example 29: *personalInfo Field Table File*

```
# personalInfo Field Table
# name      number     type        flags     comment
name        100        string      -         Person's name
age         102        short       -         Person's age
hairColor   103        string      -         Person's hair color
```

The C++ header file generated by the Tuxedo `mkfldhdr` tool to represent the `personlInfo` FML buffer is shown in Example 30. Even if you are not planning to access the FML buffer using the compile time method, you will need to generate the header file when using Artix because this will give you the `fldid` values for the fields in the buffer.

Example 30: *personalInfo C++ header*

```
/*      fname        fldid            */
/*      -----        -----            */
#define name         ((FLDID)41060)   /* number: 100 type: string */
#define age          ((FLDID)102)     /* number: 102 type: short */
#define hairColor    ((FLDID)41063)   /* number: 103 type: string */
```

The order of the elements in the sequence used to logically describe the FML buffer are ordered in increasing order by `fldid` value. For the `personalInfo` FML buffer `age` must be listed first in the Artix contract

despite the fact that it is the second element listed in the field table. The corresponding logical description of the FML buffer in an Artix contract is shown in Example 31.

Example 31: *Logical description of personalInfo FML buffer*

```
<types>
  <schema targetNamespace="http://soapinterop.org/xsd"
          xmlns="http://www.w3.org/2001/XMLSchema"
          xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="personalInfoFML16">
      <sequence>
        <element name="age" type="xsd:short" minOccurs="0" maxOccurs="unbounded"/>
        <element name="name" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
        <element name="hairColor" type="xsd:string" minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </schema>
</types>
```

Mapping to the physical FML binding

To add the binding that maps the logical description of the FML buffer to a physical FML binding do the following:

1. Add the following line in the `<definition>` element at the beginning of the contract.

```
xmlns:fml="http://www.iona.com/bus/fml"
```

2. Create a new `<binding>` element in your contract to define the FML buffer's binding.
3. Add an `<fml:binding>` element to identify that this binding defines an FML buffer.

   The `<fml:binding>` element has two required attributes:

   - `style` specifies the encoding style used for the data. The valid encoding styles are `doc` and `rpc`.
   - `transport` sepcifies the transport this data will be sent over. This attribute can take the URI for any of the valid Artix transport definitions. You must be sure that the transport specified in the `<service>` element of the contract matched the transport specified here. See "Adding Transports" on page 297.

4. Add an <fml:idNameMapping> element to the binding to describe how the element names defined in the logical portion of the contract map to the fldid values for the corresponding fields in the FML buffer.

   The <fml:idNameMapping> has a mandatory type attribute. type can be either fml16 for specifying that the application uses FML16 buffers or fml32 for specifying that the application uses FML32 buffers.

5. For each element in the logical data type, add an <fml:element> element to the <fml:idNameMapping> element.

   <fml:element> defines how the logical data elements map to the physical FML buffer. It has two mandatory attributes:

   - fieldName specifies the name of the logical type describing the field.
   - fieldId specifies the fldid value for the field in the FML buffer.

Note: The field elements must be listed in increasing order of their fldid values.

6. For each operation in the interface, create a standard WSDL <operation> element to define the operation being bound.
7. For each operation, add a standard WSDL <input> and <output> elements to the <operation> element to define the messages used by the operation.
8. For each operation, add an <fml:operation> element to the <operation> element.

   <fml:operation> informs Artix that the operation's messages are to be packed into an FML buffer. <fml:operation> takes a single attribute, name, whose value must be identical to the name attribute of the <operation> element.

For example, the binding for the personalInfo FML buffer, defined in Example 29 on page 249, will be similar to the binding shown in Example 32.

Example 32: *personalInfo FML binding*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="personalInfoService" targetNamespace="http://info.org/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://soapinterop.org/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://soapinterop.org/xsd"
    xmlns:fml="http://www.iona.com/bus/fml">
...
  <message name="requestInfo">
    <part name="request" type="xsd1:personalInfoFML16"/>
  </message>
  <message name="infoReply">
    <part name="reply" type="xsd1:personalInfoFML16"/>
  </message>

  <portType name="personalInfoPort">
    <operation name="infoRequest">
      <input message="tns:requestInfo" name="requestInfo" />
      <output message="tns:infoReply" name="infoReply" />
    </operation>
  </portType>

  <binding name="personalInfoBinding" type="tns:personalInfoPort">
    <fml:binding style="rpc" transport="http://schemas.iona.com/transports/tuxedo"/>
    <fml:idNameMapping type="fml16">
      <fml:element fieldName="age" fieldId="102" />
      <fml:element fieldName="name" fieldId="41060" />
      <fml:element fieldName="hairColor" fieldId="41063" />
    </fml:idNameMapping>

    <operation name="infoRequest">
      <fml:operation name="infoRequest"/>
      <input name="requestInfo" />
      <output name="infoReply" />
    </operation>
  </binding>

...
</definitions>
```

Adding a Fixed Binding

Overview

The Artix fixed binding is used to represent fixed record length data. Common uses for this type of payload format are communicating with back-end services on mainframes and applications written in COBOL. Artix provides two means for creating a contract containing a fixed binding. If you are integrating with an application written in COBOL and have the COBOL copybook defining the data to be used, you can use the `coboltowsdl` tool documented in "Creating Contracts from COBOL Copybooks" on page 354.

Alternatively, if you do not have access to the COBOL copybook or have a logical interface you want to map to a fixed binding you can enter the binding information using any text editor or XML editor. To map a logical interface to a fixed binding do the following:

1. Add the proper namespace reference to the `<definition>` element of your contract. See "Fixed binding namespace" on page 254.
2. Add a standard WSDL `<binding>` element to your contract to hold the fixed binding, give the binding a unique `name`, and specify the port type that represents the interface being bound.
3. Add a <fixed:binding> element as a child of the new `<binding>` element to identify this as a fixed binding and set the element's attributes to properly configure the binding. See "<fixed:binding>" on page 254.
4. For each operation defined in the bound interface, add a standard WSDL `<operation>` element to hold the binding information for the operation's messages.
5. For each operation added to the binding, add a <fixed:operation> child element to the `<operation>` element. See "<fixed:operation>" on page 254.
6. For each operation added to the binding, add the `<input>`, `<output>`, and `<fault>` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the port type definition of the logical operation.

7. For each <input>, <output>, and <fault> element in the binding, add a <fixed:body> child element to define how the message parts are mapped into the concrete fixed record length payload. See "<fixed:body>" on page 255.

Fixed binding namespace

The IONA extensions used to describe fixed record length bindings are defined in the namespace http://schemas.iona.com/bindings/fixed. Artix tools use the prefix fixed to represent the fixed record length extensions. Add the following line to your contract:

```
xmlns:fixed="http://schemas.iona.com/bindings/fixed
```

<fixed:binding>

<fixed:binding> specifies that the binding is for fixed record length data. It has three optional attributes:

justification	Specifies the default justification of the data contained in the messages. Valid values are left and right. Default is left.
encoding	Specifies the codeset used to encode the text data. Valid values are any valid ISO locale or IANA codeset name. Default is UTF-8.
padHexCode	Specifies the hex value of the character used to pad the record.

The settings for the attributes on these elements become the default settings for all the messages being mapped to the current binding. All of the values can be overridden on a message-by-message basis.

<fixed:operation>

<fixed:operation> is a child element of the WSDL <operation> element and specifies that the operation's messages are being mapped to fixed record length data.

<fixed:operation> has one attribute, discriminator, that assigns a unique identifier to the operation. If your service only defines a single operation, you do not need to provide a discriminator. However, if your service has more than one service, you must define a unique discriminator for each operation in the service. Not doing so will result in unpredictable behavior when the service is deployed.

<fixed:body>

<fixed:body> is a child element of the <input>, <output>, and <fault> messages being mapped to fixed record length data. It specifies that the message body is mapped to fixed record length data on the wire and describes the exact mapping for the message's parts.

To fully describe how a message is mapped into the fixed message do the following:

1. If the default justification, padding, or encoding settings for the attribute are not correct for this particular message, override them by setting the optional attributes of <fixed:body>:

justification	Specifies how the data in the messages are justified. Valid values are left and right.
encoding	Specifies the codeset used to encode text data. Valid values are any valid ISO locale or IANA codeset name.
padHexCode	Specifies the hex value of the character used to pad the record.

2. For each part in the message the <fixed:body> element is binding, add the appropriate child element to define the part's concrete format on the wire.

   Three child elements are used in defining how logical data is mapped to a concrete fixed format message. These are:

<fixed:field>	Maps message parts defined using a simple type. See "XMLSchema Simple Types" on page 206.
<fixed:sequence>	Maps message parts defined using a sequence complex type. Complex types defined using <all> are not supported by the fixed format binding. See "Defining Data Structures" on page 209.
<fixed:choice>	Maps message parts defined using a choice complex type. See "Defining Data Structures" on page 209.

3. If you need to add any fields that are specific to the binding and that will not be passed to the applications, define them using a <fixed:field> element with its bindingOnly attribute set to true.

   When bindingOnly is set to true, the field described by the <fixed:field> element is not propagated beyond the binding. For input messages, this means that the field is read in and then

discarded. For output messages, you must also use the `fixedValue` attribute.

The order in which the message parts are listed in the `<fixed:body>` element represent the order in which they are placed on the wire. It does not need to correspond to the order in which they are specified in the `<message>` element defining the logical message.

<fixed:field>

`<fixed:field>` is used to map simple data types to a fixed length record. To define how the logical data is mapped to a fixed field do the following:

1. Create a `<fixed:field>` child element to the `<fixed:body>` element representing the message.
2. Set the `<fixed:field>` element's `name` attribute to the name of the message part defined in the logical message description that this element is mapping.
3. If the data being mapped is of type `xsd:string`, a simple type that has `xsd:string` as its base type, or an enumerated type set the `size` attribute of the `<fixed:field>` element.

   Note: If the message part is going to hold a date you can opt to use the `format` attribute described in step **4** instead of the `size` attribute.

   `size` specifies the length of the string record in the concrete fixed message. For example, the logical message part, `raverID`, described in Example 33 would be mapped to a `<fixed:field>` similar to Example 34.

Example 33: *Fixed String Message*

```
<message name="fixedStringMessage">
 <part name="raverID" type="xsd:string" />
</message>
```

In order to complete the mapping, you must know the length of the record field and supply it. In this case, the field, `raverID`, can contain no more than twenty characters.

Example 34: *Fixed String Mapping*

```
<fixed:field name="raverID" size="20" />
```

4. If the data being mapped is of a numerical type, like `xsd:int`, or a simple type that has a numerical type as its base type, set the `<fixed:field>` element's `format` attribute.

   `format` specifies how non-string data is formatted. For example, if a field contains a 2-digit numeric value with one decimal place, it would be described in the logical part of the contract as an `xsd:float`, as shown in Example 35.

Example 35: *Fixed Record Numeric Message*

```
<message name="fixedNumberMessage">
  <part name="rageLevel" type="xsd:float" />
</message>
```

From the logical description of the message, Artix has no way of determining that the value of `rageLevel` is a 2-digit number with one decimal place because the fixed record length binding treats all data as characters. When mapping `rageLevel` in the fixed binding you would specify its `format` with `##.#`, as shown in Example 36. This provides Artix with the meta-data needed to properly handle the data.

Example 36: *Mapping Numerical Data to a Fixed Binding*

```
<fixed:flield name="rageLevel" format="##.#" />
```

Dates are specified in a similar fashion. For example, the `format` of the date 12/02/72 is `MM/DD/YY`. When using the fixed binding it is recommended that dates are described in the logical part of the contract using `xsd:string`. For example, a message containing a date

would be described in the logical part of the contract as shown in Example 37.

Example 37: *Fixed Date Message*

```
<message name="fixedDateMessage">
  <part name="goDate" type="xsd:string" />
</message>
```

If `goDate` is entered using the standard short date format for US English locales, `mm/dd/yyyy`, you would map it to a fixed record field as shown in Example 38.

Example 38: *Fixed Format Date Mapping*

```
<fixed:field name="goDate" format="mm/dd/yyyy" />
```

5. If you want the message part to have a fixed value no matter what data is set in the message part by the application, set the `<fixed:field>` element's `fixedValue` attribute instead of the `size` or the `format` attribute.

   `fixedValue` specifies a static value to be passed on the wire. When used without `bindingOnly="true"`, the value specified by `fixedValue` replaces any data that is stored in the message part passed to the fixed record binding. For example, if `goDate`, shown in Example 37 on page 258, were mapped to the fixed field shown in Example 39, the actual message returned from the binding would always have the date 11/11/2112.

Example 39: *fixedValue Mapping*

```
<fixed:field name="goDate" fixedValue="11/11/2112" />
```

6. If the data being mapped is of an enumerated type, see "Defining Enumerated Types" on page 216, add a `<fixed:enumeration>` child element to the `<fixed:field>` element for each possible value of the enumerated type.

   `<fixed:enumeration>` takes two required attributes, `value` and `fixedValue`. `value` corresponds to the enumeration value as specified in the logical description of the enumerated type. `fixedValue` specifies

the concrete value that will be used to represent the logical value on the wire.

For example, if you had an enumerated type with the values `FruityTooty`, `Rainbow`, `BerryBomb`, and `OrangeTango` the logical description of the type would be similar to Example 40.

Example 40: *Ice Cream Enumeration*

```
<xs:simpleType name="flavorType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="FruityTooty"/>
    <xs:enumeration value="Rainbow"/>
    <xs:enumeration value="BerryBomb"/>
    <xs:enumeration value="OrangeTango"/>
  </xs:restriction>
</xs:simpleType>
```

When you map the enumerated type, you need to know the concrete representation for each of the enumerated values. The concrete representations can be identical to the logical or some other value. The enumerated type in Example 40 could be mapped to the fixed field shown in Example 41. Using this mapping Artix will write OT to the wire for this field if the enumerations value is set to `OrangeTango`.

Example 41: *Fixed Ice Cream Mapping*

```
<fixed:field name="flavor" size="2">
  <fixed:enumeration value="FruityTooty" fixedValue="FT" />
  <fixed:enumeration value="Rainbow" fixedValue="RB" />
  <fixed:enumeration value="BerryBomb" fixedValue="BB" />
  <fixed:enumeration value="OrangeTango" fixedValue="OT" />
</fixed:field>
```

Note that the parent `<fixed:field>` element uses the `size` attribute to specify that the concrete representation is two characters long. When mapping enumerations, the `size` attribute will always be used to represent the size of the concrete representation.

<fixed:choice>

<fixed:choice> is used to map choice complex types into fixed record length messages. To map a choice complex type to a <fixed:choice> do the following:

1. Add a <fixed:choice> child element to the <fixed:body> element.
2. Set the <fixed:choice> element's name attribute to the name of the logical message part being mapped.
3. Set the <fixed:choice> element's optional discriminatorName attribute to the name of the field used as the discriminator for the union.

   The value for discriminatorName corresponds to the name of a bindingOnly <fixed:field> that describes the type used for the union's discriminator as shown in Example 42. The only restriction in describing the discriminator is that it must be able to handle the values used to determine the case of the union. Therefore the values used in the union mapped in Example 42 must be two-digit integers.

Example 42: *Using discriminatorName*

```
<fixed:field name="disc" format="##" bindingOnly="true"/>
<fixed:choice name="unionStation" discriminatorName="disc">
...
</fixed:choice>
```

4. For each element in the logical definition of the message part, add a <fixed:case> child element to the <fixed:choice>.

<fixed:case>

<fixed:case> elements describe the complete mapping of a choice complex type element to a fixed record length message. To map a choice complex type element to a <fixed:case> do the following:

1. Set the <fixed:case> element's name attribute to the name of the logical definition's element.
2. Set the <fixed:case> element's fixedValue attribute to the value of the discriminator that selects this element. The value of fixedValue must correspond to the format specified by the discriminatorName attribute of the parent <fixed:choice> element.

3. Add a child element to define how the element's data is mapped into a fixed record.

   The child elements used to map the part's type to the fixed message are the same as the possible child elements of a `<fixed:body>` element. As with a `<fixed:body>` element, a `<fixed:sequence>` is made up of <fixed:field> elements to describe simple types, <fixed:choice> elements to describe choice complex types, and <fixed:sequence> elements to describe sequence complex types.

Example 43 shows an Artix contract fragment mapping a choice complex type to a fixed record length message.

Example 43: *Mapping a Union to a Fixed Record Length Message*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
   targetNamespace="http://www.iona.com/FixedService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:fixed="http://schemas.iona.com/bindings/fixed"
    xmlns:tns="http://www.iona.com/FixedService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/FixedService"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="unionStationType">
     <xsd:choice>
       <xsd:element name="train"  type="xsd:string"/>
       <xsd:element name="bus"    type="xsd:int"/>
       <xsd:element name="cab"    type="xsd:int"/>
       <xsd:element name="subway" type="xsd:string" />
     </xsd:choice>
   </xsd:complexType>
...
</types>
<message name="fixedSequence">
  <part name="stationPart" type="tns:unionStationType" />
</message>
<portType name="fixedSequencePortType">
...
</portType>
<binding name="fixedSequenceBinding"
         type="tns:fixedSequencePortType">
  <fixed:binding />
...
```

Example 43: *Mapping a Union to a Fixed Record Length Message*

```
    <fixed:field name="disc" format="##" bindingOnly="true" />
    <fixed:choice name="stationPart"
                  descriminatorName="disc">
      <fixed:case name="train" fixedValue="01">
        <fixed:field name="name" size="20" />
      </fixed:case>
      <fixed:case name="bus" fixedValue="02">
        <fixed:field name="number" format="###" />
      </fixed:case>
      <fixed:case name="cab" fixedValue="03">
        <fixed:field name="number" format="###" />
      </fixed:case>
      <fixed:case name="subway" fixedValue="04">
        <fixed:field name="name" format="10" />
      </fixed:case>
    </fixed:choice>
...
</binding>
...
</definition>
```

<fixed:sequence>

<fixed:sequence> maps sequence complext types to a fixed record length message. To map a sequence complex type to a <fixed:sequence> do the following:

1. Add a <fixed:sequnce> child element to the <fixed:body> element.
2. Set the <fixed:sequence> element's name attribute to the name of the logical message part being mapped.
3. For each element in the logical definition of the message part, add a child element to define the mapping for the part's type to the physical fixed message.

   The child elements used to map the part's type to the fixed message are the same as the possible child elements of a <fixed:body> element. As with a <fixed:body> element, a <fixed:sequence> is made up of <fixed:field> elements to describe simple types, <fixed:choice> elements to describe choice complex types, and <fixed:sequence> elements to describe sequence complex types.

4. If any elements of the logical data definition have occurrence constraints, see "Defining Data Structures" on page 209, map the element into a `<fixed:sequence>` element with its `occurs` and `counterName` attributes set.

   The `occurs` attribute specifies the number of times this sequence occurs in the message buffer. `counterName` specifies the name of the field used for specifying the number of sequence elements that are actually being sent in the message. The value of `counterName` corresponds to a binding only `<fixed:field>` with at least enough digits to count to the value specified in `occurs` as shown in Example 44. The value passed to the counter field can be any number up to the value specified by `occurs` and allows operations to use less than the specified number of sequence elements. Artix will pad out the sequence to the number of elements specified by occurs when the data is transmitted to the receiver so that the receiver will get the data in the promised fixed format.

Example 44: *Using counterName*

```
<fixed:field name="count" format="##" bindingOnly="true"/>
<fixed:sequence name="items" counterName="count" occurs="10">
...
</fixed:sequence>
```

For example, a structure containing a name, a date, and an ID number would contain three `<fixed:field>` elements to fully describe the mapping of the data to the fixed record message. Example 45 shows an Artix contract fragment for such a mapping.

Example 45: *Mapping a Sequence to a Fixed Record Length Message*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
   targetNamespace="http://www.iona.com/FixedService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:fixed="http://schemas.iona.com/bindings/fixed"
    xmlns:tns="http://www.iona.com/FixedService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

Example 45: *Mapping a Sequence to a Fixed Record Length Message*

```
<types>
  <schema targetNamespace="http://www.iona.com/FixedService"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="person">
     <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="date" type="xsd:string"/>
       <xsd:element name="ID" type="xsd:int"/>
     </xsd:sequence>
   </xsd:complexType>
...
</types>
<message name="fixedSequence">
  <part name="personPart" type="tns:person" />
</message>
<portType name="fixedSequencePortType">
...
</portType>
<binding name="fixedSequenceBinding"
         type="tns:fixedSequencePortType">
  <fixed:binding />
...
    <fixed:sequence name="personPart">
      <fixed:field name="name" size="20" />
      <fixed:field name="date" format="MM/DD/YY" />
      <fixed:field name="ID" format="#####" />
    </fixed:sequence>
...
</binding>
...
</definition>
```

Example

Example 46 shows an example of an Artix contract containing a fixed record length message binding.

Example 46: *Fixed Record Length Message Binding*

```
<?xml version="1.0" encoding="UTF-8"?>
```

Example 46: *Fixed Record Length Message Binding*

```
<definitions name="widgetOrderForm.wsdl"
 targetNamespace="http://widgetVendor.com/widgetOrderForm"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://widgetVendor.com/widgetOrderForm"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:fixed="http://schemas.iona.com/binings/fixed"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:simpleType name="widgetSize">
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="big"/>
          <xsd:enumeration value="large"/>
          <xsd:enumeration value="mungo"/>
          <xsd:enumeration value="gargantuan"/>
        </xsd:restriction>
      </xsd:simpleType>
      <xsd:complexType name="Address">
        <xsd:sequence>
          <xsd:element name="name" type="xsd:string"/>
          <xsd:element name="street1" type="xsd:string"/>
          <xsd:element name="street2" type="xsd:string"/>
          <xsd:element name="city" type="xsd:string"/>
          <xsd:element name="state" type="xsd:string"/>
          <xsd:element name="zipCode" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType name="widgetOrderInfo">
        <xsd:sequence>
          <xsd:element name="amount" type="xsd:int"/>
          <xsd:element name="order_date" type="xsd:string"/>
          <xsd:element name="type" type="xsd1:widgetSize"/>
          <xsd:element name="shippingAddress" type="xsd1:Address"/>
        </xsd:sequence>
      </xsd:complexType>
```

Example 46: *Fixed Record Length Message Binding*

```
    <xsd:complexType name="widgetOrderBillInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="amtDue" type="xsd:float"/>
        <xsd:element name="orderNumber" type="xsd:string"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
  </schema>
</types>
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
```

Example 46: *Fixed Record Length Message Binding*

```
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <fixed:binding/>
    <operation name="placeWidgetOrder">
      <fixed:operation discriminator="widgetDisc"/>
      <input name="widgetOrder">
        <fixed:body>
          <fixed:sequence name="widgetOrderForm">
            <fixed:field name="amount" format="###" />
            <fixed:field name="order_date" format="MM/DD/YYYY" />
            <fixed:field name="type" size="2">
              <fixed:enumeration value="big" fixedValue="bg" />
              <fixed:enumeration value="large" fixedValue="lg" />
              <fixed:enumeration value="mungo" fixedValue="mg" />
              <fixed:enumeration value="gargantuan" fixedValue="gg" />
            </fixed:field>
            <fixed:sequence name="shippingAddress">
              <fixed:field name="name" size="30" />
              <fixed:field name="street1" size="100" />
              <fixed:field name="street2" size="100" />
              <fixed:field name="city" size="20" />
              <fixed:field name="state" size="2" />
              <fixed:field name="zip" size="5" />
            </fixed:sequence>
          </fixed:sequence>
        </fixed:body>
      </input>
```

Example 46: *Fixed Record Length Message Binding*

```
        <output name="widgetOrderBill">
          <fixed:body>
            <fixed:sequence name="widgetOrderConformation">
              <fixed:field name="amount" format="###" />
              <fixed:field name="order_date" format="MM/DD/YYYY" />
              <fixed:field name="type" size="2">
                <fixed:enumeration value="big" fixedValue="bg" />
                <fixed:enumeration value="large" fixedValue="lg" />
                <fixed:enumeration value="mungo" fixedValue="mg" />
                <fixed:enumeration value="gargantuan" fixedValue="gg" />
              </fixed:field>
              <fixed:field name="amtDue" format="####.##" />
              <fixed:field name="orderNumber" size="20" />
              <fixed:sequence name="shippingAddress">
                <fixed:field name="name" size="30" />
                <fixed:field name="street1" size="100" />
                <fixed:field name="street2" size="100" />
                <fixed:field name="city" size="20" />
                <fixed:field name="state" size="2" />
                <fixed:field name="zip" size="5" />
              </fixed:sequence>
            </fixed:sequence>
          </fixed:body>
        </output>
    </operation>
  </binding>
  <service name="orderWidgetsService">
    <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
      <http:address location="http://localhost:8080"/>
    </port>
  </service>
</definitions>
```

Adding a Tagged Binding

Overview

The tagged data format supports applications that use self-describing, or delimited, messages to communicate. Artix can read tagged data and write it out in any supported data format. Similarly, Artix is capable of converting a message from any of its supported data formats into a self-describing or tagged data message.

To map a logical interface to a tagged data format do the following:

1. Add the proper namespace reference to the `<definition>` element of your contract. See "Tagged binding namespace" on page 270.
2. Add a standard WSDL `<binding>` element to your contract to hold the tagged binding, give the binding a unique `name`, and specify the port type that represents the interface being bound.
3. Add a <tagged:binding> element as a child of the new `<binding>` element to identify this as a tagged binding and set the element's attributes to properly configure the binding. See "<tagged:binding>" on page 270.
4. For each operation defined in the bound interface, add a standard WSDL `<operation>` element to hold the binding information for the operation's messages.
5. For each operation added to the binding, add a <tagged:operation> child element to the `<operation>` element. See "<tagged:operation>" on page 271.
6. For each operation added to the binding, add the `<input>`, `<output>`, and `<fault>` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the port type definition of the logical operation.
7. For each `<input>`, `<output>`, and `<fault>` element in the binding, add a <tagged:body> child element to define how the message parts are mapped into the concrete tagged data payload. See "<tagged:body>" on page 271.

Tagged binding namespace

The IONA extensions used to describe tagged data bindings are defined in the namespace `http://schemas.iona.com/bindings/tagged`. Artix tools use the prefix `tagged` to represent the tagged data extensions. Add the following line to the `<definitions>` element of your contract:

```
xmlns:tagged="http://schemas.iona.com/bindings/tagged"
```

<tagged:binding>

`<tagged:binding>` specifies that the binding is for tagged data format messages. It has ten attributes:

`selfDescribing`	Required attribute specifying if the message data on the wire includes the field names. Valid values are `true` or `false`. If this attribute is set to `false`, the setting for `fieldNameValueSeparator` is ignored.
`fieldSeparator`	Required attribute that specifies the delimiter the message uses to separate fields. Supported values are `newline`(\n), `comma`(,), `semicolon`(;), and `pipe`(|).
`fieldNameValueSeparator`	Specifies the delimiter used to separate field names from field values in self-describing messages. Supported vales are: `equals`(=), `tab`(\t), and `colon`(:).
`scopeType`	Specifies the scope identifier for complex messages. Supported values are `tab`(\t), `curlybrace`({*data*}), and `none`. The default is `tab`.
`flattened`	Specifies if data structures are flattened when they are put on the wire. If `selfDescribing` is `false`, then this attribute is automatically set to `true`.
`messageStart`	Specifies a special token at the start of a message. It is used when messages that require a special character at the start of a the data sequence. Currently the only supported value is `star`(*).
`messageEnd`	Specifies a special token at the end of a message. Supported values are `newline`(\n) and `percent`(%).

`unscopedArrayElement`	Specifies if array elements need to be scoped as children of the array. If set to `true` arrays take the form `echoArray{myArray=2;item=abc;item=def}`. If set to `false` arrays take the form `echoArray{myArray=2;{0=abc;1=def;}}`. Default is `false`.
`ignoreUnknownElements`	Specifies if Artix ignores undefined element in the message payload. Default is `false`.
`ignoreCase`	Specifies if Artix ignores the case with element names in the message payload. Default is `false`.

The settings for the attributes on these elements become the default settings for all the messages being mapped to the current binding.

<tagged:operation>

`<tagged:operation>` is a child element of the WSDL `<operation>` element and specifies that the operation's messages are being mapped to a tagged data format. It takes two optional attributes:

`discriminator`	Specifies a name to the operation for identifying the operation as it is sent down the wire by the Artix runtime.
`discriminatorStyle`	Specifies how the discriminator will identify data as it is sent down the wire by the Artix runtime. Supported values are `msgname`, `partlist`, and `fieldname`.

<tagged:body>

`<tagged:body>` is a child element of the `<input>`, `<output>`, and `<fault>` messages being mapped to a tagged data format. It specifies that the message body is mapped to tagged data on the wire and describes the exact mapping for the message's parts.

`<tagged:body>` will have one or more of the following child elements:

- <tagged:field>
- <tagged:sequence>
- <tagged:choice>

They describe the detailed mapping of the message to the tagged data to be sent on the wire.

<tagged:field>

<tagged:field> is used to map simple types and enumerations to a tagged data format. It has two attributes:

name	A required attribute that must correspond to the name of the logical message part that is being mapped to the tagged data field.
alias	An optional attribute specifying an alias for the field that can be used to identify it on the wire.

When describing enumerated types <tagged:field> will have a number of <tagged:enumeration> child elements.

<tagged:enumeration>

<tagged:enumeration> is a child element of <taggeded:field> and is used to map enumerated types to a tagged data format. It takes one required attribute, value, that corresponds to the enumeration value as specified in the logical description of the enumerated type.

For example, if you had an enumerated type, flavorType, with the values FruityTooty, Rainbow, BerryBomb, and OrangeTango the logical description of the type would be similar to Example 47.

Example 47: *Ice Cream Enumeration*

```
<xs:simpleType name="flavorType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="FruityTooty"/>
    <xs:enumeration value="Rainbow"/>
    <xs:enumeration value="BerryBomb"/>
    <xs:enumeration value="OrangeTango"/>
  </xs:restriction>
</xs:simpleType>
```

flavorType would be mapped to the tagged data format shown in Example 48.

Example 48: *Tagged Data Ice Cream Mapping*

```
<tagged:field name="flavor">
  <tagged:enumeration value="FruityTooty" />
  <tagged:enumeration value="Rainbow" />
  <tagged:enumeration value="BerryBomb" />
  <tagged:enumeration value="OrangeTango" />
</tagged:field>
```

<tagged:sequence>

<taggeded:sequence> maps arrays and sequences to a tagged data format. It has three attributes:

name	A required attribute that must correspond to the name of the logical message part that is being mapped to the tagged data sequence.
alias	An optional attribute specifying an alias for the sequence that can be used to identify it on the wire.
occurs	An optional attribute specifying the number of times the sequence appears. This attribute is used to map arrays.

A <tagged:sequence> can contain any number of <tagged:field>, <tagged:sequence>, or <tagged:choice> child elements to describe the data contained within the sequence being mapped. For example, a structure containing a name, a date, and an ID number would contain three <tagged:field> elements to fully describe the mapping of the data to the fixed record message. Example 49 shows an Artix contract fragment for such a mapping.

Example 49: *Mapping a Sequence to a Tagged Data Format*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="taggedDataMappingsample"
   targetNamespace="http://www.iona.com/taggedService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:fixed="http://schemas.iona.com/bindings/tagged"
    xmlns:tns="http://www.iona.com/taggedService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/taggedService"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="person">
     <xsd:sequence>
       <xsd:element name="name" type="xsd:string"/>
       <xsd:element name="date" type="xsd:string"/>
       <xsd:element name="ID" type="xsd:int"/>
     </xsd:sequence>
   </xsd:complexType>
...
</types>
```

Example 49: *Mapping a Sequence to a Tagged Data Format*

```
<message name="taggedSequence">
  <part name="personPart" type="tns:person" />
</message>
<portType name="taggedSequencePortType">
...
</portType>
<binding name="taggedSequenceBinding"
         type="tns:taggedSequencePortType">
  <tagged:binding selfDescribing="false" fieldSeparator="pipe"/>
...
    <tagged:sequence name="personPart">
      <tagged:field name="name"/>
      <tagged:field name="date" />
      <tagged:field name="ID" />
    </tagged:sequence>
...
</binding>
...
</definition>
```

<tagged:choice>

`<tagged:choice>` maps unions to a tagged data format. It takes three attributes:

`name`	A required attribute that must correspond to the name of the logical message `part` that is being mapped to the tagged data union.
`discriminatorName`	Specifies the message part used as the discriminator for the union.
`alias`	An optional attribute specifying an alias for the union that can be used to identify it on the wire.

A `<tagged:choice>` may contain one or more <tagged:case> child elements to map the cases for the union to a tagged data format.

<tagged:case>

`<tagged:case>` is a child element of `<tagged:choice>` and describes the complete mapping of a union's individual cases to a tagged data format. It takes one required attribute, `name`, that corresponds to the name of the case element in the union's logical description.

<tagged:case> must contain one child element to describe the mapping of the case's data to a tagged data format. Valid child elements are <tagged:field>, <tagged:sequence>, and <tagged:choice>. Example 50 shows an Artix contract fragment mapping a union to a tagged data format.

Example 50: *Mapping a Union to a Tagged Data Format*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="fixedMappingsample"
   targetNamespace="http://www.iona.com/tagService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:fixed="http://schemas.iona.com/bindings/tagged"
    xmlns:tns="http://www.iona.com/tagService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
  <schema targetNamespace="http://www.iona.com/tagService"
   xmlns="http://www.w3.org/2001/XMLSchema"
   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
   <xsd:complexType name="unionStationType">
     <xsd:choice>
       <xsd:element name="train"  type="xsd:string"/>
       <xsd:element name="bus"    type="xsd:int"/>
       <xsd:element name="cab"    type="xsd:int"/>
       <xsd:element name="subway" type="xsd:string" />
     </xsd:choice>
   </xsd:complexType>
...
</types>
<message name="tagUnion">
  <part name="stationPart" type="tns:unionStationType" />
</message>
<portType name="tagUnionPortType">
...
</portType>
<binding name="tagUnionBinding" type="tns:tagUnionPortType">
  <tagged:binding selfDescribing="false"
                  fieldSeparator="comma"/>
...
```

Example 50: *Mapping a Union to a Tagged Data Format*

```
    <tagged:choice name="stationPart" descriminatorName="disc">
      <tagged:case name="train">
        <tagged:field name="name" />
      </tagged:case>
      <tagged:case name="bus">
        <tagged:field name="number" />
      </tagged:case>
      <tagged:case name="cab">
        <tagged:field name="number" />
      </tagged:case>
      <tagged:case name="subway">
        <tagged:field name="name"/>
      </tagged:case>
    </tagged:choice>
...
</binding>
...
</definition>
```

Example

Example 51 shows an example of an Artix contract containing a tagged data format binding.

Example 51: *Tagged Data Format Binding*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="widgetOrderForm.wsdl"
 targetNamespace="http://widgetVendor.com/widgetOrderForm"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://widgetVendor.com/widgetOrderForm"
 xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
 xmlns:fixed="http://schames.iona.com/binings/tagged"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://widgetVendor.com/types/widgetTypes">
  <types>
    <schema targetNamespace="http://widgetVendor.com/types/widgetTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Example 51: *Tagged Data Format Binding*

```
    <xsd:simpleType name="widgetSize">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="big"/>
        <xsd:enumeration value="large"/>
        <xsd:enumeration value="mungo"/>
        <xsd:enumeration value="gargantuan"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Address">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="street1" type="xsd:string"/>
        <xsd:element name="street2" type="xsd:string"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="zipCode" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="widgetOrderBillInfo">
      <xsd:sequence>
        <xsd:element name="amount" type="xsd:int"/>
        <xsd:element name="order_date" type="xsd:string"/>
        <xsd:element name="type" type="xsd1:widgetSize"/>
        <xsd:element name="amtDue" type="xsd:float"/>
        <xsd:element name="orderNumber" type="xsd:string"/>
        <xsd:element name="shippingAddress" type="xsd1:Address"/>
      </xsd:sequence>
    </xsd:complexType>
  </schema>
</types>
<message name="widgetOrder">
  <part name="widgetOrderForm" type="xsd1:widgetOrderInfo"/>
</message>
<message name="widgetOrderBill">
  <part name="widgetOrderConformation" type="xsd1:widgetOrderBillInfo"/>
</message>
```

Example 51: *Tagged Data Format Binding*

```
<portType name="orderWidgets">
  <operation name="placeWidgetOrder">
    <input message="tns:widgetOrder" name="order"/>
    <output message="tns:widgetOrderBill" name="bill"/>
  </operation>
</portType>
<binding name="orderWidgetsBinding" type="tns:orderWidgets">
  <tagged:binding selfDescribing="false" fieldSeparator="pipe" />
    <operation name="placeWidgetOrder">
      <tagged:operation discriminator="widgetDisc"/>
      <input name="widgetOrder">
        <tagged:body>
          <tagged:sequence name="widgetOrderForm">
            <tagged:field name="amount" />
            <tagged:field name="order_date" />
            <tagged:field name="type" >
              <tagged:enumeration value="big" />
              <tagged:enumeration value="large" />
              <tagged:enumeration value="mungo" />
              <tagged:enumeration value="gargantuan" />
            </tagged:field>
            <tagged:sequence name="shippingAddress">
              <tagged:field name="name" />
              <tagged:field name="street1" />
              <tagged:field name="street2" />
              <tagged:field name="city" />
              <tagged:field name="state" />
              <tagged:field name="zip" />
            </tagged:sequence>
          </tagged:sequence>
        </tagged:body>
      </input>
```

Example 51: *Tagged Data Format Binding*

```
        <output name="widgetOrderBill">
          <tagged:body>
            <tagged:sequence name="widgetOrderConformation">
              <tagged:field name="amount" />
              <tagged:field name="order_date" />
              <tagged:field name="type">
                <tagged:enumeration value="big" />
                <tagged:enumeration value="large" />
                <tagged:enumeration value="mungo" />
                <tagged:enumeration value="gargantuan" />
              </tagged:field>
              <tagged:field name="amtDue" />
              <tagged:field name="orderNumber" />
              <tagged:sequence name="shippingAddress">
                <tagged:field name="name"/>
                <tagged:field name="street1"/>
                <tagged:field name="street2" />
                <tagged:field name="city" />
                <tagged:field name="state" />
                <tagged:field name="zip" />
              </tagged:sequence>
            </tagged:sequence>
          </tagged:body>
        </output>
    </operation>
  </binding>
  <service name="orderWidgetsService">
    <port name="widgetOrderPort" binding="tns:orderWidgetsBinding">
      <http:address location="http://localhost:8080"/>
    </port>
  </service>
</definitions>
```

Adding a TibMsg Binding

Overview

Artix supports the use of the TibrvMsg format when using the TIBCO Rendezvous transport.

Binding tags

To use this message format you need to define a binding between the interface you are exposing and the TibrvMsg format. The binding description is placed inside the standard `<binding>` tag and uses the tags listed in Table 5.

Table 5: *TibrvMsg Binding Attributes*

Attribute	Description
`tibrv:binding`	Specifies that the interface is exposed using TibrvMsgs.
`tibrv:binding@stringEncoding`	Specifies the charset used to encode `TIBRVMSG_STRING` data. Use IANA preferred MIME charset names (http://www.iana.org/assignments/character-sets). This parameter must be the same for both client and server.
`tibrv:operation`	Specifies that the operation is exposed using TibrvMsgs.
`tibrv:input`	Specifies that the input message is mapped to a TibrvMsg.
`tibrv:input@sortFields`	Specifies whether the server will sort the input message parts when they are unmarshalled.
`tibrv:input@messageNameFieldPath`	Specifies the field path that includes the input message name.
`tibrv:input@messageNameFieldValue`	Specifies the field value that corresponds to the input message name.
`tibrv:output`	Specifies that the output message is mapped to a TibrvMsg.
`tibrv:output@sortFields`	Specifies whether the client will sort the output message parts when they are unmarshalled.
`tibrv:output@messageNameFieldPath`	Specifies the field path that includes the output message name.

Table 5: *TibrvMsg Binding Attributes*

Attribute	Description
tibrv:output@messageNameFieldValue	Specifies the field value that corresponds to the output message name.

TIBRVMSG type mapping

Table 6 shows how TibrvMsg data types are mapped to XSD types in Artix contracts and C++ data types in Artix application code.

Table 6: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD
TIBRVMSG_STRING[1]	xsd:string
TIBRVMSG_BOOL	xsd:boolean
TIBRVMSG_I8	xsd:byte
TIBRVMSG_I16	xsd:short
TIBRVMSG_I32	xsd:int
TIBRVMSG_I64	xsd:long
TIBRVMSG_U8	xsd:unsignedByte
TIBRVMSG_U16	xsd:unsignedShort
TIBRVMSG_U32	xsd:unsignedInt
TIBRVMSG_U64	xsd:unsignedLong
TIBRVMSG_F32	xsd:float
TIBRVMSG_F64	xsd:double
TIBRVMSG_STRING	xsd:decimal
TIBRVMSG_DATETIME[2]	xsd:dateTime
TIBRVMSG_OPAQUE	xsd:base64Binary
TIBRVMSG_OPAQUE	xsd:hexBinary
TIBRVMSG_MSG[3]	xsd:complexType/sequence

Table 6: *TIBCO to XSD Type Mapping*

TIBRVMSG	XSD
TIBRVMSG_MSG[4]	xsd:complexType/all
TIBRVMSG_MSG[5]	xsd:complexType/choice
TIBRVMSG_*ARRAY/MSG[6]	xsd:complexType/sequence with element MaxOccurs > 1
TIBRVMSG_*ARRAY/MSG[6]	SOAP-ENC:Array[7]
TIBRVMSG_MSG[3]	SOAP-ENV:Fault[8]

1. TIB/RV does not provide any mechanism to indicate the encoding of strings in a TibrvMsg. The TIBCO plug-in port definition includes a property, `stringEncoding`, for specifying the string encoding. However, neither TIB/RV nor Artix look at this attribute; they merely pass the data along. It is up to the application developer to handle the encoding details if desired.
2. `TIBRVMSG_DATATIME` has microsecond precision. However, `xsd:dateTime` has only millisecond precision. Therefore, when using Artix sub-millisecond percision will be lost.
3. Sequences are mapped to nested messages where each element is a separate field. These fields are placed in the same order as they appear in the original sequence with field IDs beginning at 1. The fields are accessed by their field ID.
4. `Alls` are mapped to nested messages where each element is mapped to a separate field. The fields representing the elements of the all are given the same field name as element name and field IDs beginning from 1. They can be accessed by field name beginning from field ID 1. That means that the order of fields can be changed.
5. Choices are mapped to nested messages where each element is a separate field. Each field is enclosed with the same field name/type as element name/type of active member, and accessed by field name with field ID 1.
6. Arrays having `integer` or `float` elements are mapped to appropriate TIB/RV array types; otherwise they are mapped to nested messages.

7. SOAP RPC-encoded multi-dimensional arrays will be treated as one-dimensional: e.g. a 3x5 array will be serialized as a one-dimensional array having 15 elements. To keep dimensional information, use nested sequences with `maxOccurs > 1` instead.
8. When a server response message has a fault, it includes a field of type `TIBRVMSG_MSG` with the field name `fault` and field ID 1. This submessage has two fields of `TIBRVMSG_STRING`. One is named `faultcode` and has field ID 1, and the other is named `faultstring` and has field ID 2.

Adding a Pure XML Binding

Overview

The pure XML payload format provides an alternative to the SOAP binding by allowing services to exchange data using straight XML documents without the overhead of a SOAP envelope.

To bind an interface to a pure XML payload format do the following:

1. Add the namespace declaration to include the IONA extensions defining the XML binding. See "XML binding namespace" on page 285.
2. Add a standard WSDL `<binding>` element to your contract to hold the XML binding, give the binding a unique `name`, and specify the name of the WSDL `<portType>` element that represents the interface being bound.
3. Add an `<xformat:binding>` child element to the `<binding>` element to identify that the messages are being handled as pure XML documents without SOAP envelopes.
4. Optionally, set the `<xformat:binding>` element's `rootNode` attribute to a valid QName. For more information on the effect of the `rootNode` attribute see "XML messages on the wire" on page 285.
5. For each operation defined in the bound interface, add a standard WSDL `<operation>` element to hold the binding information for the operation's messages.
6. For each operation added to the binding, add the `<input>`, `<output>`, and `<fault>` children elements to represent the messages used by the operation. These elements correspond to the messages defined in the interface definition of the logical operation.

7. Optionally add an <xformat:body> element with a valid rootNode attribute to the added <input>, <output>, and <falut> elements to override the value of rootNode set at the binding level.

Note: If any of your messages have no parts, for example the output message for an operation that returns void, you must set the rootNode attribute for the message to ensure that the message written on the wire is a valid, but empty, XML document.

XML binding namespace

The IONA extensions used to describe XML format bindings are defined in the namespace http://schemas.iona.com/bindings/xmlformat. Artix tools use the prefix xformat to represent the XML binding extensions. Add the following line to your contracts:

```
xmlns:xformat="http://schemas.iona.com/bindings/xmlformat
```

XML messages on the wire

When you specify that an interface's message are to be passed as XML documents, without a SOAP envelope, you must take care to ensure that your messages form valid XML documents when they are written on the wire. You also need to ensure that non-Artix participants that receive the XML documents understand the messages generated by Artix.

A simple way to solve both problems is to use the optional rootNode attribute on either the global <xformat:binding> element or on the individual message's <xformat:body> elements. The rootNode attribute specifies the QName for the element that serves as the root node for the XML document generated by Artix. When the rootNode attribute is not set, Artix uses the root element of the message part as the root element when using doc style messages or an element using the message part name as the root element when using rpc style messages.

For example, without the `rootNode` attribute set the message defined in Example 52 would generate an XML document with the root element `<lineNumer>`.

Example 52: *Valid XML Binding Message*

```
<type ...>
  ...
  <element name="operatorID" type="xsd:int" />
  ...
</types>
<message name="operator">
 <part name="lineNumber" element="ns1:operatorID" />
</message>
```

For messages with one part, Artix will always generate a valid XML document even without the `rootNode` attribute set. However, the message in Example 53 would generate an invalid XML document.

Example 53: *Invalid XML Binding Message*

```
<types>
  ...
  <element name="pairName" type="xsd:string"/>
  <element name="entryNum" type="xsd:int"/>
  ...
</types>
<message name="matildas">
  <part name="dancing" element="ns1:pairName" />
  <part name="number" element="ns1:entryNum" />
</message>
```

Without the `rootNode` attribute specified in the XML binding, Artix will generate an XML document similar to Example 54 for the message defined in Example 53. The Artix generated XML document is invalid because it has two root elements: `<pairName>` and `<entryNum>`.

Example 54: *Invalid XML Document*

```
<pairName>
  Fred&Linda
</pairName>
<entryNum>
  123
</entryNum>
```

If you set the `rootNode` attribute, as shown in Example 55 Artix will wrap the elements in the specified root element. In this example, the `rootNode` attribute is defined for the entire binding and specifies that the root element will be named entrants.

Example 55: *XML Format Binding with rootNode set*

```
<portType name="danceParty">
  <operation name="register">
    <input message="tns:matildas" name="contestant"/>
    <output message="tns:space" name="entered"/>
  </operation>
</portType>
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant" />
    <output name="entered" />
  </operation>
</binding>
```

A an XML document generated from the input message would be similar to Example 56. Notice that the XML document now only has one root element.

Example 56: *XML Document generated using the rootNode attribute*

```
<entrants>
  <pairName>
    Fred&Linda
  </pairName>
  <entryNum>
    123
  </entryNum>
</entrants>
```

Overriding the binding's rootNode attribute setting

You can also set the rootNode attribute for each individual message, or override the global setting for a particular message by using the `<xformat:body>` element inside of the message binding. For example, if you wanted the output message defined in Example 55 to have a different root element from the input message, you could override the binding's root element as shown in Example 57.

Example 57:

```
<binding name="matildaXMLBinding" type="tns:dancingMatildas">
  <xmlformat:binding rootNode="entrants"/>
  <operation name="register">
    <input name="contestant" />
    <output name="entered" />
      <xformat:body rootNode="entryStatus"/>
  </operation>
</binding>
```

Adding a G2++ Binding

Overview

G2++ is a set of mechanisms for defining and manipulating hierarchically structured messages. G2++ messages can be thought of as records, which are described in terms of their structure and the data types they contain.

G2++ is an alternative to "raw" structures (such as C or C++ structs), which rely on common data representation characteristics that may not be present in a heterogeneous distributed system.

Simple G2++ mapping example

Consider the following instance of a G2++ message:

Note: Because tabs are significant in G2++ files (that is, tabs indicate scoping levels and are not simply treated as "white space"), examples in this chapter indicate tab characters as an up arrow (caret) followed by seven spaces.

Example 58: *ERecord G2++ Message*

```
ERecord
^       XYZ_Part
^       ^       XYZ_Code^       someValue1
^       ^       password^       someValue2
^       ^       serviceFieldName^       someValue3
^       newPart
^       ^       newActionCode^       someValue4
^       ^       newServiceClassName^       someValue5
^       ^       oldServiceClassName^       someValue6
```

This G2++ message can be mapped to the following logical description, expressed in WSDL:

Example 59: *WSDL Logical Description of ERecord Message*

```
<types>
    <schema targetNamespace="http://soapinterop.org/xsd"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Example 59: *WSDL Logical Description of ERecord Message*

```
<complexType name="XYZ_Part">
  <all>
    <element name="XYZ_Code" type="xsd:string"/>
    <element name="password" type="xsd:string"/>
    <element name="serviceFieldName" type="xsd:string"/>
  </all>
</complexType>
<complexType name="newPart">
  <all>
    <element name="newActionCode" type="xsd:string"/>
    <element name="newServiceClassName" type="xsd:string"/>
    <element name="oldServiceClassName" type="xsd:string"/>
  </all>
<complexType name="PRequest">
  <all>
    <element name="newPart" type="xsd1:newPart"/>
    <element name="XYZ_Part" type="xsd1:XYZ_Part"/>
  </all>
</complexType>
```

Note that each of the message sub-structures (`newPart` and `XYZ_Part`) are initially described separately in terms of their elements, then the two sub-structure are aggregated together to form the enclosing record (`PRequest`).

This logical description is mapped to a physical representation of the G2++ message, also expressed in WSDL:

Example 60: *WSDL Physical Representation of ERecord Message*

```
<binding name="ERecordBinding" type="tns:ERecordRequestPortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <artix:binding transport="tuxedo" format="g2++">
    <G2Definitions>
      <G2MessageDescription name="creation" type="msg">
        <G2MessageComponent name="ERecord" type="struct">
          <G2MessageComponent name="XYZ_Part" type="struct">
            <element name="XYZ_Code" type="element"/>
            <element name="password" type="element"/>
            <element name="serviceFieldName" type="element"/>
          </G2MessageComponent>
          <G2MessageComponent name="newPart" type="struct">
            <element name="newActionCode" type="element"/>
            <element name="newServiceClassName" type="element"/>
            <element name="oldServiceClassName" type="element"/>
          </G2MessageComponent>
        </G2MessageComponent>
      </G2MessageDescription>
    </G2Definitions>
</artix:binding>
```

Note that all G2++ definitions are contained within the scope of the `<G2Definitions> </G2Definitions>` tags. Each of the messages are defined with the scope of a `<G2MessageDescription> </G2MessageDescription>` construct. The type attribute for message descriptions must be `"msg"` while the name attribute simply has to be unique.

Each record is described within the scope of a `<G2MessageComponent> </G2MessageComponent>` construct. Within this, the `name` attribute must reflect the G2++ record name and the `type` attribute must be `"struct"`.

Nested within the records are the element definitions, however if required a record could be nested here by inclusion of a nested `<G2MessageComponent>` scope (`newPart` and `XYZ_Part` are nested records of parent `ERecord`). Element “name” attributes must match the G2 element name. Defining a record and then referencing it as a nested struct of a parent is legal for the logical mapping but not the physical. In the physical mapping, nested structs must be defined in-place.

The following example illustrates the custom mapping of arrays, which differs from strictly defined G2++ array mappings. The array definition is shown below:

```
IMS_MetaData^       2
^       0
^       ^       columnName^         SERVICENAME
^       ^       columnValue^       someValue1
^       1
^       ^       columnName^         SERVICEACTION
^       ^       columnValue^       someValue2
```

This represents an array with two elements. When placed in a G2++ message, the result is as follows:

Example 61: *Extended ERecord G2++ Message*

```
ERecord
^       XYZ_Part
^       ^       XYZ_Code^       someValue1
^       ^       password^       someValue2
^       ^       serviceFieldName^       someValue3
^       XYZ_Metadata^       1
^       ^       0
^       ^       ^       columnName^       pushToTalk
^       ^       ^       columnValue^       PT01
^       newPart
^       ^       newActionCode^       someValue4
^       ^       newServiceClassName^       someValue5
^       ^       oldServiceClassName^       someValue6
```

In this version of the ERecord record, `XYZ_Part` contains an array called `XYZ_MetaData`, whose size is one. The single entry can be thought of as a name/value pair: `pushToTalk/PT01`, which allows us to ignore `columnName` and `columnValue`.

Mapping the new ERecord record to a WSDL logical description results in the following:

Example 62: *WSDL Logical Description of Extended ERecord Message*

```
<types>
    <schema targetNamespace="http://soapinterop.org/xsd"
      xmlns="http://www.w3.org/2001/XMLSchema"
      xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

      <complexType name="XYZ_Part">
        <all>
          <element name="XYZ_Code" type="xsd:string"/>
          <element name="password" type="xsd:string"/>
          <element name="serviceFieldName" type="xsd:string"/>
           <element name="pushToTalk" type="xsd:string"/>
        </all>
      </complexType>

      <complexType name="newPart">
        <all>
          <element name="newActionCode" type="xsd:string"/>
          <element name="newServiceClassName" type="xsd:string"/>
          <element name="oldServiceClassName" type="xsd:string"/>
        </all>

      <complexType name="PRequest">
        <all>
          <element name="newPart" type="xsd1:newPart"/>
          <element name="XYZ_Part" type="xsd1:XYZ_Part"/>
        </all>
      </complexType>
```

Thus the array elements `columnName` and `columnValue` are "promoted" to a name/Value pair in the logical mapping. This physical G2++ representation can now be mapped as follows:

Example 63: *WSDL Physical Representation of Extended ERecord Message*

```
<binding name="ERecordBinding" type="tns:ERecordRequestPortType">
  <soap:binding style="rpc"
transport="http://schemas.xmlsoap.org/soap/http"/>
  <artix:binding transport="tuxedo" format="g2++">
    <G2Definitions>
      <G2MessageDescription name="creating" type="msg">
        <G2MessageComponent name="ERecord" type="struct">
          <G2MessageComponent name="XYZ_Part" type="struct">
            <element name="XYZ_Code" type="element"/>
            <element name="password" type="element"/>
            <element name="serviceFieldName" type="element"/>
             <G2MessageComponent name="XYZ_MetaData" type="array" size="1">
              <element name="pushToTalk" type="element"/>
             </G2MessageComponent>
          </G2MessageComponent>
          <G2MessageComponent name="newPart" type="struct">
            <element name="newActionCode" type="element"/>
            <element name="newServiceClassName" type="element"/>
            <element name="oldServiceClassName" type="element"/>
          </G2MessageComponent>
        </G2MessageComponent>
      </G2MessageDescription>
    </G2Definitions>
</artix:binding>
```

This physical mapping of the extended ERecord message now contains an array, described with its `XYZ_MetaData` name (as per the G2++ record definition). Its `type` is `"array"` and its size is one. This `G2MessageComponent` contains a single element called `"pushToTalk"`.

Ignoring unknown elements

It is possible to create a `G2Definitions` scope that begins with a G2-specific configuration scope. This configuration scope is called `G2Config` in the following example:

```
<G2Definitions>
^       <G2Config>
^       ^       <IgnoreUnknownElements value="true"/>
</G2Config>
   .
   .
   .
```

In this scope, the only variable used is `IgnoreUnknownElements`, which can have a value of “true” or “false”. If the value is set to true, elements or array elements that are not defined in the G2 message definitions will be ignored. For example the following record would be valid if `IgnoreUnknownElements` is set to true.

Example 64: *Valid G2++ Record With Ignored Fields*

```
ERecord
^       XYZ_Part
^       XYZ_Code^       someValue1
^       AnElement^       foo
^       password^       someValue2
^       serviceFieldName^       someValue3
^       XYZ_MetaData^       2
^       ^       0
^       ^       ^       columnName^       pushToTalk
^       ^       ^       columnValue^       PT01
^       ^       1
^       ^       ^       columnName^       AnArrayElement
^       ^       ^       columnValue^       bar
^       newPart
^       ^       newActionCode^       someValue4
^       ^       newServiceClassName^       someValue5
^       ^       oldServiceClassName^       someValue6
```

When parsed, the above ERecord would not include the elements `"AnElement"` or `"AnArrayElement"`. If `IgnoreUnknownElements` is set to false, the above record would be rejected as invalid.

CHAPTER 14

Adding Transports

To fully define a service you need to add a transport.

Overview

The final piece of information needed to describe a service are the transport details defining how it connects to a network. 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.

In this chapter

This chapter discusses the following topics:

Defining a Service

Overview

All of the transport details for an endpoint are defined in `<service>` elements. A `<service>` element defines a collection of `<port>` elements. The `<port>` elements defines the relationship between a particular `<binding>` element and the transport on which the messages are to be sent. The `<port>` element contains all of the information defining the endpoints connection to a network including what type of transport to use, the address, and any other transport details.

The <service> element

The `<service>` element contains a group of one or more ports that have some relationship. How the ports are related is up to you. For example you could build a contract where every port is contained in its own `<service>`, or you could decide to group all of the ports that are bound to a particular interface into `<service>` elements.

A `<service>` element has one required attribute, `name`, that identifies the service. The identifier must be unique among all of the services defined in the contract. Example 65 shows an example of a service named `riotService`.

Example 65: *Sample Service*

```
<service name="riotService">
  <port ....>
    ...
  </port>
</service>
```

The <port> element

The `<port>` element defines how a binding is tied to a specific network transport. You specify the binding from which messages will be sent over the network using the `<port>` element's `binding` attribute. The value of the binding attribute must correspond to a binding defined with in the same contract, or a contract imported into the same contract, in which the port is defined.

The port element also has an attribute, `name`, that identifies the port. The identifier must be unique among the ports describe within the containing `<service>` element. shows a `<port>` element, `riotPort`, that defines a port bound to `riotBinding`.

Example 66: *Sample Port*

```
<service name="riotService">
  <port name="riotPort" binding="riotBinding">
    ...
  </port>
</service>
```

Contained within the `<port>` element are the elements used to define the details of the transport that is used to send messages. In a standard WSDL contract the transport details would be represented using a `<soap:address>` element. However, Artix provides a number of transports and the elements to define them. The following sections describe the details of adding the details for these transports.

Creating an HTTP Service

Overview

HTTP is the standard TCP/IP-based protocol used for client-server communications on the World Wide Web. The main function of HTTP is to establish a connection between a web browser (client) and a web server for the purposes of exchanging files and possibly other information on the Web.

In addition to the standard `<soap:address>` element, Artix provides a number of proprietary HTTP extensions. The Artix extensions allow you to specify a number of the HTTP port's configuration in the contract.

In this section

This section discusses the following topics:

Specifying the Service Address

Overview

Artix provides two ways of specifying an HTTP service's address depending on the payload format you are using. SOAP has a standardized `<soap:address>` element. All other payload formats use Artix's `<http:address>` element.

Using <soap:address>

When you are sending SOAP over HTTP you must use the `<soap:address>` element to specify the service's address. It has one attribute, location, that specifies the service's address as a URL. Example 67 shows a port used to send SOAP over HTTP.

Example 67: *SOAP Port*

```
<service name="artieSOAPService">
    <port binding="artieSOAPBinding" name="artieSOAPPort">
        <soap:address location="http://artie.com/index.xml">
    </port>
</service>
```

Using <http:address>

When your messages are formatted using any other payload format than SOAP, such as fixed, you must use Artix's `<http:address>` element to specify the service's address.

Example 68 shows the namespace entries you need to add to the `<definitions>` element of your contract to use the HTTP extensions.

Example 68: *Artix HTTP Extension Namespaces*

```
<definitions
  ...
  xmlns:http="http://schemas.iona.com/transports/http"
  ... >
```

The `<http:address>` element is similar to the `<soap:address>` element. It has one attribute, `location`, that specifies the service's address as a URL. Example 69 shows a port used to send fixed data over HTTP.

Example 69: *Generic HTTP Port*

```
<service name="artieFixedService">
    <port binding="artieFixedBinding" name="artieFixedPort">
        <http:address location="http://artie.com/index.xml">
    </port>
</service>
```

Configuring HTTP Transport Attributes

Overview

To allow you more flexibility in configuring an HTTP port, Artix has its own set of WSDL extensions that can be used to define an HTTP port. All of the configuration elements are optional. An HTTP port is fully defined by the address element.

Example 70 shows the namespace entries you need to add to the `<definitions>` element of your contract to use the HTTP extensions.

Example 70: *Artix HTTP Extension Namespaces*

```
<definitions
  ...
  xmlns:http-conf="http://schemas.iona.com/transports/http/configuration"
  ... >
```

Because HTTP client ports and HTTP server ports have slightly different configuration options, Artix uses two elements to configure an HTTP port. `<http-conf:client>` defines a client port. `<http-conf:server>` defines a server port.

HTTP client configuration

Table 7 describes the client-side configuration attributes for the HTTP transport that are defined within the `http-conf:client` element.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`SendTimeout`	This specifies the length of time, in milliseconds, that the client can continue to try to send a request to the server before the connection is timed out. The default is `30000`.
`ReceiveTimeout`	This specifies the length of time, in milliseconds, that the client can continue to try to receive a response from the server before the connection is timed out. The default is `30000`.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`AutoRedirect`	This specifies whether a client request should be automatically redirected on behalf of the client when the server issues a redirection reply via the `RedirectURL` server-side configuration attribute. Valid values are `true` and `false`. The default is `false`, to let the client redirect the request itself.
`UserName`	Some servers require that client users can be authenticated. In the case of basic authentication, the server requires the client user to supply a username and password. This specifies the user name that is to be used for authentication. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Password`	Some servers require that client users can be authenticated. In the case of basic authentication, the server requires the client user to supply a username and password. This specifies the password that is to be used for authentication. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`AuthorizationType`	This specifies the name of the authorization scheme in use. This name is specified and handled at application level. Artix does not perform any validation on this value. It is the user's responsibility to ensure that the correct scheme name is specified, as appropriate. **Note:** If basic username and password-based authentication is being used, this does not need to be set. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Authorization`	This specifies the authorization credentials used to perform the authorization. These are encoded and handled at application-level. Artix does not perform any validation on the specified value. It is the user's responsibility to ensure that the correct authorization credentials are specified, as appropriate. **Note:** If basic username and password-based authentication is being used, this does not need to be set. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`Accept`	This specifies what media types the client is prepared to handle. These are also known as multipurpose internet mail extensions (MIME) types. MIME types are regulated by the Internet Assigned Numbers Authority (IANA). See `http://www.iana.org/assignments/media-types/` for more details. Specified values consist of a main type and sub-type, separated by a forward slash. For example, a main type of `text` might be qualified as follows: `text/html` or `text/xml`. Similarly, a main type of image might be qualified as follows: `image/gif` or `image/jpeg`. An asterisk (that is, *) can be used as a wildcard to specify a group of related types. For example, if you specify `image/*`, this means that the client can accept any image, regardless of whether it is a GIF or a JPEG, and so on. A value of `*/*` indicates that the client is prepared to handle any type. Examples of typical types that might be set are `text/xml`, `text/html`, `text/text`, `image/gif`, `image/jpeg`, `application/jpeg`, `application/msword`, `application/xbitmap`, `audio/au`, `audio/wav`, `video/avi`, `video/mpeg`. A full list of MIME types is available at `http://www.iana.org/assignments/media-types/`. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`AcceptLanguage`	This specifies what language (for example, American English) the client prefers for the purposes of receiving a response. Language tags are regulated by the International Organization for Standards (ISO) and are typically formed by combining a language code (determined by the ISO-639 standard) and country code (determined by the ISO-3166 standard) separated by a hyphen. For example, `en-US` represents American English. A full list of language codes is available at `http://www.w3.org/WAI/ER/IG/ert/iso639.htm`. A full list of country codes is available at `http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html`. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`AcceptEncoding`	This specifies what content codings the client is prepared to handle. The primary use of content codings is to allow documents to be compressed using some encoding mechanism, such as zip or gzip. Content codings are regulated by the Internet Assigned Numbers Authority (IANA). See `http://www.w3.org/Protocols/rfc2616/rfc2616-sec3.html` for more details of content codings. Possible content coding values include `zip`, `gzip`, `compress`, `deflate`, and `identity`. Artix performs no validation on content codings. It is the user's responsibility to ensure that a specified content coding is supported at application level. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`ContentType`	This is relevant if the client request specifies the `POST` method, to send data to the server for processing. This specifies the media type of the data being sent in the body of the client request. For web services, this should be set to `text/xml`. If the client is sending HTML form data to a CGI script, this should be set to `application/x-www-form-urlencoded`. If the HTTP `POST` request is bound to a fixed payload format (as opposed to SOAP), the content type is typically set to `application/octet-stream`. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Host`	This specifies the internet host and port number of the resource on which the client request is being invoked. This is sent by default based upon the URL specified in the `URL` attribute. It indicates what host the client prefers for clusters (that is, for virtual servers mapping to the same internet protocol (IP) address). **Note:** Certain DNS scenarios or application designs might request you to set this, but it is not typically required. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`Connection`	This specifies whether a particular connection is to be kept open or closed after each request/response dialog. Valid values are `close` and `Keep-Alive`. The default is `close`, to close the connection to the server after each request/response dialog. If `Keep-Alive` is specified, and the server honors it, the connection is reused for subsequent request/response dialogs. **Note:** The server can choose to not honor a request to keep the connection open, and many servers and proxies (caches) do not honor such requests. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`ConnectionAttempts`	This specifies the number of times a client will transparently attempt to connect to server.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`CacheControl`	This specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a request from a client to a server. Valid values are: • `no-cache` prevents a cache from using a particular response to satisfy subsequent client requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response. • `no-store` indicates that a cache must not store any part of a response or any part of the request that evoked it. • `max-age` indicates that the client can accept a response whose age is no greater than the specified time in seconds. • `max-stale` indicates that the client can accept a response that has exceeded its expiration time. If a value is assigned to `max-stale`, it represents the number of seconds beyond the expiration time of a response up to which the client can still accept that response. If no value is assigned, it means the client can accept a stale response of any age. • `min-fresh` indicates that the client wants a response that will be still be fresh for at least the specified number of seconds indicated by the value set for min-fresh. • `no-transform` indicates that a cache must not modify media type or location of the content in a response between a server and a client. • `only-if-cached` indicates that a cache should return only responses that are currently stored in the cache, and not responses that need to be reloaded or revalidated.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
	• `cache-extension` indicates additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Cookie`	This specifies a static cookie to be sent to the server. Some session designs that maintain state use cookies to identify sessions. **Note:** If the cookie is dynamic, it must be set by the server when the server is first accessed, and is then handled automatically by the application runtime. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`BrowserType`	This specifies information about the browser from which the client request originates. In the standard HTTP specification from the World Wide Web consortium (W3C) this is also known as the *user-agent*. Some servers optimize based upon the client that is sending the request. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.
`Referer`	If a client request is as a result of the browser user clicking on a hyperlink rather than typing a URL, this specifies the URL of the resource that provided the hyperlink. This is sent automatically if `AutoRedirect` is set to `true`. This can allow the server to optimize processing based upon previous task flow, and to generate lists of back-links to resources for the purposes of logging, optimized caching, tracing of obsolete or mistyped links, and so on. However, it is typically not used in web services applications. If this is set, it is sent as a transport attribute in the header of a request message from the client to the server.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`ProxyServer`	This specifies the URL of the proxy server, if one exists along the message path. A proxy can receive client requests, possibly modify the request in some way, and then forward the request along the chain possibly to the target server. A proxy can act as a special kind of security firewall. **Note:** Artix does not support the existence of more than one proxy server along the message path.
`ProxyUserName`	This is only relevant if a proxy server exists along the message path. This specifies the username to use for authentication on the proxy server if it requires separate authorization. **Note:** Artix does not perform any validation on user names specified. It is the user's responsibility to ensure that user names are correct.
`ProxyPassword`	This is only relevant if a proxy server exists along the message path. This specifies the password to use for authentication on the proxy server if it requires separate authorization. **Note:** Artix does not perform any validation on passwords specified. It is the user's responsibility to ensure that passwords are correct.
`ProxyAuthorizationType`	This is only relevant if a proxy server exists along the message path. If basic username and password-based authentication is not in use by the proxy server, this specifies the type of authentication that is in use. This specifies the name of the authorization scheme in use. This name is specified and handled at application level. Artix does not perform any validation on this value. It is the user's responsibility to ensure that the correct scheme name is specified, as appropriate. **Note:** If basic username and password-based authentication is being used by the proxy server, this does not need to be set.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`ProxyAuthorization`	This is only relevant if proxy servers are in use along the request-response chain. If basic username and password-based authentication is not in used by the proxy server, this specifies the actual data that the proxy server should use to authenticate the client. This specifies the authorization credentials used to perform the authorization. These are encoded and handled at application-level. Artix does not perform any validation on the specified value. It is the user's responsibility to ensure that the correct authorization credentials are specified, as appropriate. **Note:** If basic username and password-based authentication is being used by the proxy server, this does not need to be set.
`UseSecureSockets`	This indicates whether the client wants to open a secure connection. A secure HTTP connection is commonly referred to as HTTPS. Valid values are `true` and `false`. The default is `false`, to indicate that the client does not want to open a secure connection. **Note:** If the `http-conf:client URL` attribute has a value with a prefix of `https://`, a secure HTTP connection is automatically enabled, even if `UseSecureSockets` is not set to `true`.
`ClientCertificate`	This is only relevant if `UseSecureSockets` is set to `true`. This specifies the full path to the PEM-encoded X509 certificate issued by the certificate authority for the client.
`ClientCertificateChain`	This is only relevant if `UseSecureSockets` is set to `true`. This specifies the full path to the file that contains all the certificates in the chain.
`ClientPrivateKey`	This is only relevant if `UseSecureSockets` is set to `true`. This is used in conjunction with `ClientCertificate`. It specifies the full path to the PEM-encoded private key that corresponds to the X509 certificate specified by `ClientCertificate`. This is required only if `ClientCertificate` has been specified.

Table 7: *HTTP Client Configuration Attributes*

Configuration Attribute	Explanation
`ClientPrivateKeyPassword`	This is only relevant if `UseSecureSockets` is set to `true`. This specifies a password that is used to decrypt the PEM-encoded private key, if it has been encrypted with a password.
`TrustedRootCertificate`	This is only relevant if `UseSecureSockets` is set to `true`). This specifies the full path to the PEM-encoded X509 certificate for the certificate authority.

HTTP server configuration

Table 8 describes the server-side configuration attributes for the HTTP transport that are defined within the `http-conf:server` element.

Table 8: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`SendTimeout`	This specifies the length of time, in milliseconds, that the server can continue to try to send a response to the client before the connection is timed out. The default is `30000`.
`ReceiveTimeout`	This specifies the length of time, in milliseconds, that the server can continue to try to receive a request from the client before the connection is timed out. The default is `30000`.
`SuppressClientSendErrors`	This specifies whether exceptions are to be thrown when an error is encountered on receiving a client request. Valid values are `true` and `false`. The default is `false`, to throw exceptions on encountering errors.
`SuppressClientReceiveErrors`	This specifies whether exceptions are to be thrown when an error is encountered on sending a response to a client. Valid values are `true` and `false`. The default is `false`, to throw exceptions on encountering errors.

Table 8: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`HonorKeepAlive`	This specifies whether the server should honor client requests for a connection to remain open after a server response has been sent to a client. Servers can achieve higher concurrency per thread by honoring requests to keep connections alive. Valid values are `true` and `false`. The default is `false`, to close the connection after a server response is sent. If set to `true`, the request socket is kept open provided the client is using at least version 1.1 of HTTP and has requested that the connection is kept alive. Otherwise, the connection is closed. If set to `false`, the socket is automatically closed after a server response is sent, even if the client has requested the server to keep the connection alive.
`RedirectURL`	This specifies the URL to which the client request should be redirected if the URL specified in the client request is no longer appropriate for the requested resource. In this case, if a status code is not automatically set in the first line of the server response, the status code is set to `302` and the status description is set to `Object Moved`. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.

Table 8: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`CacheControl`	This specifies directives about the behavior that must be adhered to by caches involved in the chain comprising a response from a server to a client. Valid values are: • `no-cache` prevents a cache from using a particular response to satisfy subsequent client requests without first revalidating that response with the server. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response. • `public` indicates that a response can be cached by any cache. • `private` indicates that a response is intended only for a single user and cannot be cached by a public (*shared*) cache. If specific response header fields are specified with this value, the restriction applies only to those header fields within the response. If no response header fields are specified, the restriction applies to the entire response. • `no-store` indicates that a cache must not store any part of a response or any part of the request that evoked it. • `no-transform` indicates that a cache must not modify the media type or location of the content in a response between a server and a client. • `must-revalidate` indicates that if a cache entry relates to a server response that has exceeded its expiration time, the cache must revalidate that cache entry with the server before it can be used in a subsequent response. • `proxy-revalidate` indicates the same as `must-revalidate`, except that it can only be enforced on shared caches and is ignored by private unshared caches. If using this directive, the `public` cache directive must also be used.

Table 8: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
	• `max-age` indicates that the client can accept a response whose age is no greater than the specified time in seconds. • `s-maxage` indicates the same as `max-age`, except that it can only be enforced on shared caches and is ignored by private unshared caches. The age specified by `s-maxage` overrides the age specified by `max-age`. If using this directive, the `proxy-revalidate` directive must also be used. • `cache-extension` indicates additional extensions to the other cache directives. Extensions might be informational or behavioral. An extended directive is specified in the context of a standard directive, so that applications not understanding the extended directive can at least adhere to the behavior mandated by the standard directive. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.
`ContentLocation`	This specifies the URL where the resource being sent in a server response is located. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.
`ContentType`	This specifies the media type of the information being sent in a server response (for example, text/html, image/gif, and so on). This is also known as the multipurpose internet mail extensions (MIME) type. MIME types are regulated by the Internet Assigned Numbers Authority (IANA). See http://www.iana.org/assignments/media-types/ for more details. Specified values consist of a main type and sub-type, separated by a forward slash. For example, a main type of `text` might be qualified as follows: `text/html` or `text/xml`. Similarly, a main type of image might be qualified as follows: `image/gif` or `image/jpeg`. The default type is `text/xml`. Other specifically supported types include: `application/jpeg`, `application/msword`, `application/xbitmap`, `audio/au`, `audio/wav`, `text/html`, `text/text`, `image/gif`, `image/jpeg`, `video/avi`, `video/mpeg`. Any content that does not fit into any type in the preceding list should be specified as `application/octet-stream`. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.

Table 8: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`ContentEncoding`	This can be used in conjunction with `ContentType`. It specifies what additional content codings have been applied to the information being sent by the server, and what decoding mechanisms the client therefore needs to retrieve the information. The primary use of `ContentEncoding` is to allow a document to be compressed using some encoding mechanism, such as zip or gzip. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.
`ServerType`	This specifies what type of server is sending the response to the client. Values in this case take the form *`program-name/version`*. For example, `Apache/1.2.5`. If this is set, it is sent as a transport attribute in the header of a response message from the server to the client.
`UseSecureSockets`	This indicates whether the server wants a secure HTTP connection running over SSL or TLS. A secure HTTP connection is commonly referred to as HTTPS. Valid values are `true` and `false`. The default is `false`, to indicate that the server does not want to open a secure connection. **Note:** If the `http-conf:client URL` attribute has a value with a prefix of `https://`, a secure HTTP connection is automatically enabled, even if `UseSecureSockets` is not set to true.
`ServerCertificate`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This specifies the full path to the PEM-encoded X509 certificate issued by the certificate authority for the server. A server must present such a certificate, so that the client can authenticate the server.

Table 8: *HTTP Server Configuration Attributes*

Configuration Attribute	Explanation
`ServerCertificateChain`	This is only relevant if the HTTP connection is running securely over SSL or TLS. PEM-encoded X509 certificates can be issued by intermediate certificate authorities that are not trusted by the client, but which have had their certificates issued in turn by a trusted certificate authority. If this is the case, you can use `ServerCertificateChain` to allow the certificate chain of PEM-encoded X509 certificates to be presented to the client for verification. This specifies the full path to the file that contains all the certificates in the chain.
`ServerPrivateKey`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This is used in conjunction with `ServerCertificate`. It specifies the full path to the PEM-encoded private key that corresponds to the X509 certificate specified by `ServerCertificate`. This is required if, and only if, `ServerCertificate` has been specified.
`ServerPrivateKeyPassword`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This specifies a password that is used to decrypt the PEM-encoded private key, if it has been encrypted with a password.
`TrustedRootCertificate`	This is only relevant if the HTTP connection is running securely over SSL or TLS. This specifies the full path to the PEM-encoded X509 certificate for the certificate authority. This is used to validate the certificate presented by the client.

Creating a CORBA Service

Overview

Generally, when you are creating a CORBA service with Artix, you need to do two things. First, you must configure the Artix port information in the Artix contract so that Artix can instantiate the appropriate port. Second, you must generate the IDL describing your service so that a native CORBA application can understand the interfaces of the new Artix service.

In this section

This section discusses the following topics:

Configuring an Artix CORBA Port

Overview

CORBA ports are described using the IONA-specific WSDL elements `<corba:address>` and `<corba:policy>` within the WSDL `<port>` element, to specify how a CORBA object is exposed.

Namespace

Example 71 shows the namespace entries you need to add to the `<definitions>` element of your contract to use the CORBA extensions.

Example 71: *Artix CORBA Extension Namespaces*

```
<definitions
  ...
  xmlns:iiop="http://schemas.iona.com/bindings/corba"
  ... >
```

Address specification

The IOR of the CORBA object is specified using the `<corba:address>` element. You have four options for specifying IORs in Artix contracts:

- Specify the objects IOR directly, by entering the object's IOR directly into the contract using the stringified IOR format:

  ```
  IOR:22342....
  ```

- Specify a file location for the IOR, using the following syntax:

  ```
  file:///file_name
  ```

  Note: The file specification requires three backslashes (`///`).

- Specify that the IOR is published to a CORBA name service, by entering the object's name using the `corbaname` format:

  ```
  corbaname:rir/NameService#object_name
  ```

  For more information on using the name service with Artix see *Deploying and Managing Artix Solutions*.

- Specify the IOR using `corbaloc`, by specifying the port at which the service exposes itself, using the `corbaloc` syntax.

  ```
  corbaloc:iiop:host:port/service_name
  ```

  When using `corbaloc`, you must be sure to configure your service to start up on the specified host and port.

Specifying POA policies

Using the optional `<corba:policy>` element, you can describe a number of POA polices the Artix service will use when creating the POA for connecting to a CORBA application. These policies include:

- POA Name
- Persistence
- ID Assignment

Setting these policies lets you exploit some of the enterprise features of IONA's Orbix 6.x, such as load balancing and fault tolerance, when deploying an Artix integration project. For information on using these advanced CORBA features, see the Orbix documentation.

POA Name

Artix POAs are created with the default name of `WS_ORB`. To specify the name of the POA Artix creates to connect with a CORBA object, you use the following:

```
<corba:policy poaname="poa_name" />
```

Persistence

By default Artix POA's have a persistence policy of `false`. To set the POA's persistence policy to `true`, use the following:

```
<corba:policy persistent="true" />
```

ID Assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by the ORB. To specify that the POA connecting a specific object should use a user-assigned ID, use the following:

```
<corba:policy serviceid="POAid" />
```

This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Example

For example, a CORBA port for the `personalInfoLookup` binding would look similar to Example 74:

Example 72: *CORBA personalInfoLookup Port*

```
<service name="personalInfoLookupService">
  <port name="personalInfoLookupPort"
        binding="tns:personalInfoLookupBinding">
    <corba:address location="file:///objref.ior" />
    <corba:policy persistent="true" />
    <corba:policy serviceid="personalInfoLookup" />
  </ port>
</ service>
```

Artix expects the IOR for the CORBA object to be located in a file called `objref.ior`, and creates a persistent POA with an object id of `personalInfo` to connect the CORBA application.

Generating CORBA IDL

Overview

Artix clients that use a CORBA transport require that the IDL defining the interface exist and be accessible. Artix provides tools to generate the required IDL from an existing WSDL contract. The generated IDL captures the information in the logical portion of the contract and uses that to generate the IDL interface. Each `<portType>` in the contract generates an IDL module.

From the command line

The `wsdltocorba` tool compiles Artix contracts and generates IDL for the specified CORBA binding and port type. To generate IDL using `wsdltocorba` use the following command:

```
wsdltocorba -idl -b binding [-corba][-i portType][-d dir]
            [-o file] wsdl_file
```

The command has the following options:

`-idl`	Instructs the tool to generate an IDL file from the specified binding.
`-b binding`	Specifies the CORBA binding from which to generate IDL.
`-corba`	Instructs the tool to generate a CORBA binding for the specified port type.
`-i portType`	Specifies the name of the port type being mapped to a CORBA binding.
`-d dir`	Specifies the directory into which the new WSDL file is written.
`-o file`	Specifies the name of the generated WSDL file. Defaults to *wsdl_file*.idl.

By combining the `-idl` and `-corba` flags with `wsdltocorba`, you can generate a CORBA binding for a logical operation and then generate the IDL for the generated CORBA binding. When doing so, you must also use the `-i portType` flag to specify the port type from which to generate the binding and the `-b binding` flag to specify the name of the binding from which to generate the IDL.

Creating an IIOP Service

Overview

Artix allows you to use IIOP as a generic transport for send data using any of the payload formats. When using IIOP as a generic transport, you define your service's address using `<iiop:address>`. The benefit of using the generic IIOP transport is that it allows you to use CORBA services without requiring your applications to be CORBA applications. For example, you could use an IIOP tunnel to send fixed format messages to an endpoint whose address is published in a CORBA naming service.

Note: Generic IIOP is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports IIOP.

Namespace

Example 73 shows the namespace entries you need to add to the `<definitions>` element of your contract to use the IIOP extensions.

Example 73: *Artix IIOP Extension Namespaces*

```
<definitions
  ...
  xmlns:iiop="http://schemas.iona.com/transports/iiop_tunnel"
  ... >
```

Address specification

The IOR, or address, of the IIOP port is specified using the `<iiop:address>` element. You have four options for specifying IORs in Artix contracts:

- Specify the objects IOR directly, by entering the object's IOR directly into the contract using the stringified IOR format:

  ```
  IOR:22342....
  ```

- Specify a file location for the IOR, using the following syntax:

  ```
  file:///file_name
  ```

Note: The file specification requires three backslashes (`///`).

- Specify that the IOR is published to a CORBA name service, by entering the object's name using the `corbaname` format:

  ```
  corbaname:rir/NameService#object_name
  ```

  For more information on using the name service with Artix see *Deploying and Managing Artix Solutions*.
- Specify the IOR using `corbaloc`, by specifying the port at which the service exposes itself, using the `corbaloc` syntax.

  ```
  corbaloc:iiop:host:port/service_name
  ```

  When using `corbaloc`, you must be sure to configure your service to start up on the specified host and port.

Specifying type of payload encoding

The IIOP transport can perform codeset negotiation on the encoded messages passed through it if your CORBA system supports it. By default, this feature is turned off so that the agents sending the message maintain complete control over codeset conversion. If you wish to turn automatic codeset negotiation on use the following:

```
<iiop:payload type="string" />
```

Specifying POA policies

Using the optional `<iiop:policy>` element, you can describe a number of POA polices the Artix service will use when creating the IIOP port. These policies include:

- POA Name
- Persistence
- ID Assignment

Setting these policies lets you exploit some of the enterprise features of IONA's Orbix 6.x, such as load balancing and fault tolerance, when deploying an Artix integration project using the IIOP transport. For information on using these advanced CORBA features, see the Orbix documentation.

POA Name

Artix POAs are created with the default name of `WS_ORB`. To specify a name of the POA that Artix creates for the IIOP port, you use the following:

```
<iiop:policy poaname="poa_name" />
```

The POA name is used for setting certain policies, such as direct persistence and well-known port numbers in the CORBA configuration.

Persistence

By default Artix POA's have a persistence policy of `false`. To set the POA's persistence policy to `true`, use the following:

```
<iiop:policy persistent="true" />
```

ID Assignment

By default Artix POAs are created with a `SYSTEM_ID` policy, meaning that their ID is assigned by Artix. To specify that the IIOP port's POA should use a user-assigned ID, use the following:

```
<corba:policy serviceid="POAid" />
```

This creates a POA with a `USER_ID` policy and an object id of *`POAid`*.

Example

For example, an IIOP port for the `personalInfoLookup` binding would look similar to Example 74:

Example 74: *CORBA personalInfoLookup Port*

```
<service name="personalInfoLookupService">
  <port name="personalInfoLookupPort"
        binding="tns:personalInfoLookupBinding">
    <iiop:address location="file:///objref.ior" />
    <iiop:policy persistent="true" />
    <iiop:policy serviceid="personalInfoLookup" />
  </ port>
</ service>
```

Artix expects the IOR for the IIOP port to be located in a file called `objref.ior`, and creates a persistent POA with an object id of `personalInfo` to configure the IIOP port.

Creating a WebSphere MQ Service

Overview

The description for an Artix WebSphere MQ port is entered in a `<port>` element of the Artix contract containing the interface to be exposed over WebSphere MQ. Artix defines two elements to describe WebSphere MQ ports and their attributes:

<mq:client> defines a port for a WebSphere MQ client application.

<mq:server> defines a port a WebSphere MQ server application.

You can use one or both of the WebSphere MQ elements to describe the Artix WebSphere MQ port. Each can have different configurations depending on the attributes you choose to set.

WebSphere MQ namespace

The WSDL extensions used to describe WebSphere MQ transport details are defined in the WSDL namespace `http://schemas.iona.com/bindings/mq`. To use the WebSphere MQ extensions you will need to include the following in the `<definitions>` tag of your contract:

```
xmlns:mq="http://schemas.iona.com/bindings/mq"
```

Required attributes

When you define a WebSphere MQ service you need to provide at least enough information for the service to connect to its message queues. For any WebSphere application that means setting the `QueueManager` and `QueueName` attributes of the port. In addition, if you are configuring a client that expects to receive replies from the server, you need to set the `ReplyQueueManager` and `ReplyQueueName` attributes of the `<mq:client>` element defining it.

In addition, if you are deploying applications on a machine that only has an MQ client installation, you need to set the `Server_Client` attribute to `client`. This setting intructs Artix to load `libmqic` instead of `libmqm`.

Example

An Artix contract exposing an interface, `monsterBash`, bound to a SOAP payload format, `Raydon`, on an WebSphere MQ queue, `UltraMan` would contain a service element similar to Example 75.

Example 75: *Sample WebSphere MQ Port*

```
<service name="Mothra">
  <port name="X" binding="tns:Raydon">
    <mq:server QueueManager="UMA"
               QueueName="UltraMan"
               ReplyQueueManager="WINR"
               ReplyQueueName="Elek"
               AccessMode="receive"
               CorrelationStyle="messageId copy"/>
  </port>
</service>
```

More information

For a detailed description of the WebSphere MQ transport configuration attributes see "WebSphere MQ Artix Extensions" on page 485.

Creating a Java Messaging System Service

Overview

Artix provides a transport plug-in that enables systems to place and receive messages from Java Messaging System (JMS) queues and topics. One large advantage of this is that Artix allows C++ applications to interact directly with Java applications over JMS.

Artix's JMS transport plug-in uses JNDI to locate and obtain references to the JMS provider that brokers for the JMS destination with which it wants to connect. Once Artix has established a connection to a JMS provider, Artix supports the passing of messages packaged as either a JMS `ObjectMessage` or a JMS `TextMessage`.

Message formatting

The JMS transport takes the payload formatting and packages it into either a JMS `ObjectMessage` or a `TextMessage`. When a message is packaged as an `ObjectMessage` the message information, including any format-specific information, is serialized into a `byte[]` and placed into the JMS message body. When a message is packaged as a `TextMessage`, the message information, including any format-specific information, is converted into a string and placed into the JMS message body.

When a message sent by Artix is received by a JMS application, the JMS application is responsible for understanding how to interpret the message and the formatting information. For example, if the Artix contract specifies that the binding used for a JMS port is SOAP, and the messages are packaged as `TextMessage`, the receiving JMS application will get a text message containing all of the SOAP envelope information. For a message encoded using the fixed binding, the message will contain no formatting information, simply a string of characters, numbers, and spaces.

Port configuration

The JMS port configuration is done by using a `<jms:address>` element in your service's `<port>` description. `<jms:address>` uses six attributes to configure the JMS connection:

`destinationStyle`	Specifies if the JMS destination is a JMS queue or a JMS topic.

`jndiProviderURL`	Specifies the URL of the JNDI service where the connection information for the JMS destination is stored.
`initialContextFactory`	Specifies the name of the `InitialContextFactory` class or a list of package prefixes used to construct URL context factory classnames. For more details on specifying a JNDI `InitialContextFactory`, see "JNDI InitialContextFactory settings" on page 331.
`jndiConnectionFactoryName`	Specifies the JNDI name bound to the JMS connection factory to use when connecting to the JMS destination.
`jndiDestinationName`	Specifies the JNDI name bound to the JMS destination to which Artix connects.
`messageType`	Specifies how the message data will be packaged as a JMS message. `text` specifies that the data will be packaged as a `TextMessage`. `binary` specifies that the data will be packaged as an `ObjectMessage`.

Using correlation IDs

If you want to configure Artix to use JMS message IDs as the correlation IDs you can set the optional `useMessageIDAsCorrelationID` attribute to true. The default for this attribute is false.

Optional JNDI settings

To increase interoperability with JMS and JNDI providers, the `<jms:address>` element has a number of optional attributes to faciltate configuring a JNDI connection. These optional attributes are:

- `java.naming.factory.initial`
- `java.naming.provider.url`
- `java.naming.factory.object`
- `java.naming.factory.state`
- `java.naming.factory.url.pkgs`
- `java.naming.dns.url`
- `java.naming.authoritative`
- `java.naming.batchsize`
- `java.naming.referral`

- java.naming.security.protocol
- java.naming.security.authentication
- java.naming.security.principal
- java.naming.security.credentials
- java.naming.language
- java.naming.applet

For more details on what information to using these attributes, check your JNDI provider's documentation and consult the Java API reference material.

Example

Example 76 shows an example of an Artix JMS port specification.

Example 76: *Artix JMS Port*

```
<service name="HelloWorldService">
  <port binding="tns:HelloWorldPortBinding" name="HelloWorldPort">
    <jms:address destinationStyle="queue"
                 jndiProviderURL="tcp://localhost:2506"
                 initialContextFactory="com.sonicsw.jndi.mfcontext.MFContextFactory"
                 jndiConnectionFactoryName="QCF"
                 jndiDestinationName="testQueue"
                 messageType="text" />
  </port>
</service>
```

JNDI InitialContextFactory settings

The usual method of specifying the JNDI is to enter the class name provided by your JNDI provider. In Example 76, the JMS port is using the JNDI provided with SonicMQ and the class specified, `com.sonicsw.jndi.mfcontext.MFContextFactory`, is the class used by Sonic's JNDI server to create a JNDI context.

Alternatively, you can specify a colon-separated list of package prefixes to use when loading URL context factories. The JNDI service takes each package prefix and appends the URL schema name to form a sub-package. It then prepends the URL schema name to `URLContextFactory` to form a class name within the sub-package. Once the new class name is formed, the JNDI service then tries to instantiate the class using the newly formed name. For example, if your Artix contract described the JMS port shown in

Example 77, the JNDI service would instantiate a context factory with the class name `com.iona.jbus.jms.naming.sonic.sonicURLContextFactory` to perform lookups.

Example 77: *JMS Port with Alternate InitialContextFactory Specification*

```
<service name="HelloWorldService">
  <port binding="tns:HelloWorldPortBinding" name="HelloWorldPort">
    <jms:address destinationStyle="queue"
                 jndiProviderURL="tcp://localhost:2506"
                 initialContextFactory="com.iona.jbus.jms.naming"
                 jndiConnectionFactoryName="sonic:jms/queue/connectionFactory"
                 jndiDestinationName="sonic:jms/queue/helloWorldQueue"
                 messageType="text" />
  </port>
</service>
```

The `URLContextFactory` then uses the URL specified in the `jndiConnectionFactoryName` and the `jndiDestinationFactoryName` attributes to obtain references to the desired JMS `ConnectionFactory` and the desired JMS `Destination`. The JNDI service is completely bypassed using this method and allows you to connect to JMS implementations that do not use JNDI or to connect to JMS `Destination` that are not registered with the JNDI service.

So instead of looking up the JMS `ConnectionFactory` using the JNDI name bound to it, Artix will get a reference directly to `ConnectionFactory` using the name given to it when it was created. Using the contract in Example 77, Artix would use the URL `sonic:jms/queue/helloWorldQueue` to get a reference to the desired queue. Artix would be handed a reference to a queue named `helloWorldQueue` if the JMS broker has such a queue.

Note: Due to a known bug in the SonicMQ JNDI service, it is recommended that you use this method of specifying the `InitialContextFactory` when using SonicMQ.

Adding a TIBCO Service

Overview

The TIBCO Rendezvous transport lets you use Artix to integrate systems based on TIBCO Rendezvous (TIB/RV) software.

Note: TIBCO Rendezvous integration is unavailable in some editions of Artix. Please check the conditions of your Artix license to see whether your installation supports TIBCO Rendezvous integration.

Supported Features

Table 9 shows the matrix of TIBCO Rendezvous features Artix supports.

Table 9: *Supported TIBCO Rendezvous Features*

Feature	Supported	Not Supported
Server Side Advisory Callbacks	x	
Certified Message Delivery	x	
Fault Tolerance (`TibrvFtMember/Monitor`)		x
Virtual Connections (`TibrvVcTransport`)		x
Secure Daemon (`rvsd/TibrvSDContext`)		x
`TIBRVMSG_IPADDR32`		x
`TIBRVMSG_IPPORT16`		x

Namespace

To use the TIB/RV transport, you need to describe the port using TIB/RV in the physical part of an Artix contract. The extensions used to describe a TIB/RV port are defined in the namespace:

```
xmlns:tibrv="http://schemas.iona.com/transports/tibrv"
```

This namespace will need to be included in your Artix contract's `<definition>` element.

Describing the port

As with other transports, the TIB/RV transport description is contained within a `<port>` element. Artix uses `<tibrv:port>` to describe the attributes of a TIB/RV port. The only required attribute for a `<tibrv:port>` is `serverSubject` which specifies the subject to which the server listens.

Example

Example 78 shows an Artix description for a TIB/RV port.

Example 78: *TIB/RV Port Description*

```
<service name="BaseService">
    <port binding="tns:BasePortBinding" name="BasePort">
        <tibrv:port serverSubject="Artix.BaseService.BasePort"
        />
    </port>
</service>
```

More information

For a complete listing of the attribute used in configuring a TIB/RV service see "Tibco Transport Extensions" on page 521.

Creating a Tuxedo Service

Overview

Artix allows services to connect using Tuxedo's transport mechanism. This provides them with all of the qualities of service associated with Tuxedo.

Tuxedo namespaces

To use the Tuxedo transport, you need to describe the port using Tuxedo in the physical part of an Artix contract. The extensions used to describe a Tuxedo port are defined in the following namespace:

```
xmlns:tuxedo="http://schemas.iona.com/transports/tuxedo"
```

This namespace will need to be included in your Artix contract's `<definition>` element.

Defining the Tuxedo services

As with other transports, the Tuxedo transport description is contained within a `<port>` element. Artix uses `<tuxedo:server>` to describe the attributes of a Tuxedo port. `<tuxedo:server>` has a child element, `<tuxedo:service>`, that gives the bulletin board name of a Tuxedo port. The bulletin board name for the service is specified in the element's `name` attribute. You can define more than one Tuxedo service to act as an endpoint.

Mapping operations to a Tuxedo service

For each of the Tuxedo services that are endpoints, you must specify which of the operations bound to the port being defined are handled by the Tuxedo service. This is done using one or more `<tuxedo:input>` child elements. `<tuxedo:input>` takes one required attribute, `operation`, that specifies the WSDL operation that is handled by this Tuxedo service endpoint.

Example

An Artix contract exposing the `personalInfoService`, defined in Example 32 on page 252, as a Tuxedo service would contain a `<service>` element similar to Example 79 on page 336.

Example 79: *Tuxedo Port Description*

```
<service name="personalInfoService">
  <port binding="tns:personalInfoBinding" name="tuxInfoPort">
    <tuxedo:server>
      <tuxedo:service name="personalInfoService">
        <tuxedo:input operation="infoRequest" />
      </tuxedo:service>
    </tuxedo:server>
  </port>
</service>
```

CHAPTER 15

Creating Artix Contracts from Existing Applications

Artix provides a number of command line tools to help you create contracts from applications that you have already developed.

In this chapter

This chapter discusses the following topics:

Creating Artix Contracts from CORBA IDL

Overview

If you are starting from a CORBA server or client, Artix can build the logical portion of the Artix contract from IDL. Contracts generated from IDL have CORBA-specific entries and namespaces added.

The IDL to WSDL compiler also generates the binding information required to format the operations specified in the IDL. However, since port information is specific to the deployment environment, the port information is left blank.

CORBA WSDL namespaces

Contracts generated from IDL include two additional name spaces:

```
xmlns:corba="http://schemas.iona.com/bindings/corba"
xmlns:corbatm="http://schemas.iona.com/bindings/corba/typemap"
```

Unsupported type handling

Be aware that the IDL to WSDL compiler ignores any definitions that use unsupported CORBA types. The IDL to WSDL compiler also ignores any definition that uses a previously ignored definition. For example, assume you have the following IDL definitions in `file.idl`:

```
interface A
{
  struct S
  {
    A member;
  };

  S get_op();
};
```

The IDL to WSDL compiler does not generate any corresponding contract information for the structure `S` because it contains a member that uses an object reference. Similarly, the IDL to WSDL compiler does not generate any contract information for the operation `get_op()` because it references structure `S`.

Using the command line

IONA's IDL to WSDL compiler supports several command line flags that specify how to create a WSDL file from an IDL file. The default behavior of the tool is to create WSDL file that uses wrapped doc/literal style messages. Wraped doc/literal style messages have a single part, defined using an element, that wraps all of the elements in the message. See Example 81 on page 341 for a sample.

The IDL to WSDL compiler is run using the following command:

```
idltowsdl [-useypes][-unwrap][-a address][-f file][-o dir][-s type][-r file][-L file][-P file]
          [-w namespace][-x namespace][-t namespace][-T file][-n file][-b] idlfile
```

The command has the following options:

`-usetypes`	Generate rpc style messages. rpc style messages have parts defined using XMLSchema types instead of XML elements.
`-unwrap`	Generate unwrapped doc/literal messages. Unwrapped messages have parts that represent individual elements. Unlike wrapped messages, unwrapped messages can have multiple parts and are not allowed by the WS-I.
`-a` *`address`*	Specifies an absolute address through which the object reference may be accessed. The *`address`* may be a relative or absolute path to a file, or a corbaname URL
`-f` *`file`*	Specifies a file containing a string representation of an object reference. The object reference is placed in the `<corba:address>` element in the `<port>` definition of the generated service. The *`file`* must exist when you run the IDL compiler.
`-o` *`dir`*	Specifies the directory into which the WSDL file is written.
`-s` *`type`*	Specifies the XMLSchema type used to map the IDL `sequence<octet>` type. Valid values are `base64Binary` and `hexBinary`. The default is `base64Binary`.
`-r` *`file`*	Specify the pathname of the schema file imported to define the `Reference` type. If the `-r` option is not given, the idl compiler gets the schema file pathname from `etc/idl.cfg`.

`-L` *`file`*	Specifies that the logical portion of the generated WSDL specification into is written to *`file`*. *`file`* is then imported into the default generated file.
`-P` *`file`*	Specifies that the physical portion of the generated WSDL specification into is written to *`file`*. *`file`* is then imported into the default generated file.
`-w` *`namespace`*	Specifies the namespace to use for the WSDL `targetNamespace`. The default is `http://schemas.iona.com/idl/`*`idl_name`*.
`-x` *`namespace`*	Specifies the namespace to use for the Schema `targetNamespace`. The default is `http://schemas.iona.com/idltypes/`*`idl_name`*.
`-t` *`namespace`*	Specifies the namespace to use for the CORBA TypeMapping `targetNamespace`. The default is `http://schemas.iona.com/typemap/corba/`*`idl_name`*.
`-T` *`file`*	Specifies that the schema types are to be generated into a separate file. The schema file is included in the generated contract using an import statement. This option cannot be used with the `-n` option.
`-n` *`file`*	Specifies that a schema file, `file`, is to be included in the generated contract by an import statement. This option cannot be used with the `-T` option.
`-b`	Specifies that bounded strings are to be treated as unbounded. This eliminates the generation of the special types for the bounded string.

To combine multiple flags in the same command, use a colon-delimited list. The colon is only interpreted as a delimiter if it is followed by a dash. Consequently, the colons in a `corbaname` URL are interpreted as part of the URL syntax and not as delimiters.

Note: The command line flag entries are case sensitive even on Windows. Capitalization in your generated WSDL file must match the capitalization used in the prewritten code.

Example

Imagine you needed to generate an Artix contract for a CORBA server that exposes the interface shown in Example 80.

Example 80: *personalInfoService Interface*

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

  struct personalInfo
  {
    string name;
    long age;
    hairColorType hairColor;
  };

  exception idNotFound
  {
    short id;
  };

  personalInfo lookup(in long empId)
  raises (idNotFound);
};
```

To generate the contract, you run it through the IDL compiler using either the GUI or the command line. The resulting contract is similar to that shown in Example 81.

Example 81: *personalInfoService Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="personalInfo.idl"
 targetNamespace="http://schemas.iona.com/idl/personalInfo.idl"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.iona.com/idl/personalInfo.idl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://schemas.iona.com/idltypes/personalInfo.idl"
 xmlns:corba="http://schemas.iona.com/bindings/corba"
 xmlns:corbatm="http://schemas.iona.com/typemap/corba/personalInfo.idl"
 xmlns:references="http://schemas.iona.com/references">
  <types>
    <schema targetNamespace="http://schemas.iona.com/idltypes/personalInfo.idl"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
```

Example 81: *personalInfoService Contract*

```
    <xsd:simpleType name="personalInfoService.hairColorType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="red"/>
        <xsd:enumeration value="brunette"/>
        <xsd:enumeration value="blonde"/>
      </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="personalInfoService.personalInfo">
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="age" type="xsd:int"/>
        <xsd:element name="hairColor" type="xsd1:personalInfoService.hairColorType"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="personalInfoService.idNotFound">
      <xsd:sequence>
        <xsd:element name="id" type="xsd:short"/>
      </xsd:sequence>
    </xsd:complexType>
    <xsd:element name="personalInfoService.lookup">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="empId" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="personalInfoService.lookupResult">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="return" type="xsd1:personalInfoService.personalInfo"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="personalInfoService.idNotFound"
                 type="xsd1:personalInfoService.idNotFound"/>
  </schema>
</types>
<message name="personalInfoService.lookup">
  <part name="parameters" element="xsd1:personalInfoService.lookup"/>
</message>
<message name="personalInfoService.lookupResponse">
  <part name="parameters" element="xsd1:personalInfoService.lookupResult"/>
</message>
```

Example 81: *personalInfoService Contract*

```
<message name="personalInfoService.idNotFound">
  <part name="exception" element="xsd1:personalInfoService.idNotFound"/>
</message>
<portType name="personalInfoService">
  <operation name="lookup">
    <input message="tns:personalInfoService.lookup" name="lookup"/>
    <output message="tns:personalInfoService.lookupResponse" name="lookupResponse"/>
    <fault message="tns:personalInfoService.idNotFound" name="personalInfoService.idNotFound"/>
  </operation>
</portType>
<binding name="personalInfoServiceBinding" type="tns:personalInfoService">
  <corba:binding repositoryID="IDL:personalInfoService:1.0"/>
  <operation name="lookup">
    <corba:operation name="lookup">
      <corba:param name="empId" mode="in" idltype="corba:long"/>
      <corba:return name="return" idltype="corbatm:personalInfoService.personalInfo"/>
      <corba:raises exception="corbatm:personalInfoService.idNotFound"/>
    </corba:operation>
    <input/>
    <output/>
    <fault name="personalInfoService.idNotFound"/>
  </operation>
</binding>
<service name="personalInfoServiceService">
  <port name="personalInfoServicePort" binding="tns:personalInfoServiceBinding">
    <corba:address location="..."/>
  </port>
</service>
<corba:typeMapping targetNamespace="http://schemas.iona.com/typemap/corba/personalInfo.idl">
  <corba:enum name="personalInfoService.hairColorType"
              type="xsd1:personalInfoService.hairColorType"
              repositoryID="IDL:personalInfoService/hairColorType:1.0">
    <corba:enumerator value="red"/>
    <corba:enumerator value="brunette"/>
    <corba:enumerator value="blonde"/>
  </corba:enum>
  <corba:struct name="personalInfoService.personalInfo"
                type="xsd1:personalInfoService.personalInfo"
                repositoryID="IDL:personalInfoService/personalInfo:1.0">
    <corba:member name="name" idltype="corba:string"/>
    <corba:member name="age" idltype="corba:long"/>
    <corba:member name="hairColor" idltype="corbatm:personalInfoService.hairColorType"/>
  </corba:struct>
```

Example 81: *personalInfoService Contract*

```
    <corba:exception name="personalInfoService.idNotFound"
                     type="xsd1:personalInfoService.idNotFound"
                     repositoryID="IDL:personalInfoService/idNotFound:1.0">
      <corba:member name="id" idltype="corba:short"/>
    </corba:exception>
  </corba:typeMapping>
</definitions>
```

Creating Contracts from Java Classes

Overview

Many applications have been developed using Java to take advantage of Java's platform independence among other things. Java's platform independence is a perfect complement to Artix's transport independence. To facilitate the integration of Java applications with Artix, Artix provides tools for generating the logical portion of an Artix contract from existing Java classes. These tools use the mapping rules described in Sun's JAX-RPC 1.1 specification.

javatowsdl tool

Artix supplies a command line tool, `javatowsdl`, that generates the logical portion of an Artix contract for existing Java class files. To generate the logical portion of an Artix contract using the `javatowsdl` tool use the following command:

```
javatowsdl  [-t namespace][-x namespace][-i porttype]
            [-o file][-useTypes][-v][-?] ClassName
```

The command has the following options:

`-t` *`namespace`*	Specifies the target namespace of the generated WSDL document. By default, the java package name will be used as the target namespace. If no package name is specified, the generated target namespace will be `http:\\www.iona.com\`*`ClassName`*.
`-x` *`namespace`*	Specifies the target namespace of the XMLSchema information generated to represent the data types inside the WSDL document. By default, the generated target namespace of the XMLSchema will be `http:\\www.iona.com\`*`ClassName`*`\xsd`.
`-i` *`porttype`*	Specifies the name of the generated `<portType>` in the WSDL document. By default, the name of the class from which the WSDL is generated is used.
`-o` *`file`*	Specifies output file into which the WSDL is written.

`-useTypes`	Specifies that the generated WSDL will use types in the WSDL message parts. By default, messages are generated using wrapped `doc/literal` style. A wrapper element with a sequence will be created to hold method parameters.
`-v`	Prints out the version of the tool.
`-?`	Prints out a help message explaining the command line flags.

The generated WSDL will not contain any physical details concerning the payload formats or network transports that will be used when exposing the service. You will need to add this information manually.

Note: When generating contracts, `javatowsdl` will add newly generated WSDL to an existing contract if a contract of the same name exists. It will not generate a new file or warn you that a previous contract exists.

Supported types

Table 10 shows the Java types Artix can map to an Artix contract.

Table 10: *Java to WSDL Mappings*

Java	Artix Contract
`boolean`	`xsd:boolean`
`byte`	`xsd:byte`
`short`	`xsd:short`
`int`	`xsd:int`
`long`	`xsd:long`
`float`	`xsd:float`
`double`	`xsd:double`
`byte[]`	`xsd:base64binary`
`java.lang.String`	`xsd:string`
`java.math.BigInteger`	`xsd:integer`
`java.math.BigDecimal`	`xsd:decimal`

Table 10: *Java to WSDL Mappings*

Java	Artix Contract
java.util.Calendar	xsd:dateTime
java.util.Date	xsd:dateTime
java.xml.namespace.QName	xsd:QName
java.net.URI	xsd:anyURI

In the case of helper classes for a Java primitive, such as java.lang.Integer, the instance is mapped to an element with the nillable attribute set to true and the type set to the corresponding Java primitive type. Example 82 shows the mapping for a java.lang.Float.

Example 82: *Mapping of java.lang.Float to XMLSchema*

```
<element name="floatie" nillable="true" type="xsd:float" />
```

Exceptions

Artix will map user-defined exceptions to the logical Artix contract according to the rules laid out in the JAX-RPC specification. The exception will be mapped to a <fault> within the operation representing the corresponding Java method. The generated <fault> will reference a generated <message> describing the Java exception class. The name attribute of the <message> will be taken from the name of the Java exception class.

Because SOAP only supports <fault> messages with a single <part>, the generated <message> is mapped to have only one <part>. When the Java exception only has one field, it is used as the <part> and its name and type attributes are mapped from the exception's field. When the Java exception contains more than one field, Artix generates a <complexType> to describe the exception's data. The generated <complexType> will have one element for each field of the exception. The name and type attributes of the generated element will be taken from the corresponding field in the exception.

Note: Standard Java exceptions are not mapped into the generated Artix contract.

Example

For example, if you had a Java interface similar to that shown in Example 83, you could generate an Artix contract on it by compiling the interface into a `.class` file and running the command `javatowsdl Base`.

Example 83: *Base Java Class*

```
//Java
public interface Base
{
  public byte[] echoBase64(byte[] inputBase64);

  public boolean echoBoolean(boolean inputBoolean);

  public float echoFloat(float inputFloat);

  public float[] echoFloatArray(float[] inputFloatArray);

  public int echoInteger(int inputInteger);

  public int[] echoIntegerArray(int[] inputIntegerArray);
}
```

The resulting Artix contract will be similar to Example 84.

Example 84: *Base Artix Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="Base" targetNamespace="http://www.iona.com/Base"
    xmlns:ns1="http://www.iona.com/Base" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsd1="http://www.iona.com/Base/xsd">
    <wsdl:types>
        <schema targetNamespace="http://www.iona.com/Base/xsd"
            xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="echoBoolean">
                <complexType>
                    <sequence>
                        <element name="booleanParam0" type="xsd:boolean"/>
                    </sequence>
                </complexType>
            </element>
```

Example 84: *Base Artix Contract*

```
        <element name="echoBooleanResponse">
            <complexType>
                <sequence>
                    <element name="return" type="xsd:boolean"/>
                </sequence>
            </complexType>
        </element>
        <element name="echoBase64">
            <complexType>
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="_bParam0"
                        type="xsd:byte"/>
                </sequence>
            </complexType>
        </element>
        <element name="echoBase64Response">
            <complexType>
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="return"
                        type="xsd:byte"/>
                </sequence>
            </complexType>
        </element>
        <element name="echoHexBinary">
            <complexType>
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="_bParam0"
                        type="xsd:byte"/>
                </sequence>
            </complexType>
        </element>
        <element name="echoHexBinaryResponse">
            <complexType>
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="return"
                        type="xsd:byte"/>
                </sequence>
            </complexType>
        </element>
```

Example 84: *Base Artix Contract*

```
            <element name="echoFloat">
                <complexType>
                    <sequence>
                        <element name="floatParam0" type="xsd:float"/>
                    </sequence>
                </complexType>
            </element>
            <element name="echoFloatResponse">
                <complexType>
                    <sequence>
                        <element name="return" type="xsd:float"/>
                    </sequence>
                </complexType>
            </element>
            <element name="echoFloatArray">
                <complexType>
                    <sequence>
                        <element maxOccurs="unbounded" minOccurs="0" name="_fParam0"
                            type="xsd:float"/>
                    </sequence>
                </complexType>
            </element>
            <element name="echoFloatArrayResponse">
                <complexType>
                    <sequence>
                        <element maxOccurs="unbounded" minOccurs="0" name="return"
                            type="xsd:float"/>
                    </sequence>
                </complexType>
            </element>
            <element name="echoInteger">
                <complexType>
                    <sequence>
                        <element name="intParam0" type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
            <element name="echoIntegerResponse">
                <complexType>
                    <sequence>
                        <element name="return" type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
```

Example 84: *Base Artix Contract*

```
            <element name="echoIntegerArray">
                <complexType>
                    <sequence>
                        <element maxOccurs="unbounded" minOccurs="0" name="_iParam0"
                            type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
            <element name="echoIntegerArrayResponse">
                <complexType>
                    <sequence>
                        <element maxOccurs="unbounded" minOccurs="0" name="return"
                            type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
    </wsdl:types>
    <wsdl:message name="echoBoolean">
        <wsdl:part element="xsd1:echoBoolean" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoBooleanResponse">
        <wsdl:part element="xsd1:echoBooleanResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoBase64">
        <wsdl:part element="xsd1:echoBase64" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoBase64Response">
        <wsdl:part element="xsd1:echoBase64Response" name="parameters"/>
    </wsdl:message>
```

Example 84: *Base Artix Contract*

```
    <wsdl:message name="echoHexBinary">
        <wsdl:part element="xsd1:echoHexBinary" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoHexBinaryResponse">
        <wsdl:part element="xsd1:echoHexBinaryResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloat">
        <wsdl:part element="xsd1:echoFloat" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloatResponse">
        <wsdl:part element="xsd1:echoFloatResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloatArray">
        <wsdl:part element="xsd1:echoFloatArray" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoFloatArrayResponse">
        <wsdl:part element="xsd1:echoFloatArrayResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoInteger">
        <wsdl:part element="xsd1:echoInteger" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoIntegerResponse">
        <wsdl:part element="xsd1:echoIntegerResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoIntegerArray">
        <wsdl:part element="xsd1:echoIntegerArray" name="parameters"/>
    </wsdl:message>
    <wsdl:message name="echoIntegerArrayResponse">
        <wsdl:part element="xsd1:echoIntegerArrayResponse" name="parameters"/>
    </wsdl:message>
    <wsdl:portType name="Base">
        <wsdl:operation name="echoBoolean">
            <wsdl:input message="ns1:echoBoolean" name="echoBoolean"/>
            <wsdl:output message="ns1:echoBooleanResponse" name="echoBoolean"/>
        </wsdl:operation>
        <wsdl:operation name="echoBase64">
            <wsdl:input message="ns1:echoBase64" name="echoBase64"/>
            <wsdl:output message="ns1:echoBase64Response" name="echoBase64"/>
        </wsdl:operation>
```

Example 84: *Base Artix Contract*

```
        <wsdl:operation name="echoHexBinary">
            <wsdl:input message="ns1:echoHexBinary" name="echoHexBinary"/>
            <wsdl:output message="ns1:echoHexBinaryResponse" name="echoHexBinary"/>
        </wsdl:operation>
        <wsdl:operation name="echoFloat">
            <wsdl:input message="ns1:echoFloat" name="echoFloat"/>
            <wsdl:output message="ns1:echoFloatResponse" name="echoFloat"/>
        </wsdl:operation>
        <wsdl:operation name="echoFloatArray">
            <wsdl:input message="ns1:echoFloatArray" name="echoFloatArray"/>
            <wsdl:output message="ns1:echoFloatArrayResponse" name="echoFloatArray"/>
        </wsdl:operation>
        <wsdl:operation name="echoInteger">
            <wsdl:input message="ns1:echoInteger" name="echoInteger"/>
            <wsdl:output message="ns1:echoIntegerResponse" name="echoInteger"/>
        </wsdl:operation>
        <wsdl:operation name="echoIntegerArray">
            <wsdl:input message="ns1:echoIntegerArray" name="echoIntegerArray"/>
            <wsdl:output message="ns1:echoIntegerArrayResponse" name="echoIntegerArray"/>
        </wsdl:operation>
    </wsdl:portType>
</wsdl:definitions>
```

Creating Contracts from COBOL Copybooks

Overview

To facilitate the mapping of COBOL operations to Artix contracts, Artix provides a command line tool, `coboltowsdl`, that will import COBOL copybook data and generate an Artix contract containing a fixed binding to define the COBOL interface for Artix applications.

Using the tool

To generate an Artix contract from COBOL copybook data use the following command:

```
coboltowsdl -b binding -op operation -im [inmessage:]incopybook
            [-om [outmessage:]outcopybook]
            [-fm [faultmessage:]faultbook]
            [-i portType][-t target]
            [-x schema_name][-useTypes][-o file]
```

The command has the following options:

`-b binding`	Specifies the name for the generated binding.
`-op operation`	Specifies the name for the generated operation.
`-im [inmessage:]incopybook`	Specifies the name of the input message and the copybook file from which the data defining the message is taken. The input message name, *inmessage*, is optional. However, if the copybook has more than one `01` levels, you will be asked to choose the one you want to use as the input message.
`-om [outmessage:]outcopybook`	Specifies the name of the output message and the copybook file from which the data defining the message is taken. The output message name, *outmessage*, is optional. However, if the copybook has more than one `01` levels, you will be asked to choose the one you want to use as the output message.

`-fm [faultmessage:]faultbook`	Specifies the name of a fault message and the copybook file from which the data defining the message is taken. The fault message name, *faultmessage*, is optional. However, if the copybook has more than one `01` levels, you will be asked to choose the one you want to use as the fault message. You can specify more than one fault message.
`-i portType`	Specifies the name of the port type in the generated WSDL. Defaults to *binding*`PortType`.[a]
`-t target`	Specifies the target namespace for the generated WSDL. Defaults to `http://www.iona.com/`*binding*.
`-x schema_name`	Specifies the namespace for the schema in the generated WSDL. Defaults to `http://www.iona.com/`*binding*`/types`.
`-useTypes`	Specifies that the generated WSDL will use `<types>`. Default is to generate `<element>` for schema types.
`-o file`	Specifies the name of the generated WSDL file. Defaults to *binding*`.wsdl`.

a. If *binding* ends in `Binding` or `binding`, it is stripped off before being used in any of the default names.

Once the new contract is generated, you will still need to add the port information before you can use the contract to develop an Artix solution.

CHAPTER 16

Adding Routing Instructions

Artix provides messages routing based on operations, ports, or message attributes.

In this chapter

This chapter discusses the following topics:

Artix Routing

Overview

Artix routing is implemented within Artix service access points and is controlled by rules specified in the SAP's contract. Artix SAPs that include routing rules can be deployed either in standalone mode or embedded into an Artix service.

Artix supports the following types of routing:

- Port-based
- Operation-based

A router's contract must include definitions for the source services and destination services. The contract also defines the routes that connect source and destination ports, according to some specified criteria. This routing information is all that is required to implement port-based or operation-based routing. Content-based routing requires that application code be written to implement the routing logic.

Port-based

Port-based routing acts on the port or transport-level identifier, specified by a `<port>` element in an Artix contract. This is the most efficient form of routing. Port-based routing can also make a routing decision based on port properties, such as the message header or message identifier. Thus Artix can route messages based on the origin of a message or service request, or based on the message header or identifier.

Operation-based

Operation-based routing lets you route messages based on the logical operations described in an Artix contract. Messages can be routed between operations whose arguments are equivalent. Operation-based routing can be specified on the interface, `<portType>`, level or the finer grained operation level.

Compatibility of Ports and Operations

Overview

Artix can route messages between services that expect similar messages. The services can use different message transports and different payload formats, but the messages must be logically identical. For example, if you have a baseball scoring service that transmits data using SOAP over HTTP, Artix can route the score data to a reporting service that consumes data using CORBA. The only requirement for operation-based routing is that the two services have an operation that uses messages with the same logical description in the Artix contract defining their integration. For port-based routing, the destination service must have a matching operation defined for each of the operations defined for the source service.

Port-based routing

Port-based routing is rough grained in that the routing rules are defined on the `<port>` elements of an Artix contract and do not look at the individual operations defined in the logical interface, or `<portType>`, to which the port is bound. Therefore, port-based routing requires that the services between which messages are being routed must have compatible logical interface descriptions.

For two ports to have compatible logical interfaces the following conditions must be met:

- The destination's logical interface must contain a matching operation for each operation in the source's logical interface. Matching operations must have the same name.
- Each of the matching operations must have the same number of input, output, and fault messages.
- Each of the matching operations' messages must have the same sequence of port types.

For example, given the two logical interfaces defined in Example 85 you could construct a route from a port bound to `baseballScorePortType` to a port bound to `baseballGamePortType`. However, you could not create a

route from a port bound to `finalScorePortType` to a port bound to `baseballGamePortType` because the message types used for the `getScore` operation do not match.

Example 85: *Logical interface compatibility example*

```
<message name="scoreRequest>
  <part name="gameNumber" type="xsd:int" />
</message>
<message name="baseballScore">
  <part name="homeTeam" type="xsd:int" />
  <part name="awayTeam" type="xsd:int" />
  <part name="final" type="xsd:boolean" />
</message>
<message name="finalScore">
  <part name="home" type="xsd:int" />
  <part name="away" type="xsd:int" />
  <part name="winningTeam" type="xsd:string" />
</message>
<message name="winner">
  <part name="winningTeam" type="xsd:string" />
</message>
<portType name="baseballGamePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:basballScore" name="baseballScore"/>
  </operation>
  <operation name="getWinner">
    <input message="tns:scoreRequest" name="winnerRequest"/>
    <output message="tns:winner" name="winner"/>
  </operation>
</portType>
<portType name="baseballScorePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:basballScore" name="baseballScore"/>
  </operation>
</portType>
<portType name="finalScorePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:finalScore" name="finalScore"/>
  </operation>
</portType>
```

Operation-based routing

Operation-based routing provides a finer grained level of control over how messages can be routed. Operation-based routing rules check for compatibility on the `<operation>` level of the logical interface description. Therefore, messages can be routed between any two compatible messages.

The following conditions must be met for operations to be compatible:

- The operations must have the same number of input, output, and fault messages.
- The messages must have the same sequence of part types.

For example, if you added the logical interface in Example 86 to the interfaces in Example 85 on page 360, you could specify a route from `getFinalScore` defined in `fullScorePortType` to `getScore` defined in `finalScorePortType`. You could also define a route from `getScore` defined in `fullScorePortType` to `getScore` defined in `baseballScorePortType`.

Example 86: *Operation-based routing interface*

```
<portType name="fullScorePortType">
  <operation name="getScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:basballScore" name="baseballScore"/>
  </operation>
  <operation name="getFinalScore">
    <input message="tns:scoreRequest" name="scoreRequest"/>
    <output message="tns:finalScore" name="finalScore"/>
  </operation>
</portType>
```

Defining Routes in Artix Contracts

Overview

Artix port-based and operation-based routing are fully implemented in the contract defining the integration of your systems. Routes are defined using WSDL extensions that are defined in the namespace `http://schemas.iona.com/routing`. The most commonly used of these extensions are:

<routing:route> is the root element of any route defined in the contract.

<routing:source> specifies the port that serves as the source for messages that will be routed using the route.

<routing:destination> specifies the port to which messages will be routed.

You do not need to do any programming and your applications need not be aware that any routing is taking place.

In this section

This section discusses the following topics:

Using Port-Based Routing

Overview

Port-based routing is the highest performance type of routing Artix performs. It is also the easiest to implement. All of the rules are specified in the Artix contract describing how your systems are integrated. The routes specify the source port for the messages and the destination port to which messages are routed.

Describing routes in an Artix contract

The Artix routing elements are defined in the `http://schemas.iona.com/routing` namespace. When describing routes in an Artix contract you must add the following to your contract's definition element:

```
<definition ...
  xmlns:routing="http://schemas.iona.com/routing"
  ...>
```

To describe a port-based route you use three elements:

<routing:route>

`<routing:route>` is the root element of each route you describe in your contract. It takes on required attribute, `name`, that specifies a unique identifier for the route. `route` also has an optional attribute, `multiRoute`, which is discussed in "Advanced Routing Features" on page 369.

<routing:source>

`<routing:source>` specifies the port from which the route will redirect messages. A route can have several source elements as long as they all meet the compatibility rules for port-based routing discussed in "Port-based routing" on page 359.

`<routing:source>` requires two attributes, `service` and `port`. `service` specifies the service element in which the source port is defined. `port` specifies the name of the port element from which messages are being received.

<routing:destination>

<routing:destination> specifies the port to which the source messages are directed. The destination must be compatible with all of the source elements. For a discussion of the compatibility rules for port-based routing see "Port-based routing" on page 359.

In standard routing only one destination is allowed per route. Multiple destinations are allowed in conjunction with the route element's muliRoute attribute that is discussed in "Advanced Routing Features" on page 369.

<routing:destination> requires two attributes, service and port. service specifies the service element in which the destination port is defined. port specifies the name of the port element to which messages are being sent.

Example

For example, to define a route from baseballScorePortType to baseballGamePortType, defined in Example 85 on page 360, your Artix contract would contain the elements in Example 87.

Example 87: *Port-based routing example*

```
1  <service name="baseballScoreService">
     <port binding="tns:baseballScoreBinding"
           name="baseballScorePort">
       <soap:address location="http://localhost:8991"/>
     </port>
   </service>
   <service name="baseballGameService">
     <port binding="tns:baseballGameBinding"
           name="baseballGamePort">
       <corba:address location="file://baseball.ref"/>
     </port>
   </service>
2  <routing:route name="baseballRoute">
     <routing:source service="tns:baseballScoreService"
                     port="tns:baseballScorePort" />
     <routing:destination service="tns:baseballGameService"
                          port="tns:baseballGamePort" />
   </routing:route>
```

There are two sections to the contract fragment shown in Example 87:

1. The logical interfaces must be bound to physical ports in `<service>` elements of the Artix contract.
2. The route, `baseballRoute`, is defined with the appropriate service and port attributes.

Using Operation-Based Routing

Overview

Operation-based routing is a refinement of port-based routing. With operation-based routing you can specify specific operations within a logical interface as a source or a destination.

Like port-based routing, operation-based routing is fully implemented by adding routing rules to Artix contracts.

Describing routes in an Artix contract

The contract elements for defining operation-based routes are defined in the same namespace as the elements for port-based routing and you will need to include in your contract's namespace declarations to use operation based routing.

To specify an operation-based route you need to specify one additional element in your route description: `<routing:operation>`. `<routing:operation>` specifies an operation defined in the source port's logical interface and an optional target operation in the destination port's logical interface. You can specify any number of operation elements in a route. The operation elements must be specified after all of the source elements and before any destination elements.

`operation` takes one required attribute, `name`, that specifies the name of the operation in the source port's logical interface that is to be used in the route.

`operation` also has an optional attribute, `target`, that specifies the name operation in the destination port's logical interface to which the message is to be sent. If a target is specified, messages are routed between the two operations. If no target is specified, the source operation's name is used as the name of the target operation. The source and target operations must meet the compatibility requirements discussed in "Operation-based routing" on page 361.

How operation-based rules are applied

Operation-based routing rules apply to all of the source elements listed in the route. Therefore, if an operation-based routing rule is specified, a message will be routed if all of the following are true:

- The message is received from one of the ports specified in a source element.

- The operation name associated with the received message is specified in one of the `<operation>` elements.

If there are multiple operation-based rules in the route, the message will be routed to the destination specified in the matching operation's `target` attribute.

Example

For example to route messages from `getFinalScore` defined in `fullScorePortType`, shown in Example 86 on page 361, to `getScore` defined in `finalScorePortType`, shown in Example 85 on page 360, your Artix contract would contain the elements in Example 88.

Example 88: *Operation to Operation Routing*

```
1 <service name="fullScoreService">
    <port binding="tns:fullScoreBinding"
          name="fullScorePort">
      <corba:address="file://score.ref" />
    </port>
  </service>
  <service name="finalScoreSerice">
    <port binding="tns:finalScoreBinding"
          name="finalScorePort">
      <tuxedo:address serviceName="finalScoreServer" />
    </port>
  </service>
2 <routing:route name="scoreRoute">
    <routing:source service="tns:fullScoreService"
                    port="tns:fullScorePort"/>
    <routing:operation name="getFinalScore" target="getScore"/>
    <routing:destination service="tns:finalScoreService"
                         port="tns:finalScorePort"/>
  </routing:route>
```

There are two sections to the contract fragment shown in Example 88:

1. The logical interfaces must be bound to physical ports in `<service>` elements of the Artix contract.
2. The route, `scoreRoute`, is defined using the `<route:operation>` element.

You could also create a route between `getScore` in `baseballGamePortType` to a port bound to `baseballScorePortType`; see Example 85 on page 360.The resulting contract would include the fragment shown in Example 89.

Example 89: *Operation to Port Routing Example*

```
<service name="baseballGameService">
  <port binding="tns:baseballGameBinding"
        name="baseballGamePort">
    <soap:address location="http://localhost:8991"/>
  </port>
</service>
<service name="baseballScoreService">
  <port binding="tns:baseballScoreBinding"
        name="baseballScorePort">
    <iiop:address location="file:\\score.ref"/>
  </port>
</service>
<routing:route name="scoreRoute">
  <routing:source service="tns:baseballGameService"
                  port="tns:baseballGamePort"/>
  <routing:operation name="getScore"/>
  <routing:destination service="tns:baseballScoreService"
                       port="tns:baseballScorePort"/>
</routing:route>
```

Note that the `<routing:operation>` element only uses the name attribute. In this case the logical interface bound to `baseballScorePort`, `baseballScorePortType`, must contain an operation `getScore` that has matching messages as discussed in "Port-based routing" on page 359.

Advanced Routing Features

Overview

Artix routing also supports the following advanced routing capabilities:

- Broadcasting a message to a number of destinations.
- Specifying a failover service to route messages to provide a level of high-availability.
- Routing messages based on transport attributes in the received message's header.

Message broadcasting

Broadcasting a message with Artix is controlled by the routing rules in an Artix contract. Setting the `multiRoute` attribute to the `<routing:route>` element to `fanout` in your route definition allows you to specify multiple destinations in your route definition to which the source messages are broadcast.

There are three restrictions to using the fanout method of message broadcasting:

- All of the sources and destinations must be oneways. In other words, they cannot have any output messages.
- The sources and destinations cannot have any fault messages.
- The input messages of the sources and destinations must meet the compatibility requirements as described in "Compatibility of Ports and Operations" on page 359.

Example 90 shows an Artix contract fragment describing a route for broadcasting a message to a number of ports.

Example 90: *Fanout Broadcasting*

```
<message name="statusAlert">
  <part name="alertType" type="xsd:int"/>
  <part name="alertText" type="xsd:string"/>
</message>
<portType name="statusGenerator">
  <operation name="eventHappens">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
```

Example 90: *Fanout Broadcasting*

```
<portType name="statusChecker">
  <operation name="eventChecker">
    <input message="tns:statusAlert" name="statusAlert"/>
  </operation>
</portType>
<service name="statusGeneratorService">
  <port binding="tns:statusGeneratorBinding"
        name="statusGeneratorPort">
    <soap:address location="http:\\localhost:8081"/>
  </port>
</service>
<service name="statusCheckerService">
  <port binding="tns:statusCheckerBinding"
        name="statusCheckerPort1">
    <corba:address location="file:\\status1.ref"/>
  </port>
  <port binding="tns:statusCheckerBinding"
        name="statusCheckerPort2">
    <tuxedo:address serviceName="statusService"/>
  </port>
</service>
<routing:route name="statusBroadcast" multiRoute="fanout">
  <routing:source service="tns:statusGeneratorService"
                  port="tns:statusGeneratorPort"/>
  <routing:operation name="eventHappens" target="eventChecker"/>
  <routing:destination service="tns:statusCheckerService"
                       port="tns:statusCheckerPort1"/>
  <routing:destination service="tns:statusCheckerService"
                       port="tns:statusCheckerPort2"/>
</routing:route>
```

Failover routing

Artix failover routing is also specified using the `<routing:route>`'s `multiRoute` attribute. To define a failover route you set `multiRoute` to equal `failover`. When you designate a route as failover, the routed message's target is selected in the order that the destinations are listed in the route. If the first target in the list is unable to receive the message, it is routed to the second target. The route will traverse the destination list until either one of the target services can receive the message or the end of the list is reached.

Given the route shown in Example 91, the message will first be routed to `destinationPortA`. If service on `destinationPortA` cannot receive the message, it is routed to `destinationPortB`.

Example 91: *Failover Route*

```
<routing:route name="failoverRoute" multiRoute="failover">
  <routing:source service="tns:sourceService"
                  port="tns:sourcePort"/>
  <routing:destination service="tns:destinationServiceA"
                       port="tns:destinationPortA"/>
  <routing:destination service="tns:destinationServiceB"
                       port="tns:destinationPortB"/>
</routing:route>
```

Routing based on transport attributes

Artix allows you to specify routing rules based on the transport attributes set in a message's header when using HTTP or WebSphere MQ. Rules based on message header transport attributes are defined in `<routing:transportAttribute>` elements in the route definition. Transport attribute rules are defined after all of the operation-based routing rules and before any destinations are listed.

The criteria for determining if a message meets the transport attribute rule are specified in sub-elements to the `<routing:tranportAttribute>`. A message passes the rule if it meets each criterion specified in the listed sub-element.

Each sub-element has a `contextName` attribute to specify the context in which the attribute is defined and `contextAttributeName` attribute to specify the name of the attribute to be evaluated. The `contextName` attribute is specified using the QName of the context in which the attribute is defined. The two contexts shipped with Artix are described in Table 11.The `contextAttributeName` is also a QName and is relative to the context specified. For example, `UserName` is a valid attribute name for any of the HTTP contexts, but not for the MQ contexts.

Table 11: *Context QNames*

Context QName	Details
`http-conf:HTTPServerIncomingContexts`	Contains the attributes for HTTP messages being received by a server.

Table 11: *Context QNames*

Context QName	Details
`http-conf:HTTPServerOutgoingContexts`	Contains the attributes for HTTP messages being sent by a server.
`http-conf:HTTPClientIncomingContexts`	Contains the attributes for HTTP messages being received by a client.
`http-conf:HTTPClientOutgoingContexts`	Contains the attributes for HTTP messages being sent by a client.
`mq:MQConnectionAttributes`	Contains the attributes defining a connection to WebSphere MQ.
`mq:MQIncomingMessageAttributes`	Contains the attributes of WebSphere MQ messages being received by an Artix server.
`mq:MQOutoingMessageAttributes`	Contains the attributes of WebSphere MQ messages being sent by an Artix sent.

Most sub-elements have a `value` attribute that can be tested. Attributes dealing with string comparisons have an optional `ignorecase` attribute that can have the values `yes` or `no` (`no` is the default). Each of the sub-elements can occur zero or more times, in any order:

<routing:equals> applies to string or numeric attributes. For strings, the `ignorecase` attribute may be used.

<routing:greater> applies only to numeric attributes and tests whether the attribute is greater than the value.

<routing:less> applies only to numeric attributes and tests whether the attribute is less than the value.

<routing:startswith> applies to string attributes and tests whether the attribute starts with the specified value.

<routing:endswith> applies to string attributes and tests whether the attribute ends with the specified value.

<routing:contains> applies to string or list attributes. For strings, it tests whether the attribute contains the value. For lists, it tests whether the value is a member of the list. `contains` accepts an optional `ignorecase` attribute for both strings and lists.

<routing:empty> applies to string or list attributes. For lists, it tests whether the list is empty. For strings, it tests for an empty string.

<routing:nonempty> applies to string or list attributes. For lists, it passes if the list is not empty. For strings, it passes if the string is not empty.

For information on the transport attributes for HTTP see "Configuring HTTP Transport Attributes" on page 303. For information on the transport attributes for WebSphere MQ see "WebSphere MQ Artix Extensions" on page 485.

Example 92 shows a route using transport attribute rules based on HTTP header attributes. Only messages sent to the server whose `UserName` is equal to `JohnQ` will be passed through to the destination port.

Example 92: *Transport Attribute Rules*

```
<routing:route name="httpTransportRoute">
  <routing:source service="tns:httpService"
                  port="tns:httpPort"/>
  <routing:trasnportAttributes>
    <rotuing:equals
              contextName="http-conf:HTTPServerIncomingContexts"
              contextAttributeName="UserName"
              value="JohnQ"/>
  </routing:transportAttributes>
  <routing:destination service="tns:httpDest"
                       port="tns:httpDestPort"/>
</routing:route>
```

Error Handling

Initialization errors	Errors that can be detected during initialization while parsing the WSDL, such as routing between incompatible logical interfaces and some kinds of route ambiguity, are logged and an exception is raised. This exception aborts the initialization and shuts down the server.
Runtime errors	Errors that are detected at runtime are reported as exceptions and returned to the client; for example “no route” or “ambiguous routes”.

Service Lifecycles

Overview

When the Artix router uses dynamic proxy services, you can configure garbage collection of old proxies. Dynamic proxies are used when the router bridges services that have patterns such as callback, factory, or any interaction that passes references to other services. When the router encounters a reference in a message, it proxifies the reference into one that a receiving application can use. For example, an IOR from a CORBA server cannot be used by a SOAP client, so the router dynamically creates a new route for the SOAP client.

However, dynamic proxies persist in the router memory and can have a negative effect on performance. To overcome this, Artix provides service lifecycle garbage collection, which cleans up old proxy services that are no longer used. This garbage collection service cleans up unused proxies when a threshold has been reached on a least recently used basis.

Configuring service lifecycle

To configure service garbage collection for the Artix router, perform the following steps:

1. Add the `service_lifecycle` plug-in to the `orb_plugins` list:

```
orb_plugins = ["xmlfile_log_stream", "service_lifecycle",
   "routing"];
```

2. Configure the service lifecycle cache size:

```
plugins:service_lifecycle:max_cache_size = "30";
```

Writing client applications

When writing client applications, you must also make allowances for the garbage collection service; in particular, ensure that exceptions are handled appropriately.

For example, a client may attempt to proxify to a service that has already been garbage collected. To prevent this, do either of the following:

- Handle the exception, get a new reference, and continue. However, in some cases, this may not be possible if the service has state.
- Set `max_cache_size` to a reasonable limit to ensure that all your clients can be accommodated. For example, if you always expect to support 20 concurrent clients, each with a transient service session, you might wish to configure the `max_cache_size` to `30`.

You do not want to impact any clients, and must ensure that a service is no longer needed when it is garbage collected. However, if you set `max_cache_size` too high, this may use up too much router memory and have a negative impact on performance. For example, a suggested range for this setting is `30-100`.

Routing References to Transient Servants

Overview

Applications create transient servants by cloning a service defined in your contract. The cloned service uses the same interface, binding, and transport as the service defined in the contract. However, it has a unique QName and a unique address. So, when a transient servant's service definition only exists in the memory of the application that created it and possesses no link back to the service from which it was cloned.

Because a transient servant does not have a service definition in the physical contract and no link to one, the router, when it receives a reference to a transient servant, has no concrete information about how to create a proxy for the referenced servant. The router must make a best guess about which service in its contract to use as the template for the proxy to the transient servant. To do this, the router chooses the first compatible service definition in its contract.

Compatibility of services

A service is considered compatible with a transient servant if it uses the same interface, binding, and transport as the transient servant. For example, if transient servant was created using the `templateVendor` service defined in Example 93 it would be compatible with `IIOPVendor`. However, it would not be compatible with `SOAPVendor` because `SOAPVendor` uses a different transport than template. Also, if `IIOPVendor` was defined using different transport properties, such as having a defined POA name, transient servants created from `templateVendor` would not be compatible.

Example 93: *Contract with a Service Template*

```
<definitions ...>
  ...
  <message name="mangoRequest">
    <part name="num" type="xsd:int" />
  </message>
  <message name="mangoPrice">
    <part name="cost" type="xsd:float" />
  </message>
```

Example 93: *Contract with a Service Template*

```
  <portType name="fruitVendor">
    <operation name="sellMangos">
      <input name="num" message="tns:mangoRequest" />
      <output name="price" message="tns:mangoPrice" />
    </operation>
  </portType>
  <binding name="fruitVendorBinding" type="tns:fruitVendor">
    <soap:binding style="rpc"
                transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sellMangos">
      <soap:operation soapAction="" style="rpc"/>
      <input name="num">
        <soap:body
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://fruitVendor.com" use="encoded"/>
      </input>
      <output name="cost">
        <soap:body
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
        namespace="http://fruitVendor.com" use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="templateVendor">
    <port binding="tns:fruitVendorBinding"
          name="transientVendor">
      <iiop:address location="ior:" />
    </port>
  </service>
  <service name="SOAPVendor">
    <port binding="tns:fruitVendorBinding"
          name="SOAPVendorPort">
      <soap:address location="lcoalhost:5150" />
    </port>
  </service>
  <service name="IIOPVendor">
    <port binding="tns:fruitVendorBinding"
          name="IIOPVendorPort">
      <iiop:address location="file:///objref.ior" />
    </port>
  </service>
</definitions>
```

Contract design issues

The router's means of selecting a compatible service to create proxies for transient servants can result in odd behavior if you use the same interface to create both static servants and transient servants. When passing references to these services through the router, the potential exists for the router to select the static service to create proxies for the transient servants. WHen this happens, the router will silently redirect all of the messages to the servant defined by the static service definition.

To avoid this situation be sure to place the service templates used to create transient servants before the service definitions that will be used to create static servants. This will ensure that the router will find the service templates first when it proxifies a reference to a transient servant.

CHAPTER 17

Using the Artix Transformer to Solve Problems in Artix

The Artix Transformer allows you to perform message transformations, data validation, and interface versioning without having to write additional code.

In this chapter

This chapter discusses the following topics:

Using the Artix Transformer as an Artix Server

Overview

Using the Artix transformer, you can create a Web service that does simple tasks such as converting dates into the proper format or generating HTML output without writing any code. You can also develop services to validate the format of requests before they are sent to a busy server for processing.

The data processing is performed by the Artix transformer which uses an XSLT script to determine how to process the data.

Procedure

To use the Artix transformer as an Artix server you do the following:

1. Define the data, interface, binding, and transport details for the server in an Artix contract.
2. Write the XSLT script that defines the data processing you want the transformer to perform.
3. Configure the server with the transformer's configuration details.

Defining the server

The contract for a service that is implemented by the Artix Transformer is the same as the Artix contract for any other service in Artix. You need to define the complex types, if any, that the service uses. Then you need to define the messages used by the service to receive and respond to requests.

Once the data types and messages are defined, you then define the service's interface. The only limitation for a service that is implemented by the Artix Transformer is that it cannot have any fault messages. The interface can define multiple operations. Each operation will be processed using different XSLT scripts.

After defining the logical details of the service, you need to define the binding and network details for the service. The transformer can use any of the bindings and transports supported by Artix. For information on adding a binding for the transformer read "Binding Interfaces to a Payload Format" on page 227. For information on adding network details for the transformer read "Adding Transports" on page 297.

Writing the scripts

The XSLT scripts tell the transformer what it needs to do to process the data it receives. The scripts can be as simple or complex as they need to be to perform the task. The only requirement is that they are valid XSLT documents. For more information about writing XSLT scripts read "Writing XSLT Scripts" on page 392.

Configure the transformer

The Artix Transformer is an Artix plug-in and can be loaded by an Artix process. This provides a great deal of flexibility in how you configure and deploy the process. There are two common deployment patterns for deploying the Artix Transformer as an Artix server. The first is to configure the transformer to load in its own process using the Artix Standalone Service. The second is to configure the transformer to load directly into the client process which is making requests against it.

For a detailed discussion of how to configure and deploy the Artix Transformer see *Deploying and Managing Artix Solutions*.

Using Artix to Facilitate Interface Versioning

Overview

One of the most common and difficult problems faced in large scale client server deployments is upgrading systems. For example, if you change the interface for your server to add new functionality or streamline communications, you then need to change all of the clients that access the server. This can mean upgrading thousands of clients that may be scattered across the globe.

The Artix Transformer provides a solution to this problem that allows you to slowly upgrade the clients without disrupting their ability to function. Using the transformer you can develop an XSLT script that converts messages between the different interfaces. Then you can place the transformer between the old clients and the new server. This solution eliminates the need for operating two versions of the same server, or trying to do a massive client and server upgrade. It also does this without requiring you to do any custom programing.

Procedure

To use the Artix Transformer for interface versioning do the following:

1. Create a composite Artix contract defining both versions of the interfaces that need to be supported.
2. Define an interface for the transformer that defines operations for mapping the interfaces.
3. Add a SOAP binding to the contract for the transformer's interface.
4. Add an HTTP port to the contract to define how the transformer can be contacted.
5. Write the XSLT scripts that define the message transformations.
6. Configure the transformer.
7. Configure the Artix Chain Builder to create a chain containing the transformer and the server on which clients will make requests.

Creating a composite contract

While the server and the client applications can be run without knowledge of the other's interface, the transformer responsible for translating the messages between to the two interface versions must know about all of the interface versions used. This includes all data type definitions and message definitions used by both versions of the interface.

You can create this composite contract in several ways. The most straightforward way is to create a new contract which imports both the new interface's contract and the old interface's contract. To import the contracts you place an `<import>` element for each contract just after the `<definitions>` element in the new contract and before any other elements in the new contract. The `<import>` element has two attributes. `location` specifies the relative pathname of the file containing the contract that is being imported. `namespace` defines the XML namespace under which the imported contract can be referenced.

For example, if you were creating a composite contract for interface versioning you would have two contracts; one for the server with the updated interface and one for the client using the legacy interface. The file name for the server's contract is `r2e2.wsdl` and the contract for the client is `r2e1.wsdl`. For simplicity, they are located in the same directory as the composite contract. The composite contract importing both versions of the interface is shown in Example 94.

Example 94: *Composite WSDL*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="transformer"
  targetNamespace="http://www.widgets.com/transformer"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:r1="http://www.widgets.com/r2e2Server"
  xmlns:r2="http://www.widgets.com/r2e1Client"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.widgets.com/transformer"
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <import location="r2e2.wsdl"
          namespace="http://www.widgets.com/r2e2Server/>
  <import location="r2e1.wsdl"
          namespace="http://www.widgets.com/r2e1Client"/>
</definitions>
```

Note that in the <definitions> element of the contract, XML namespace shortcuts are defined for the imported contracts namespace. This makes using items defined in the imported contracts much easier.

Define the transformer's interface

Once you have imported all versions of the interface that you need to support into the transformer's composite contract, you need to define the transformer's interface. The transformer must have one operation defined for each transformation that is required to support all of the interface versions. For example, if you only changed the structure of the request message in when upgrading the server's interface, the transformer only needs one operation because the transformation is only one way. If you changed both the request and response messages, the transformer's interface will need two operations; one for the request message and one for the response.

The operation to transform a request from the client to the proper format for the server takes the client's message as its <input> element and the server's message as its <output> message. The operation to transform a response from the server to the proper format for a client takes the server's outgoing message as its <input> element and the client's incoming message as its <output> element.

Note: Fault messages are not supported.

When adding the operations, be sure to use the proper namespaces when referencing the messages for the different versions of the interface. Using the wrong namespaces could result in an invalid contract at the very least. If the contract is valid, and the namespaces are incorrect, your system will behave erratically.

For example, if the interface in Example 94 on page 385 was updated so that both the client's request and the server's response need to be transformed the transformer's interface would need two operations. In this

example the name of the request message is `widgetRequest` and the name of the response message is `widgetResponse`. The interface for the transformer, `versionTransform`, is shown in Example 95.

Example 95: *Versioning Interface*

```
<portType name="versionTransform">
  <operation name="requestTransform">
    <input name="oldRequest" message="r1:widgetRequest" />
    <output name="newRequest" message="r2:widgetRequest" />
  </operaiton>
  <operation name="responseTransform">
    <input name="newResponse" message="r2:widgetResponse" />
    <output name="oldReponse" message="r1:widgetResponse" />
  </operation>
</portType>
```

In the operation transforming the request, `requestTransform`, the input message is taken from the namespace `r1` which is the namespace under which the client's contract is imported. The output message is taken from `r2` which is the namespace under which the server's contract is imported. For the response message transformation, `responseTransform`, the order is reversed. The input message is from `r2` and the output message is from `r1`.

Defining the physical details for the transformer

After defining the operations used in transforming between the different version of the interface, you need to define the binding and network details for the transformer. The transformer can use any of the bindings and transports supported by Artix. For information on adding a binding for the transformer read "Binding Interfaces to a Payload Format" on page 227. For information on adding network details for the transformer read "Adding Transports" on page 297.

Writing the XSLT scripts

The XSLT scripts tell the transformer what it needs to do to process the data it receives. The scripts can be as simple or complex as they need to be to perform the task. The only requirement is that they are valid XSLT documents. For more information about writing XSLT scripts read "Writing XSLT Scripts" on page 392.

Configuring the transformer

The Artix Transformer is an Artix plugin and can be loaded by an Artix process. This provides a great deal of flexibility in how you configure and deploy the process. For a detailed discussion of how to configure and deploy the Artix Transformer see *Deploying and Managing Artix Solutions*.

Configuring a chain

When using the transformer to do interface versioning, you need to deploy it as part of a service chain. To build a service chain in Artix you deploy the Artix Chain Builder. Like the transformer, the chain builder is an Artix plugin and provides a number of deployment options. One way of deploying the chain builder along with the transformer is to deploy it alongside of the transformer in an instance of the Artix Standalone service.

For a detailed discussion of how to configure and deploy the Artix Chain Builder see *Deploying and Managing Artix Solutions*.

WSDL Messages and the Transformer

Overview

Because the Artix Transformer works on messages that can originate from any of the payload formats supported by Artix, the transformer changes the messages into an XML document based on the Artix contract describing the message before processing the data using the XSLT script. When the transformer is finished processing, it then takes the resulting XML document and changes it back into the appropriate payload format.

Because the transformer works on XML representations of the data relieves you of the burden of understanding how the data on the wire is represented. However, this fact also means that you must rely on the message descriptions in the Artix contract to guide how you write your XSLT scripts. In addition, it also requires that you are careful about producing valid results.

The incoming message

The XML document that the transformer processes is created by reading the message off the wire and using the Artix contract to reconstruct the XML document that the data represents. The reconstruction is done by looking at the `<input>` message from the appropriate `<operation>` in the `<portType>` that defines the invoked service. For example, if you had a service defined by the WSDL fragment in Example 96 and the transformer implemented the operation `configure` the XML document would be constructed using the `<input>` message for `configure`, `oldClientInput`.

Example 96: *WSDL Fragment for Transformer*

```
...
<message name="original">
  <part name="vehicle" type="xsd:string" />
  <part name="name" type="xsd:string" />
</message>
<message name="transformed">
  <part name="vehicle" type="xsd:string" />
  <part name="firstName" type="xsd:string" />
  <part name="lastName" type="xsd:string" />
</message>
...
```

Example 96: *WSDL Fragment for Transformer*

```
<portType name="parkingLotMeter">
  <operation name="configure">
    <input name="oldClientInput" message="original" />
    <output name="updatedInput" message="transformed" />
  </operation>
...
</portType>
...
```

When the message is reconstructed, the transformer uses the input message name, given in the `<input>` element, as the name of the root element of the XML document. It then uses the message definition and the schema types to recreate the data as an XML message. So if the transformer was using the contract defined in Example 96 on page 389 and received data where `vehicle` equaled `Prius` and `name` equaled `Old MacDonald` the input message processed by the transformer will look like Example 97.

Example 97: *Transformer Input Message*

```
<oldClientInput>
  <vehicle>
    Prius
  </vehicle>
  <name>
    Old MacDonald
  </name>
</oldClientInput>
```

Output message

The results from the transformer goes through the reverse of the process that turns the input message into an XML document. The transformer attempts to use the `<output>` message definition from the Artix contract to place the result message back onto the wire in the proper payload format. If the result message is not properly formed this attempt will fail, so you must be careful when writing your XSLT script to ensure that the results match the expected format.

When the result message is deconstructed, the transformer expects that the root element of the result has the name of the output message, as defined in the `<output>` element in the Artix contract. It then reads the message definition and associated type definitions from the contract to ensure that

the message is properly formed. For example, a result message for the configure operation defined in Example 96 on page 389 would look like Example 98.

Example 98: *Transformer Output Message*

```
<updatedInput>
  <vehicle>
    Prius
  </vehicle>
  <firstName>
    Old
  </firstName>
  <lastName>
    MacDonald
  </lastName>
</updatedInput>
```

Writing XSLT Scripts

Overview

XML Stylesheet Language Transformations(XSLT) is a language used to describe the transformation of XML documents. The current W3C standard for XSLT is 1.0 and can be read at the W3C web site (http://www.w3.org/TR/xslt). XSLT documents, called scripts, are well-formed XML documents that describe how a source XML document is transformed into a resulting XML document. It can be used to perform tasks as simple as splitting a name entry into first and last name entries and as complex as validating that a complex XML document matches the expectations of an interface described in a WSDL document.

Procedure

Writing an XSLT script can be done in a number of ways and using a number of tools. The steps given here assume that you are writing fairly simple scripts using a text editor.

To write a XSLT script you do the following:

1. Create an XML stylesheet with the required `<xsl:transform>` element.
2. Determine which elements in your source message need to be processed and create `<xsd:template>` elements for each of them.
3. For each element that has a matching template element, define how you want the element processed to produce a new output document.
4. If child elements need to be processed as part of processing a parent element, define a template for the child element and apply it as part of the parent element's template using `<xsd:apply-templates>`.

In this section

This section discusses the following topics:

Elements of an XSLT Script

Overview

An XSLT script is essentially an XML stylesheet containing a special set of elements that instruct an XSLT engine in the processing of other XML documents. An XSLT script must be defined in an `<xsl:transform>` element or an `<xsl:stylesheet>` element. In addition, it needs at least one valid top-level element to define the transformation.

The transform element

The `<xsl:transform>` element denotes that the document is an XML stylesheet. The `<xsl:stylesheet>` element can be used in place of the `<xsl:transform>` element. They are equivalent.

When creating an XSLT script you must set the version attribute to 1.0 to inform the transformer what version of XSLT you are using. In addition, you must provide an XML namespace shortcut for the XSLT namespace in the `<xsl:transform>` element. Example 99 shows a valid `<xsl:transform>` element for an XSLT script.

Example 99: *XSLT Script Stylesheet Element*

```
<xsl:transform version="1.0"
               xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
...
</stylesheet>
```

Top level elements

While all that is needed to make an XML document a valid XSLT script is the `<xsl:transform>` element, the `<xsl:transform>` element does not provide any instructions for processing data. The data processing instructions in an XSLT script are provided by a number of top-level XSLT elements. These element's include:

- `xsl:import`
- `xsl:include`
- `xsl:strip-space`
- `xsl:preserve-space`
- `xsl:output`
- `xsl:key`
- `xsl:decimal-format`

- xsl:namespace-alias
- xsl:attribute-set
- xsl:variable
- xsl:param
- xsl:template

An XSLT script can have any number and combination of top-level elements. Other than xsl:import, which must occur before any other elements, the top-level elements can be used in any order. However, be aware that the order determines the order in which processing steps happen.

Example

Example 100 shows a simple XSLT script that transforms <SSN> elements into <acctNum> elements.

Example 100: *Simple XSLT Script*

```
<xsl:transform version = '1.0'
               xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="SSN">
    <acctNum>
      <xsl:value-of select="."/>
    </acctNum>
  </xsl:template>
</xsl:stylesheet>
```

Using this XSLT script the transformer would change a message that contained <SSN>012457890</SSN> into a message that contained <acctNum>012457890</acctNum>.

XSLT Templates

Overview

XSLT processors use templates to determine the elements on which to apply a set of transformations. Documents are processed from the top element through their structure to determine if elements match a defined template. If a match is found, the rules specified by the template are applied.

To write a template in XSLT, do the following:

1. Create an `<xsl:template>` element.
2. Provide the path to the source element it processes.
3. Write the processing rules.

<xsl:template> elements

Templates are defined using `<xsl:template>` elements. These elements take one required attribute, `match`, which specifies the source element that triggers the rules. In addition, you can use the `name` attribute to give the template a unique identifier for referencing it elsewhere in the contract.

Specifying source elements

You specify the elements of the source document to which template rules are matched using the `match` attribute of the `<xsl:template>` element. The source elements are specified using the syntax specified by the XPath specification (http://www.w3.org/TR/xpath). The source element address looks very similar to a file path where slash(`/`) specifies the root element and child elements are listed in top down order separated by a slash(`/`). For example to specify the `<surname>` element of the XML document shown in Example 101, you would specify it as `/name/surname`.

Example 101:*Sample XML Document*

```
<name>
  <firstname>
    Joe
  </firstname>
  <surname>
    Friday
  </surname>
<name>
```

Template matching order

XSLT processors start processing with the `<xsl:template match="/">` element if it is present. All of the processing directives for this template act on the top-level elements of the source document. For example, given the XML document shown in Example 101 on page 395 any processing rules specified in `<xsl:template match="/">` would apply to the `<name>` element. In addition, specifying a template for the root element(`/`) forces you to make all your source element paths explicit from the root element. The XSLT script shown in Example 102 generates the string `Friday` when run on Example 101 on page 395.

Example 102: *XSLT Script with Root Element Template*

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <xsl:value-of select="/name/surname"/>
  </xsl:template>
</xsl:transform>
```

You do not need to specify a template for the root element of the source document in an XSLT script. When you omit the root element's template the processor treats all template paths as though they originated from the source documents top level element. The XSLT script in Example 103 generates the same output as the script in Example 102.

Example 103: *XSLT Script without Root Element Template*

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="surname">
    <xsl:value-of select="."/>
  </xsl:template>
</xsl:transform>
```

Template rules

The contents of an `<xsl:template>` element define how the source document is processed to produce an output document. You can use a combination of XSLT elements, HTML, and text to define the processing rules. Any plain text and HTML that are used in the processing rules are placed directly into the output document. For example, if you wanted to generate an HTML document from an XML document you would use an XSLT script that included HTML tags as part of its processing rules. The

script in Example 104 takes an XML document with a <title> element and a <subTitle> element and produces an HTML document where the contents of <title> are displayed using the <h1> style and the contents of <subTitle> are displayed using the <h2> style.

Example 104:*XSLT Template with HTML*

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <h1>
      <xsl:value-of select="//title"/>
    </h1>
    <h2>
      <xsl:value-of select="//subTitle"/>
    </h2>
  </xsl:template>
</xsl:transform>
```

Applying templates to child elements

You can instruct the XSLT processor to apply any templates defined in the script to the children of the element being processed using an <xsl:apply-templates> element as one of the rules in a template. <xsl:apply-templates> instructs the XSLT processor to treat the current element as a root element and run the templates in the script against it.

For example you could rewrite Example 104 as shown in Example 105 using <xsl:apply-templates> and defining a template for the <title> and <subTitle> elements.

Example 105:*XSLT Template Using apply-templates*

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>
  <xsl"template match="title">
    <h1>
      <xsl:value-of select="."/>
    </h1>
  </xsl:template>
```

Example 105: *XSLT Template Using apply-templates*

```
  <xsl"template match="subTitle">
    <h2>
      <xsl:value-of select="."/>
    </h2>
  </xsl:template>
</xsl:transform>
```

You can use the optional `select` attribute to limit the child elements to which the templates are applied. `select` takes an XPath value and operates in the same manner as the `match` attribute of `<xsl:template>`.

Example

For example, if your ordering system produced bills that looked similar to the XML document in Example 106, you could use an XSLT script to reformat the bill for a system that required the customer's name in a single element, name, and the city and state to be in a comma-separated field, city.

Example 106: *Bill XML Document*

```
<widgetBill>
  <customer>
    <firstName>
      Joe
    </firstName>
    <lastName>
      Cool
    </lastName>
  </customer>
  <address>
    <street>
      123 Main Street
    </street>
    <city>
      Hot Coffee
    </city>
    <state>
      MS
    </state>
    <zipCode>
      3942
    </zipCode>
  </address>
```

Example 106:*Bill XML Document*

```
  <amtDue>
    123.50
  </amtDue>
</widgetBill>
```

The XSLT script shown in Example 107 would result in the desired transformation.

Example 107:*XSLT Script for widgetBill*

```
<xsl:transform version = '1.0'
     xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
1   <xsl:template match="widgetBill">
      <xsl:element name="widgetBill">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:template>
2   <xsl:template match="customer">
      <xsl:element name="name">
        <xsl:value-of select="concat(//firstName,' ',//lastName)"/>
      </xsl:element>
    </xsl:template>
3   <xsl:template match="address">
      <xsl:element name="address">
        <xsl:copy-of select="//street"/>
        <xsl:element name="city">
          <xsl:value-of select="concat(//city,', ',//state)"/>
        </xsl:element>
        <xsl:copy-of select="//zipCode"/>
      </xsl:element>
    </xsl:template>
4   <xsl:template match="amtDue">
      <xsl:copy-of select="."/>
    </xsl:template>
</xsl:transform>
```

The script does the following:

1. Creates an element, `<widgetBill>`, in the output document and places the results of the other templates as its children.
2. Creates an element, `<name>`, and sets its value to the result of the concatenation.

3. Creates an element, `<address>`, and sets its value to the results of the rules. `<address>` will contain a copy of the `<street>` element from the source document, a new element, `<city>`, that is a concatenation, and a copy of the `<zipCode>` element from the source document.
4. Copy the `<amtDue>` element from the source document into the output document.

Processing the document in Example 106 on page 398 with this XSLT script would result in the XML document shown in Example 108.

Example 108: *Processed Bill XML Document*

```
<widgetBill>
  <customer>
    Joe Cool
  </customer>
  <address>
    <street>
      123 Main Street
    </street>
    <city>
      Hot Coffee, MS
    </city>
    <zipCode>
      3942
    </zipCode>
  </address>
  <amtDue>
    123.50
  </amtDue>
</widgetBill>
```

Common XSLT Functions

Overview

XSLT provides a range of capabilities in processing XML documents. These include conditional statements, looping, creating variables, and sorting. However, there are a few common functions that are used to generate output documents. These include:

- xsl:value-of
- xsl:copy-of
- xsl:element

xsl:value-of

`<xsl:value-of>` creates a text node in the ouput document. It has a required `select` attribute that specifies the text to be inserted into the output document.

The value of `select` is evaluated as an expression describing the data to insert. It can contain any of the XSLT string functions, such as `concat()`, or an XSLT axis describing an element in the source document.

Once the `select` expression is evaluated the result is placed in the output document.

xsl:copy-of

`<xsl:copy-of>` copies data from the source document into the output document. It has a required `select`. The value of `select` is an expression describing the elements to be copied.

When the result of evaluating the expression is a tree fragment, the complete fragment is copied into the output document. When the result is an element, the element, its attributes, its namespaces , and its children are copied into the output document. When the result is neither an element nor a result tree fragment, the result is converted to a string and then inserted into the output document.

xsl:element

`<xsl:element>` creates an element in the output document. It takes a required `name` attribute that specifies the name of the element that is created. In addtion, you can specify a `namespace` for the element using the optional namespace attribute.

Part III

Appendecies

In this part

This part contains the following apendecies:

APPENDIX A

Use Case Examples

Two use cases have been provided to walk you through the Artix Designer, and give you an introduction to the different ways you can perform common tasks.

In this appendix

This appendix discusses the following topics:

Create a Web Service Client Using a Template

Overview

This use case walks you through the procedure for creating a Web Service Client using a template-based method. Artix applies defaults for almost every variable, thus making this the quickest way to get your Web Service Client up and running with almost no input from you.

Before you begin

Before starting this procedure, you need:

- Artix installed on your local machine
- A WSDL document (or a URL address) that describes the target service
- A target SOAP/HTTP service to test your client against

Procedure

1. Start Artix from either the icon on your desktop or the Start menu, to display the Welcome dialog, as shown in Figure 139.

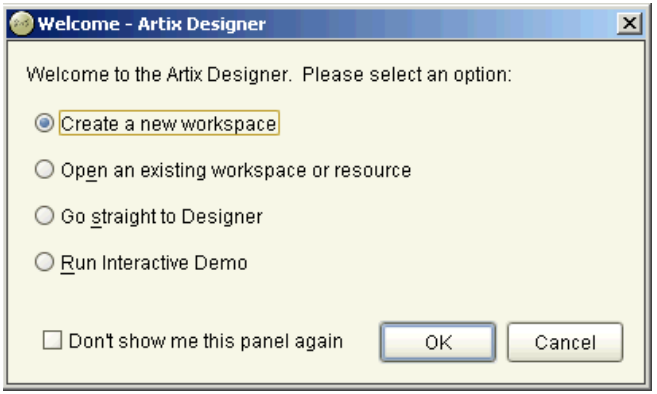

Figure 139: *Welcome dialog*

2. Select **Create a New Workspace** and click **OK** to display the New Workspace dialog, as shown in Figure 140.

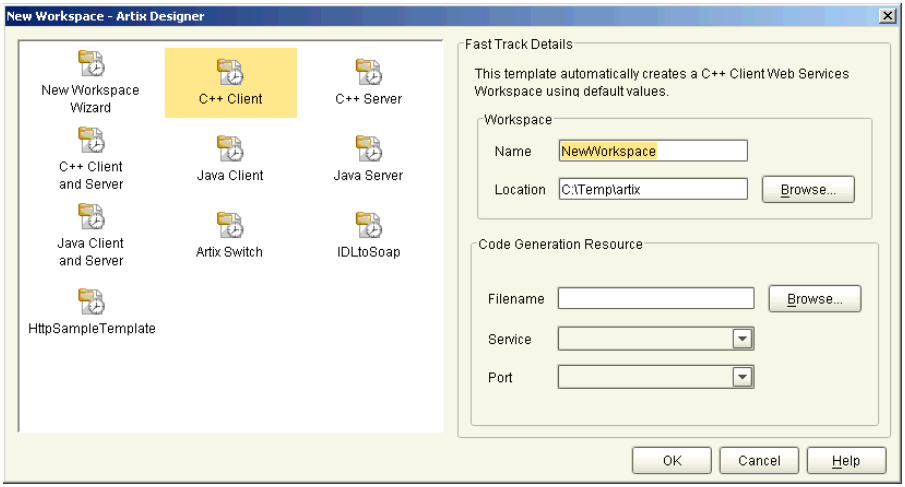

Figure 140:*New Workspace dialog*

3. Select the **C++ Client template** icon.
4. Enter a name and save location for your workspace, or accept the defaults provided. Click **Browse** to navigate to a specific location if you like.
5. Enter the file name or URL for your WSDL file in the field provided, or click **Browse** to navigate to a suitable file.
6. Click **OK** to display the Artix Designer with your Web Service Client contained in the Designer Tree.

Behind the scenes

Behind the scenes, Artix has performed the following tasks:

- Created a project directory and project file in the save location you specified
- Imported your WSDL file and added it to the project file
- Created a deployment profile configured for C++ deployment of your client

Your Web Service Client is ready for code generation.

Deploying the client

Now that Artix has automatically created the required deployment profile information, deploying your Web Service Client involves two tasks:

- Create a deployment bundle
- Generate the client code

To create a deployment bundle:

1. Select the collection name (C++ Client) in the Designer Tree to display the Collection Details panel, as shown in Figure 141.

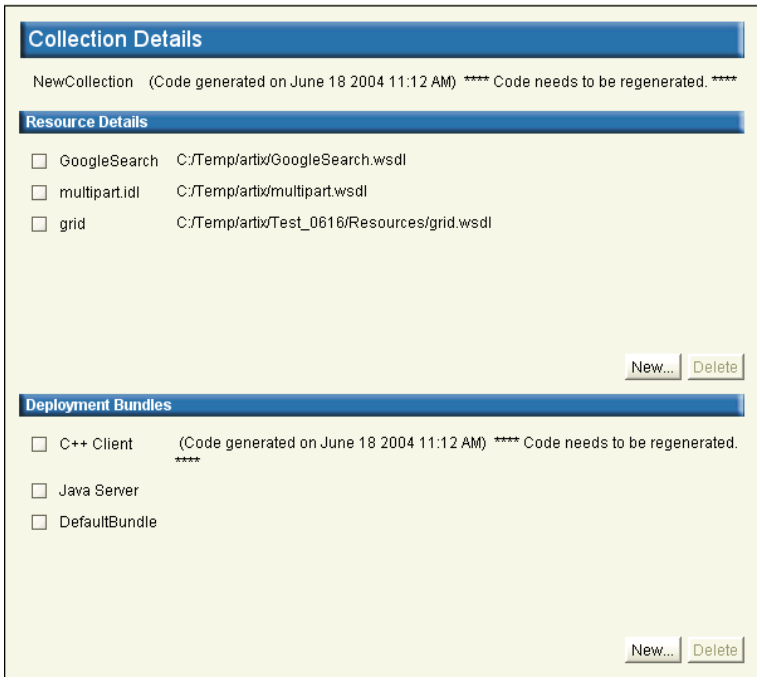

Figure 141:*Collection Details panel*

2. Click the **Add** button under the **Deployment Bundles** section to display the New Deployment Bundle wizard.

3. Move through the wizard, clicking **Next** on every panel to accept the system defaults. For more information about this process, see "Editing a Deployment Profile" on page 183.
4. Click **Finish** to exit the last panel and return to the Collection Details panel, where your new bundle is now listed.

To generate your client code:

1. Select the collection name (C++ Client) in the Designer Tree, and select **Tools | Generate Code** from the menu bar to display the Generate Code dialog, as shown in Figure 142.

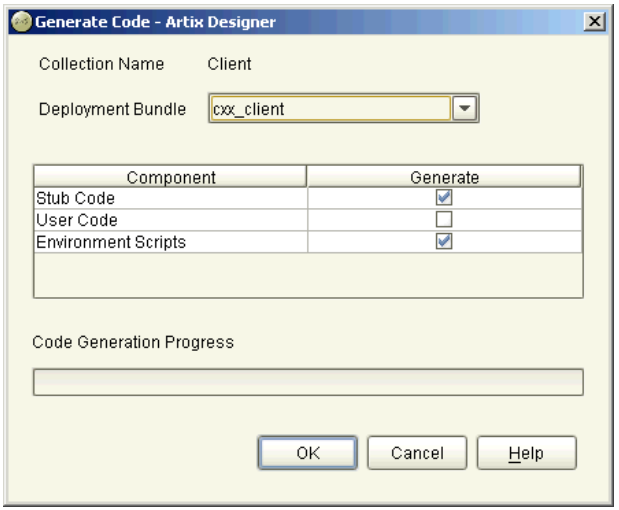

Figure 142:*Generate Code dialog*

2. Click **OK** to generate the client code.

   You will receive a confirmation when the process is complete (usually 3-4 seconds).

Create a Web Service Server Using a Wizard

Overview

This use case walks you through the procedure for creating a Web Service server. Unlike the template method described in the previous use case ("Create a Web Service Client Using a Template" on page 406), this use case walks you through the process of providing the required input for the server via the New Workspace wizard.

Before you begin

Before starting this procedure, you need:

- Artix installed on your local machine
- A WSDL document (or a URL address) that describes the target service
- A target SOAP/HTTP service to test your client against

Creating the WS Server

1. From the Welcome dialog, select **Create a new Workspace** and click **OK** to display the New Workspace dialog, as shown in "New Workspace dialog" on page 410.

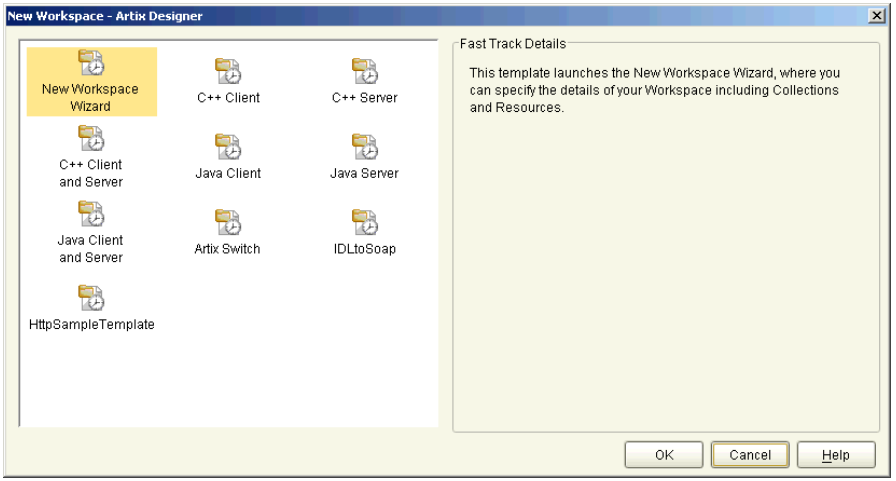

Figure 143:*New Workspace dialog*

2. Select the **New Workspace Wizard** icon, to display the New Workspace wizard, as shown in Figure 144

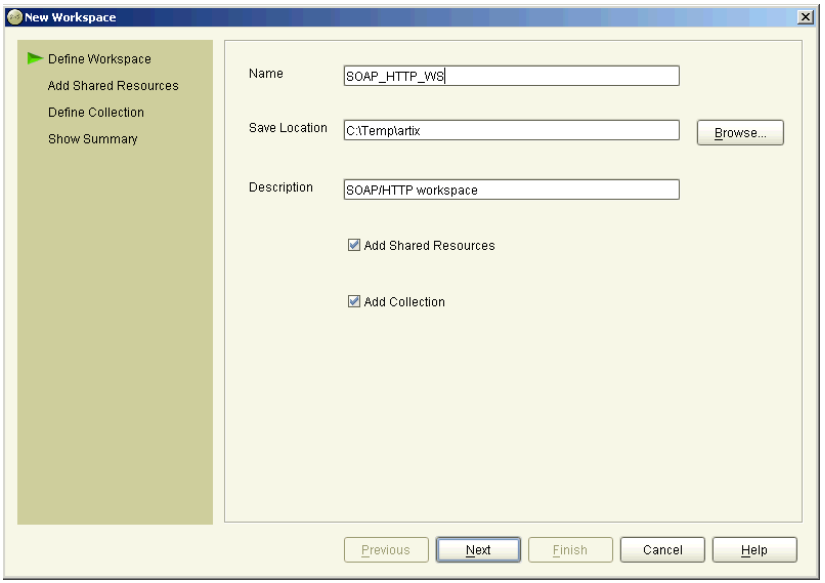

Figure 144:*New Workspace Wizard*

3. Enter a name for the workspace, or accept the default provided.
4. Select the location where you would like to save your workspace, or accept the default provided.

   Tip: To define a new default save location for all future workspaces, go to the User Preferences dialog (under the Edit menu).
5. Add a description for this workspace in the field provided.
6. Select the **Add Shared Resources** check box if you want to add resources to this workspace that will be shared between all the collections in the workspace.

   Selecting this option will add an extra panel to the wizard for you to enter the shared resource details.

7. Select the **Add Collection** check box if you want to add a collection to this workspace now. Note that this is optional - you can always add a collection later if you don't want to add one now.

   Selecting this option will add an extra panel to the wizard for you to enter the collection details.

8. Click **Next** to display one of the following panels, depending on which check boxes you selected on the first panel:

   - If you checked the Add Shared Resources option, the Shared Resources panel is displayed, as shown in Figure 145. Continue with **step 8.**

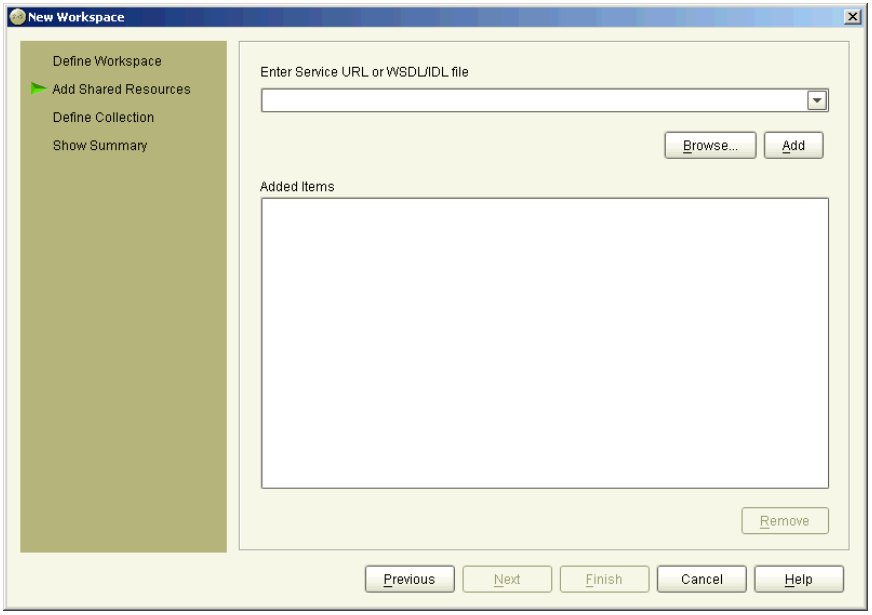

Figure 145:*New Workspace wizard—Shared Resources panel*

   - If you did not check the Add Resources option but did check the Add Collection option, the Define Collection panel is displayed, as shown in Figure 146. Continue with **step 10**.

- If you did not check either of the options on the first panel, the Summary panel is displayed as shown in Figure 147 on page 414. Continue with **step 14**.

9. Type the location of either a WSDL file or an IDL file in the Enter Service URL or WSDL/IDL file field, or click **Browse** to navigate to the file you would like to use.

   When you have selected a file to use, click **Add** to list it in the Added Items list.

10. Repeat **step 8** as many times as you like to continue adding resources to the list, then click **Next** to display Define Collection panel as shown in Figure 146. If you did not choose to Add a Collection, go to **step 14.**

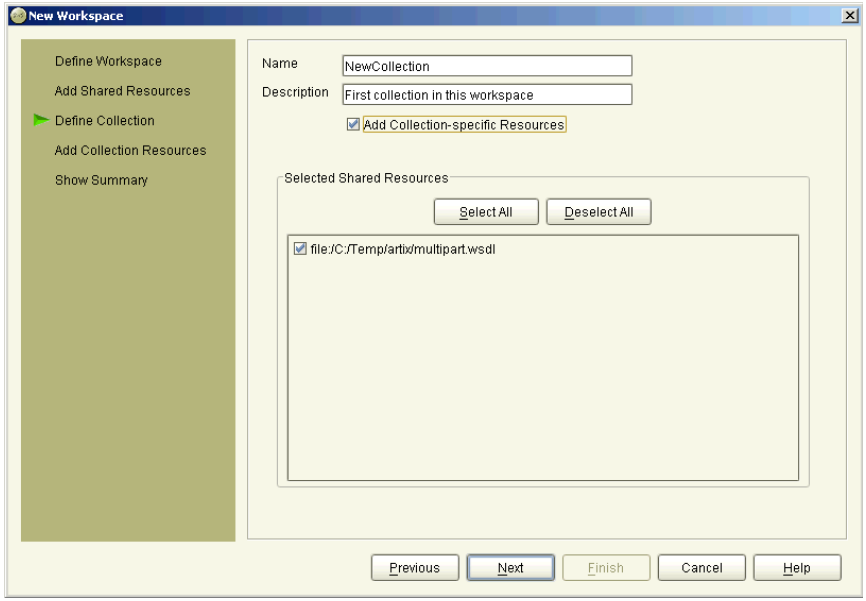

Figure 146:*New Workspace wizard—Define Collection panel*

11. Enter a name for the new collection, or accept the default provided.
12. Enter a description for the new collection in the Description field.

13. By default, all shared resources you added to this workspace on the previous panel are selected to be added to this collection. If there are any resources you do not want added, click on their check box to deselect them.
14. Click **Next** to display the display the Summary panel, as shown in Figure 147. This panel lists everything you just specified in the wizard.

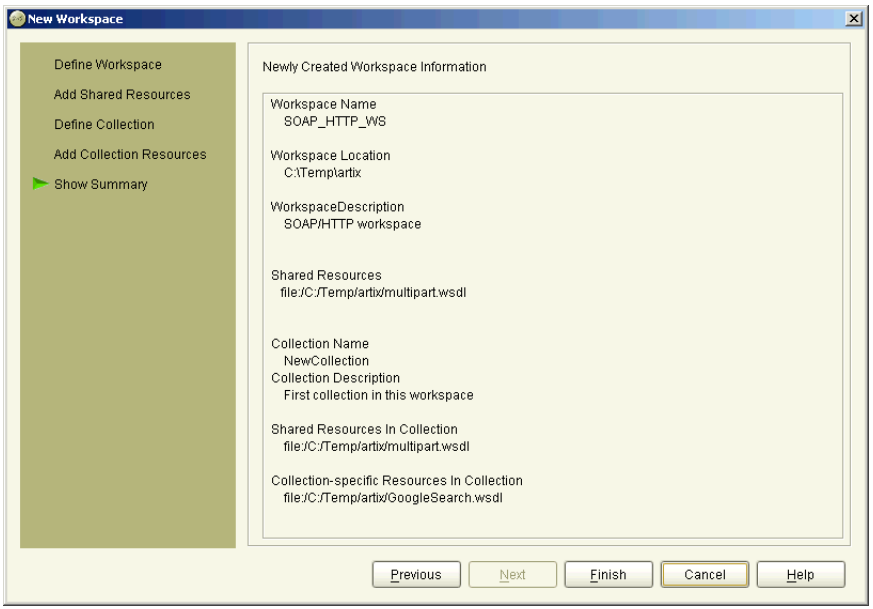

Figure 147:*New Workspace wizard—Summary panel.*

15. Click **Finish** to close the wizard and display the Artix Designer, where the Designer Tree displays your newly created workspace.

Deploying the server

Now that you have created your workspace, deploying your Web Service Server involves three tasks:

- Create a deployment profile - this contains machine-specific information that you can use multiple times to deploy as many collections as you have in your workspace. For each machine operating system, however, you would need a separate deployment profile. Turn to "Creating a Deployment Profile" on page 179 for help with this task.
- Create a deployment bundle - this defines the type of deployment you want to perform, such as a client, server, or switch. Thus, you can create a deployment profile, then deploy the same collection as a client and/or a server, and/or a switch just by creating separate deployment bundles. Turn to "Editing a Deployment Profile" on page 183 for help with this task.
- Generate the code - a very simple (one-dialog) task once the profile and bundle have been created. Turn to "Generating Code" on page 192 for help with this task.

Expose a CORBA Server as a Web Service

Overview

This use case walks you through the procedure for exposing a CORBA Server as a Web Service using a template-based method. Artix applies defaults for almost every variable, thus making this virtually a one-click process.

Before you begin

Before starting this procedure, you need:

- Artix installed on your local machine
- An IDL and an IOR file for the CORBA server
- A SOAP address to where you want to expose the CORBA server

Procedure

1. Start Artix from either the icon on your desktop or the Start menu, to display the Welcome dialog, as shown in Figure 139.

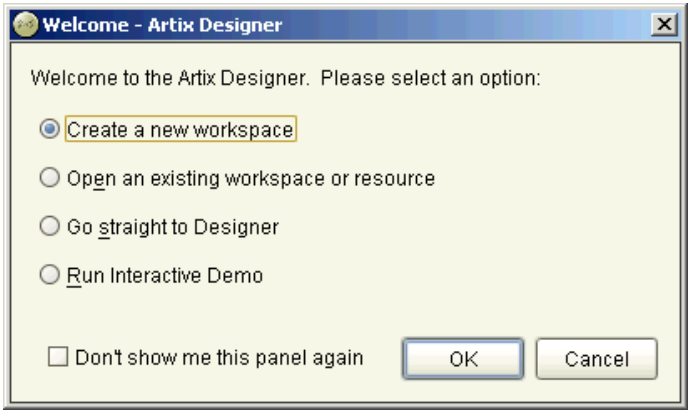

Figure 148: *Welcome dialog*

2. Select **Create a New Workspace** and click **OK** to display the New Workspace dialog, as shown in Figure 140.

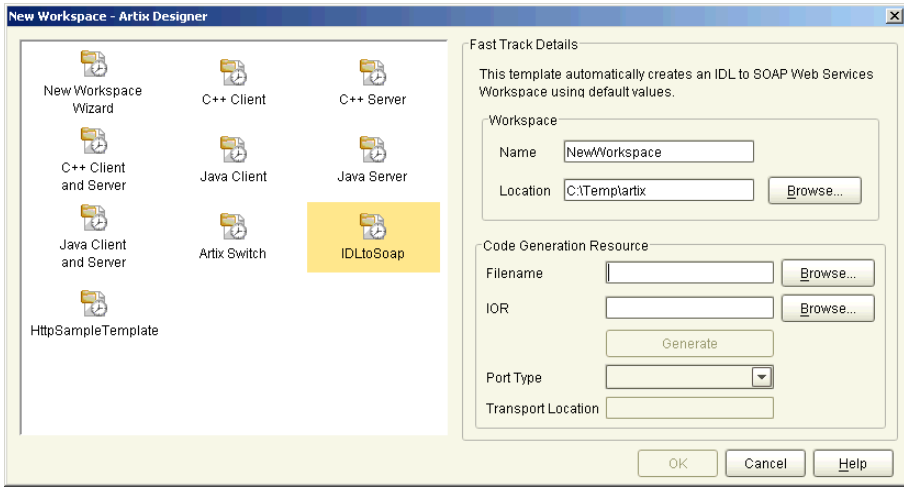

Figure 149:*New Workspace dialog*

3. Select the **IDLtoSOAP template** icon.
4. Enter a name and save location for your workspace, or accept the defaults provided. Click **Browse** to navigate to a specific location if you like.
5. Enter the file name of an IDL file in the field provided, or click **Browse** to navigate to a suitable file.
6. Enter the file name of an IOR file in the field provided, or click **Browse** to navigate to a suitable file. The IOR file defines the address of the CORBA server.
7. Click **Generate** to create a WSDL file based on the IDL and IOR.
8. Select a Port Type from the drop-down list provided. The port types list is populated as a result of the WSDL generation process.
9. Lastly, specify a SOAP address in the Transport Location field. This is the address from which you will access your Web Service.

10. Click **OK** to display the Artix Designer with your Web Service contained in the Designer Tree, as shown in Figure 150.

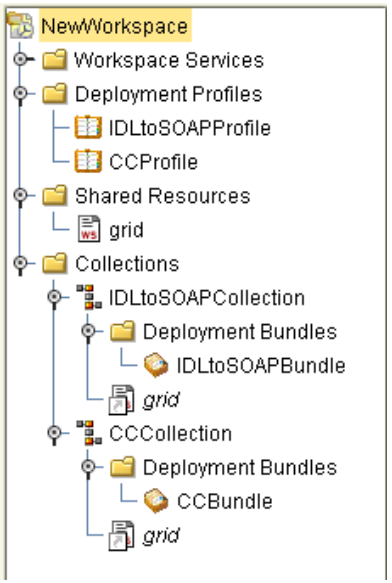

Figure 150:*Artix Designer with CORBA Server exposed as Web Service*

Behind the scenes

Behind the scenes, Artix has performed the following tasks:

- Created a workspace directory and file in the save location you specified
- Generated a WSDL file based on your IDL and IOR files and added it to the workspace file
- Created two deployment profiles configured for C++ deployment of your client
- Created a deployment bundle configured for C++ deployment of your client

Your Web Service Client is ready for code generation.

Generating the client code

Now that Artix has automatically created the required deployment profile and bundle information, generating the code for your Web Service is very simple:

1. Select the collection name (C++ Client) in the Designer Tree, and select **Tools | Generate Code** from the menu bar to display the Generate Code dialog, as shown in Figure 142.

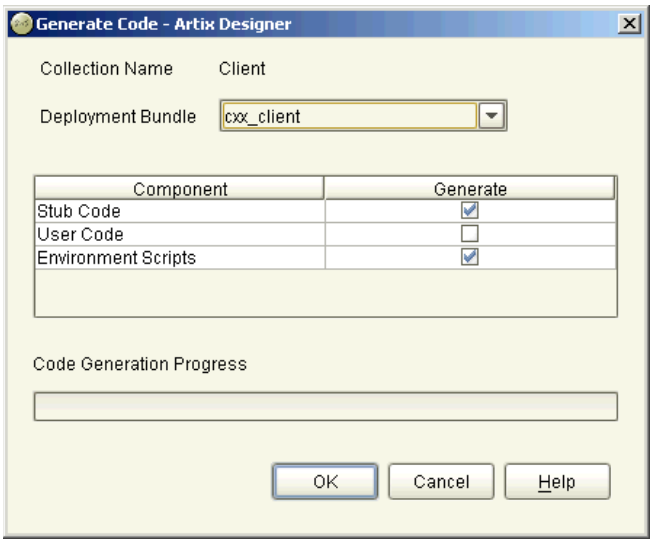

Figure 151:*Generate Code dialog*

2. Click **OK** to generate the client code.

   You will receive a confirmation when the process is complete (usually 3-4 seconds).

APPENDIX B

Command Line Use Case Examples

Two use cases have been provided to walk you through using the Artix command line tools.

In this appendix

This appendix discusses the following topics:

Create a C++ Web Service Client from a WSDL Contract

Overview

This use case walks you through the procedure for creating a C++ Web service client from an existing WSDL contract. Artix will generate all of the code needed to develop you Web service client and deploy it using the default Artix configuration.

Before you begin

Before starting this procedure, you need:

- Artix installed on your local machine
- A supported C++ compiler
- A WSDL contract that describes the target service
- A target SOAP/HTTP service to test your client against

Procedure

To create a C++ Web service client from a WSDL contract do the following:

1. From a command line prompt, change to your Artix installation's `bin` directory.

   ```
   install_dir/artix/2.1/bin
   ```

2. Run `artix_env`.
3. Change to the directory where the WSDL document describing the target service is located.
4. Run the Artix C++ code generator on you WSDL file and provide the flag to generate a sample client.

   ```
   wsdltocpp -client wsdlfile
   ```

5. Build the generated code.
6. Run the client.

Creating a C++ SOAP/HTTP Web Service from IDL

Overview

This use case walks you through the procedure for building a Web service from a CORBA IDL interface using Artix command line tools. These steps can be automated using a number of scripting languages and integrated into your build system. For more detailed information about the command line tools used and a complete listing of their options, see the *Artix Command Line Tools Reference*.

Before you begin

Before starting this procedure, you need:

- Artix installed on your local machine
- An IDL file describing the service
- A CORBA client to test your server against

Procedure

To create a Web service from an IDL file using Artix do the following:

1. From a command line prompt, change to your Artix installation's `bin` directory.

   ```
   install_dir/artix/2.1/bin
   ```

2. Run `artix_env`.
3. Change to the directory where the target IDL file is located.
4. To generate a WSDL contract containing the logical details of the service described by the IDL and a CORBA binding, run the IDL file through `idltowsdl` using the command shown below:

   ```
   idltowsdl idlfile
   ```

   This will generate a WSDL contract with a default target namespace and a default schema namespace determined from the name of the IDL file name. To set the target namespace use the `-w` flag and to set the schema namespace use the `-x` flag. For more details read "Creating Artix Contracts from CORBA IDL" on page 338.

5. To add a SOAP binding to the newly generated WSDL contract run the `wsdltosoap` tool as shown below:

   ```
   wsdltosoap wsdlfile
   ```

   This will generate a new WSDL contract, *`wsdlfile`*`-soap.wsdl`, that contains the original contract plus a default `doc/literal` SOAP binding for the logical interface. You can change the style of the SOAP binding using the `-style` flag and you can change the encoding of the SOAP binding using the `-use` flag. For more details see "Adding a Default SOAP Binding" on page 230.
6. To add an HTTP port to the contract with the SOAP binding, open it in your favorite text or XML editor.
7. After the SOAP binding in the contract, add a `<service>` element and give it a unique name.
8. Add a `<port>` element to the new `<service>` element and give the `<port>` element a unique name.
9. Add a `<soap:address>` element to the `<port>` element and enter the URL address where your Web service will be deployed in the `location` attribute.

   Example 109 shows a completed `<service>` element for the new Web service. You can also add a number of optional HTTP property elements to the port specification to define the behavior of the HTTP port. For more details see "Configuring HTTP Transport Attributes" on page 303.

Example 109:*Port*

```
<service name="newHTTPservice">
  <port name="newHTTPport">
    <soap:address location="http:\\localhost:9000" />
  </port>
</service>
```

10. To generate the C++ stub and skeleton code for the Web service, run the new contract through `wsdltocpp` as shown below.

    ```
    wsdltocpp wsdlfile-soap.wsdl
    ```

`wsdltoccp` has a number of flags that allow you to specify the C++ namespaces of used in the generated code among other options. For a detailed discussion of all of the options offered by `wsdltocpp` see *Developing Artix Applications in C++*.

11. Develop your application logic for the service and its clients in your favorite C++ development environment.
12. Build your application using your favorite C++ complier.
13. Run the applications.

    Most Artix applications will run using the default configuration supplied with Artix. For information on modifying your Artix configuration see *Deploying and Managing Artix Solutions*.

APPENDIX C

SOAP Binding Extensions

SOAP is an XML-based message specification by the W3C and is widely accepted as the de facto format for communicating over the Web.

Overview

This appendix describes each of the configuration attributes that can be set up as part of the WSDL extensions for configuring the SOAP message format plug-in for use with Artix. It discusses the following topics:

soap:binding element

Overview

The `soap:binding` element in a WSDL contract is defined within the `<binding>` component, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
```

Only one `soap:binding` element is defined in a WSDL contract. It is used to signify that SOAP is the message format being used for the binding.

Attributes

Table 12 describes the attributes defined within the `soap:binding` element.

Table 12: *Attributes for soap:binding*

Configuration Attribute	Explanation
`style`	The value of the `style` attribute within the `soap:binding` element acts as the default for the `style` attribute within each `soap:operation` element. It indicates whether request/response operations within this binding are RPC-based (that is, messages contain parameters and return values) or document-based (that is, messages contain one or more documents). Valid values are `rpc` and `document`. The specified value determines how the SOAP Body within a SOAP message is structured.

Table 12: *Attributes for soap:binding*

Configuration Attribute	Explanation

	If `rpc` is specified, each message part within the SOAP Body is a parameter or return value and will appear inside a wrapper element within the SOAP Body. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:body namespace` attribute. The message parts within the wrapper element correspond to operation parameters and must appear in the same order as the parameters in the operation. Each part name must match the parameter name to which it corresponds.
	For example, the SOAP Body of a SOAP request message is as follows if the style is RPC-based:
	`<SOAP-ENV:Body>`
	`  <m:GetStudentGrade xmlns:m="URL">`
	`  <StudentCode>815637</StudentCode>`
	`  <Subject>History</Subject>`
	`  </m:GetStudentGrade>`
	`</SOAP-ENV:Envelope>`
	If `document` is specified, message parts within the SOAP Body appear directly under the SOAP Body element as body entries and do not appear inside a wrapper element that corresponds to an operation. For example, the SOAP Body of a SOAP request message is as follows if the style is document-based:
	`<SOAP-ENV:Body>`
	`  <StudentCode>815637</StudentCode>`
	`  <Subject>History</Subject>`
	`</SOAP-ENV:Envelope>`

`transport`	This defaults to the URL that corresponds to the HTTP binding in the W3C SOAP specification (http://schemas.xmlsoap.org/soap/http). If you want to use another transport (for example, SMTP), modify this value as appropriate for the transport you want to use.

soap:operation element

Overview

A `soap:operation` element in a WSDL contract is defined within an `<operation>` component, which is defined in turn within the `<binding>` component, as follows:

```
<binding name="…" type="…" >
    <soap:binding style="…" transport="…">
    <operation name="…" >
        <soap:operation style="…" soapAction="…">
```

A `soap:operation` element is used to encompass information for an operation as a whole, in terms of input criteria, output criteria, and fault information.

Attributes

Table 13 describes the attributes defined within a `soap:operation` element.

Table 13: *Attributes for soap:operation*

Configuration Attribute	Explanation
`style`	This indicates whether the relevant operation is RPC-based (that is, messages contain parameters and return values) or document-based (that is, messages contain one or more documents). Valid values are `rpc` and `document`. See "soap:binding element" on page 428 for more details of the style attribute. The default value for `soap:operation style` is based on the value specified for the `soap:binding style` attribute.
`soapAction`	This specifies the value of the `SOAPAction` HTTP header field for the relevant operation. The value must take the form of the absolute URI that is to be used to specify the intent of the SOAP message. **Note:** This attribute is mandatory only if you want to use SOAP over HTTP. Leave it blank if you want to use SOAP over any other transport.

soap:body element

Overview

A <soap:body> element in a binding is a child of the <input>, <output>, and <fault> elements of the WSDL <operation> element, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
    <operation name="…">
        <soap:operation style="…" soapAction="…">
        <input>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
        </input>
        <output>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
        </output>
        <fault>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
        </fault>
    </operation>
```

A <soap:body> element is used to provide information on how message parts are to be appear inside the body of a SOAP message. As explained in "soap:operation element" on page 430, the structure of the SOAP Body within a SOAP message is dependent on the setting of the soap:operation style attribute.

Attributes

Table 14 describes the attributes defined within the soap:body element.

Table 14: *Attributes for soap:body*

Configuration Attribute	Explanation
`use`	This attribute indicates how message parts are used to denote data types. Each message part relates to a particular data type that in turn might relate to an abstract type definition or a concrete schema definition. An abstract type definition is a type that is defined in some remote encoding schema whose location is referenced in the WSDL contract via an `encodingStyle` attribute. In this case, types are serialized based on the set of rules defined by the specified encoding style. A concrete schema definition relates to types that are defined in the WSDL contract itself, within a `<schema>` element within the `<types>` component of the contract. Valid values for `soap:body use` are `encoded` and `literal`.

Table 14: *Attributes for soap:body*

Configuration Attribute	Explanation
	If `encoded` is specified, the `type` attribute that is specified for each message part (within the `<message>` component of the WSDL contract) is used to reference an abstract type defined in some remote encoding schema. In this case, a concrete SOAP message is produced by applying encoding rules to the abstract types. The encoding rules are based on the encoding style identified in the `soap:body encodingStyle` attribute. The encoding takes as input the `name` and `type` attribute for each message part (defined in the `<message>` component of the WSDL contract). If the encoding style allows variation in the message format for a given set of abstract types, the receiver of the message must ensure they can understand all the format variations. If `literal` is specified, either the `element` or `type` attribute that is specified for each message part (within the `<message>` component of the WSDL contract) is used to reference a concrete schema definition (defined within the `<types>` component of the WSDL contract). If the `element` attribute is used to reference a concrete schema definition, the referenced element in the SOAP message appears directly under the SOAP Body element (if the operation style is document-based) or under a part accessor element that has the same name as the message part (if the operation style is RPC-based). If the `type` attribute is used to reference a concrete schema definition, the referenced type in the SOAP message becomes the schema type of the SOAP Body (if the operation style is documented-based) or of the part accessor element (if the operation style is document-based). The `use` attribute is mandatory.
`encodingStyle`	This attribute is used when the `soap:body use` attribute is set to `encoded`. It specifies a list of URIs (each separated by a space) that represent encoding styles that are to be used within the SOAP message. The URIs should be listed in order, from the most restrictive encoding to the least restrictive. This attribute can also be used when the `soap:body use` attribute is set to `literal`, to indicate that a particular encoding was used to derive the concrete format, but that only the specified variation is supported. In this case, the sender of the SOAP message must conform exactly to the specified schema.

Table 14: *Attributes for soap:body*

Configuration Attribute	Explanation
`namespace`	If the `soap:operation style` attribute is set to `rpc`, each message part within the SOAP Body of a SOAP message is a parameter or return value and will appear inside a wrapper element within the SOAP Body. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:body namespace` attribute.
`parts`	This attribute is a space separated list of parts from the parent `<input>`, `<output>`, or `<fault>` element. When parts is set, only the specified parts of the message are included in the SOAP body. The unlisted parts are not transmitted unless they are placed into the SOAP header.

soap:header element

Overview

A <soap:header> element in a binding is an optional child of the <input>, <output>, and <fault> elements of the WSDL <operation> element, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
    <operation name="…">
        <soap:operation style="…" soapAction="…">
        <input>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
            <soap:header message="..." part="..." use="..."
                         encodingStyle="..." namespace="..."/>
        </input>
        <output>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
            <soap:header message="..." part="..." use="..."
                         encodingStyle="..." namespace="..."/>
        </output>
        <fault>
            <soap:body use="…" encodingStyle="…"
                  namespace="…" parts="..."/>
            <soap:header message="..." part="..." use="..."
                         encodingStyle="..." namespace="..."/>
        </fault>
    </operation>
```

A <soap:header> element defines the information that is placed in a SOAP header element. You can define any number of <soap:header> elements for an operation. As explained in "soap:operation element" on page 430, the structure of the SOAP header within a SOAP message is dependent on the setting of the soap:operation style attribute.

Attributes

Table 15 describes the attributes defined within the soap:body element.

Table 15: *Attributes for soap:header*

Configuration Attribute	Explanation
`message`	This attribute specifies the qualified name of the message from which the contents of the SOAP header is taken.
`part`	This attribute specifies the name of the message part that is placed into the SOAP header.
`use`	This attribute is used in the same way as the `use` attribute within the `soap:body` element. See "use" on page 432 for more details.
`encodingStyle`	This attribute is used in the same way as the `encodingStyle` attribute within the `soap:body` element. See "encodingStyle" on page 433 for more details.
`namespace`	If the `soap:operation style` attribute is set to `rpc`, each message part within the SOAP header of a SOAP message is a parameter or return value and will appear inside a wrapper element within the SOAP header. The name of the wrapper element must match the operation name. The namespace of the wrapper element is based on the value of the `soap:header namespace` attribute.

soap:fault element

Overview

A `soap:fault` element in a WSDL contract is defined within the `<fault>` component within an `<operation>` component, as follows:

```
<binding name="…" type="…">
    <soap:binding style="…" transport="…">
    <operation name="…">
        <soap:operation style="…" soapAction="…">
        <input>
            <soap:body use="…" encodingStyle="…">
        </input>
        <output>
            <soap:body use="…" encodingStyle="…">
        </output>
        <fault>
            <soap:fault name="…" use="…" encodingStyle="…"
        </fault>
    </operation>
</binding>
```

Only one `soap:fault` element is defined for a particular operation. The operation must be a request-response or solicit-response type of operation, with both `<input>` and `<output>` elements. The `soap:fault` element is used to transmit error and status information within a SOAP response message.

Note: A fault message must consist of only a single message part. Also, it is assumed that the `soap:operation style` element in the WSDL is set to `document`, because faults do not contain parameters.

Attributes

Table 16 describes the attributes defined within the `soap:fault` element.

Table 16: *soap:fault attributes*

Configuration Attribute	Explanation
`name`	This specifies the name of the fault. This relates back to the `name` attribute for the `<fault>` element specified for the corresponding operation within the `<portType>` component of the WSDL contract.

Table 16: *soap:fault attributes*

Configuration Attribute	Explanation
`use`	This attribute is used in the same way as the `use` attribute within the `soap:body` element. See "use" on page 432 for more details.
`encodingStyle`	This attribute is used in the same way as the `encodingStyle` attribute within the `soap:body` element. See "encodingStyle" on page 433 for more details.

soap:address element

Overview

The `soap:address` element in a WSDL contract is defined within the `<port>` component within the `<service>` component, as follows:

```
<service name="…">
    <port binding="…" name="…">
        <soap:address location="…">
    </port>
</service>
```

Only one `soap:address` element is defined in a WSDL contract. It is only specified when you want to use SOAP over HTTP. If you want to use SOAP over a different transport (for example, IIOP), the element name in this case is `iiop:address`. Similarly, if you want to use a different payload format over HTTP, the `http-conf:client URL` attribute is used instead.

Note: When you are using SOAP over HTTP, the `http-conf:client` and `http-conf:server` elements can still be validly specified as peer elements of the `soap:address` element. See "Creating an HTTP Service" on page 300 for more details of `http-conf:client` and `http-conf:server`.

Attributes

Table 17 describes the `location` attribute defined within the `soap:address` element.

Table 17: *Attribute for soap:address*

Configuration Attribute	Explanation

`location`	This specifies the URL of the server to which the client request is being sent.
	Valid values are of the form:
	`http://myserver/mypath/`
	`https://myserver/mypath`
	`http://myserver:9001/mypath`
	`http://myserver:9001-9010/mypath`
	The `soap:address` element is mandatory if you want to use SOAP over HTTP. It does not need to be set if you want to use SOAP over any other transport.

CORBA Type Mapping

The CORBA plug-in uses a detailed type map to ensure that data is transmitted without ambiguity.

Overview

To ensure that messages are converted into the proper format for a CORBA application to understand, Artix contracts need to unambiguously describe how data is mapped to CORBA data types. For primitive types, the mapping is straightforward. However, complex types such as structures, arrays, and exceptions require more detailed descriptions.

Unsupported types

The following CORBA types are not supported:

- value types
- boxed values
- local interfaces
- abstract interfaces
- forward-declared interfaces

In this appendix

This appendix discusses the following topics:

Primitive Type Mapping

Mapping chart

Most primitive IDL types are directly mapped to primitive XML Schema types. Table 18 lists the mappings for the supported IDL primitive types.

Table 18: *Primitive Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type	Artix Java Type
Any	xsd:anyType	corba:any	IT_Bus::AnyHolder	com.iona.webservices.reflect.types.AnyType
boolean	xsd:boolean	corba:boolean	IT_Bus::Boolean	boolean
char	xsd:byte	corba:char	IT_Bus::Char	byte
wchar	xsd:string	corba:wchar		java.lang.String
double	xsd:double	corba:double	IT_Bus::Double	double
float	xsd:float	corba:float	IT_Bus::Float	float
octet	xsd:unsignedByte	corba:octet	IT_Bus::Octet	short
long	xsd:int	corba:long	IT_Bus::Long	int
long long	xsd:long	corba:longlong	IT_Bus::LongLong	long
short	xsd:short	corba:short	IT_Bus::Short	short
string	xsd:string	corba:string	IT_Bus::String	java.lang.String
wstring	xsd:string	corba:wstring		java.lang.String
unsigned short	xsd:unsignedShort	corba:ushort	IT_Bus::UShort	int
unsigned long	xsd:unsignedInt	corba:ulong	IT_Bus::ULong	long
unsigned long long	xsd:unsignedLong	corba:ulonglong	IT_Bus::ULongLong	java.math.BigInteger
TimeBase::UtcT	xsd:dateTime[a]	corba:dateTime	IT_Bus::DateTime	java.util.Calendar

a. The mapping between `xsd:dateTime` and `TimeBase:UtcT` is only partial. For the restrictions see "Unsupported time/date values" on page 444

Unsupported types

Artix does not support the CORBA `long double` type.

Unsupported time/date values

The following `xsd:dateTime` values cannot be mapped to `TimeBase::UtcT`:

- Values with a local time zone. Local time is treated as a 0 UTC time zone offset.
- Values prior to 15 October 1582.
- Values greater than approximately 30,000 A.D..

The following `TimeBase::UtcT` values cannot be mapped to `xsd:dateTime`:

- Values with a non-zero inacclo or inacchi.
- Values with a time zone offset that is not divisible by 30 minutes.
- Values with time zone offsets greater than 14:30 or less than -14:30.
- Values with greater than millisecond accuracy.
- Values with years greater than 9999.

Example

The mapping of primitive types is handled in the CORBA binding section of the Artix contract. For example, consider an input message that has a part, `score`, that is described as an `xsd:int` as shown in Example 110.

Example 110: *WSDL Operation Definition*

```
<message name="runsScored">
  <part name="score" />
</message>
<portType ...>
  <operation name="getRuns">
    <input message="tns:runsScored" name="runsScored" />
  </operation>
</portType>
```

It is described in the CORBA binding as shown in Example 111.

Example 111: *Example CORBA Binding*

```
<binding ...>
  <operation name="getRuns">
    <corba:operation name="getRuns">
      <corba:param name="score" mode="in" idltype="corba:long"/>
    </corba:operation>
    <input/>
    <output/>
  </operation>
</binding>
```

The IDL is shown in Example 112.

Example 112: *getRuns IDL*

```
// IDL
void getRuns(in score);
```

Complex Type Mapping

Overview

Because complex types (such as structures, arrays, and exceptions) require a more involved mapping to resolve type ambiguity, the full mapping for a complex type is described in a `<corba:typeMapping>` element at the bottom of an Artix contract. This element contains a type map describing the metadata required to fully describe a complex type as a CORBA data type. This metadata may include the members of a structure, the bounds of an array, or the legal values of an enumeration.

The `<corba:typeMapping>` element requires a `targetNamespace` attribute that specifies the namespace for the elements defined by the type map. The default URI is `http://schemas.iona.com/bindings/corba/typemap`. By default, the types defined in the type map are referred to using the `corbatm:` prefix.

Mapping chart

Table 19 shows the mappings from complex IDL types to XMLSchema, Artix CORBA type, and Artix C++ types.

Table 19: *Complex Type Mapping for CORBA Plug-in*

IDL Type	XML Schema Type	CORBA Binding Type	Artix C++ Type
`struct`	See Example 114	`corba:struct`	`IT_Bus::SequenceComplexType`
`enum`	See Example 115	`corba:enum`	`IT_Bus::AnySimpleType`
`fixed`	`xsd:decimal`	`corba:fixed`	`IT_Bus::Decimal`
`union`	See Example 120	`corba:union`	`IT_Bus::ChoiceComplexType`
`typedef`	See Example 123		
`array`	See Example 125	`corba:array`	`IT_Bus::ArrayT<>`
`sequence`	See Example 131	`corba:sequence`	`IT_Bus::ArrayT<>`
`exception`	See Example 134	`corba:exception`	`IT_Bus::UserFaultException`

Structures

Mapping

Structures are mapped to `<corba:struct>` elements. A `<corba:struct>` element requires three attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`repositoryID`	The fully specified repository ID for the CORBA type.

The elements of the structure are described by a series of `<corba:member>` elements. The elements must be declared in the same order used in the IDL representation of the CORBA type. A `<corba:member>` requires two attributes:

`name`	The name of the element
`idltype`	The IDL type of the element. This type can be either a primitive type or another complex type that is defined in the type map.

Example

For example, you may have a structure, `personalInfo`, similar to the one in Example 113.

Example 113: *personalInfo*

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

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

It can be represented in the CORBA type map as shown in Example 114:

Example 114:*CORBA Type Map for personalInfo*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:struct name="personalInfo" type="xsd1:personalInfo" repositoryID="IDL:personalInfo:1.0">
    <corba:member name="name" idltype="corba:string" />
    <corba:member name="age" idltype="corba:long" />
    <corba:member name="hairColor" idltype="corbatm:hairColorType" />
  </corba:struct>
</corba:typeMapping>
```

The idltype `corbatm:hairColorType` refers to a complex type that is defined earlier in the CORBA type map.

Enumerations

Mapping

Enumerations are mapped to `<corba:enum>` elements. A `<corba:enum>` element requires three attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`repositoryID`	The fully specified repository ID for the CORBA type.

The values for the enumeration are described by a series of `<corba:enumerator>` elements. The values must be listed in the same order used in the IDL that defines the CORBA enumeration. A `<corba:enumerator>` element takes one attribute, `value`.

Example

For example, the enumeration defined in Example 113 on page 447, `hairColorType`, can be represented in the CORBA type map as shown in Example 115:

Example 115: *CORBA Type Map for hairColorType*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:enum name="hairColorType" type="xsd1:hairColorType"
   repositoryID="IDL:hairColorType:1.0">
    <corba:enumerator value="red" />
    <corba:enumerator value="brunette" />
    <corba:enumerator value="blonde" />
  </corba:enum>
</corba:typeMapping>
```

Fixed

Mapping

Fixed point data types are a special case in the Artix contract mapping. A CORBA fixed type is represented in the logical portion of the contract as the XML Schema primitive type `xsd:decimal`. However, because a CORBA fixed type requires additional information to be fully mapped to a physical CORBA data type, it must also be described in the CORBA type map section of an Artix contract.

CORBA fixed data types are described using a `<corba:fixed>` element. A `<corba:fixed>` element requires five attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`repositoryID`	The fully specified repository ID for the CORBA type.
`type`	The logical type the structure is mapping (for CORBA fixed types, this is always `xsd:decimal`).
`digits`	The upper limit for the total number of digits allowed. This corresponds to the first number in the fixed type definition.
`scale`	The number of digits allowed after the decimal point. This corresponds to the second number in the fixed type definition.

Example

For example, the fixed type defined in Example 116, `myFixed`, would be

Example 116: *myFixed Fixed Type*

```
\\IDL
typedef fixed<4,2> myFixed;
```

described by a type entry in the logical type description of the contract, as shown in Example 117.

Example 117: *Logical description from myFixed*

```
<xsd:element name="myFixed" type="xsd:decimal"/>
```

In the CORBA type map portion of the contract, it would be described by an entry similar to Example 118. Notice that the description in the CORBA type map includes the information needed to fully represent the characteristics of this particular fixed data type.

Example 118:*CORBA Type Map for myFixed*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:fixed name="myFixed" repositoryID="IDL:myFixed:1.0" type="xsd:decimal" digits="4"
    scale="2" />
</corba:typeMapping>
```

Unions

Overview

Unions are particularly difficult to describe using the WSDL framework of an Artix contract. In the logical data type descriptions, the difficulty is how to describe the union without losing the relationship between the members of the union and the discriminator used to select the members. The easiest method is to describe a union using an `<xsd:choice>` and list the members in the specified order. The OMG's proposed method is to describe the union as an `<xsd:sequence>` containing one element for the discriminator and an `<xsd:choice>` to describe the members of the union. However, neither of these methods can accurately describe all the possible permutations of a CORBA union.

Artix Mapping

Artix's IDL compiler generates a contract that describes the logical union using both methods. The description using `<xsd:sequence>` is named by prepending `_omg_` to the types name. The description using `<xsd:chioce>` is used as the representation of the union throughout the contract.

For example consider the union, `myUnion`, shown in Example 119:

Example 119: *myUnion IDL*

```
//IDL
union myUnion switch (short)
{
  case 0:
    string case0;
  case 1:
  case 2:
    float case12;
  default:
    long caseDef;
};
```

This union is described in the logical portion of the contact with entries similar to those shown in Example 120:

Example 120: *myUnion Logical Description*

```
<xsd:complexType name="myUnion">
  <xsd:choice>
    <xsd:element name="case0" type="xsd:string"/>
    <xsd:element name="case12" type="xsd:float"/>
    <xsd:element name="caseDef" type="xsd:int"/>
  </xsd:choice>
</xsd:complexType>
<xsd:complexType name="_omg_myUnion4">
  <xsd:sequence>
    <xsd:element minOccurs="1" maxOccurs="1" name="discriminator" type="xsd:short"/>
    <xsd:choice minOccurs="0" maxOccurs="1">
      <xsd:element name="case0" type="xsd:string"/>
      <xsd:element name="case12" type="xsd:float"/>
      <xsd:element name="caseDef" type="xsd:int"/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

In the CORBA type map portion of the contract, the relationship between the union's discriminator and its members must be resolved. This is accomplished using a `<corba:union>` element. A `<corba:union>` element has four mandatory attributes.

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`discriminator`	The IDL type used as the discriminator for the union.
`repositoryID`	The fully specified repository ID for the CORBA type.

The members of the union are described using a series of nested `<corba:unionbranch>` elements. A `<corba:unionbranch>` element has two required attributes and one optional attribute.

`name`	A unique identifier used to reference the union member.
`idltype`	The IDL type of the union member. This type can be either a primitive type or another complex type that is defined in the type map.

`default`	The optional attribute specifying if this member is the default case for the union. To specify that the value is the default set this attribute to `true`.

Each `<corba:unionbranch>` except for one describing the union's default member will have at least one nested `<corba:case>` element. The `<corba:case>` element's only attribute, `label`, specifies the value used to select the union member described by the `<corba:unionbranch>`.

For example `myUnion`, Example 119 on page 452, would be described with a CORBA type map entry similar to that shown in Example 121.

Example 121: *myUnion CORBA type map*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:union name="myUnion" type="xsd1:myUnion" discriminator="corba:short"
   repositoryID="IDL:myUnion:1.0">
    <corba:unionbranch name="case0" idltype="corba:string">
      <corba:case label="0" />
    </corba:unionbranch>
    <corba:unionbranch name="case12" idltype="corba:float">
      <corba:case label="1" />
      <corba:case label="2" />
    </corba:unionbranch>
    <corba:unionbranch name="caseDef" idltype="corba:long" default="true"/>
  </corba:union>
</corba:typeMapping>
```

Type Renaming

Mapping

Renaming a type using a `typedef` statement is handled using a `<corba:alias>` element in the CORBA type map. The Artix IDL compiler also adds a logical description for the renamed type in the `<types>` section of the contract, using an `<xsd:simpleType>`.

Example

For example, the definition of myLong in Example 122, can be described as

Example 122: *myLong IDL*

```
//IDL
typedef long myLong;
```

shown in Example 123:

Example 123: *myLong WSDL*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="typedef.idl" ...>
  <types>
  ...
    <xsd:simpleType name="myLong">
      <xsd:restriction base="xsd:int"/>
    </xsd:simpleType>
  ...
  </types>
...
  <corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
    <corba:alias name="myLong" type="xsd:int" repositoryID="IDL:myLong:1.0"
   basetype="corba:long"/>
  </corba:typeMapping>
</definitions>
```

Arrays

Logical mapping

Arrays are described in the logical portion of an Artix contract, using an `<xsd:sequence>` with its `minOccurs` and `maxOccurs` attributes set to the value of the array's size. For example, consider an array, `myArray`, as defined in Example 124.

Example 124: *myArray IDL*

```
//IDL
typedef long myArray[10];
```

Its logical description will be similar to that shown in Example 125:

Example 125: *myArray logical description*

```
<xsd:complexType name="myArray">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="10" maxOccurs="10" />
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

In the CORBA type map, arrays are described using a `<corba:array>` element. A `<corba:array>` has five required attributes.

`name`	A unique identifier used to reference the CORBA type in the binding.
`repositoryID`	The fully specified repository ID for the CORBA type.
`type`	The logical type the structure is mapping.
`elemtype`	The IDL type of the array's element. This type can be either a primitive type or another complex type that is defined within the type map.
`bound`	The size of the array.

For example, the array `myArray` will have a CORBA type map description similar to the one shown in Example 126.

Example 126: *myArray CORBA type map*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
  <corba:array name="myArray" repositoryID="IDL:myArray:1.0" type="xsd1:myArray"
   elemtype="corba:long" bound="10"/>
</corba:typeMapping>
```

Multidimensional Arrays

Logical mapping

Multidimensional arrays are handled by creating multiple arrays and combining them to form the multidimensional array. For example, an array defined as shown in Example 127

Example 127: *Multidimensional Array*

```
\\ IDL
typedef long array2d[10][10];
```

generates the logical description shown in Example 128.

Example 128: *Logical Description of a Multidimensional Array*

```
<xsd:complexType name="_1_array2d">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="10" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="array2d">
  <xsd:sequence>
    <xsd:element name="item" type="xsd1:_1_array2d" minOccurs="10" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

The corresponding entry in the CORBA type map is:

Example 129: *CORBA Type Map for a Multidimensional Array*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
  <corba:anonarray name="_2_array2d" type="xsd1:_2_array2d" elemtype="corba:long" bound="10"/>
  <corba:array name="array2d" repositoryID="IDL:array2d:1.0" type="xsd1:array2d"
   elemtype="corbatm:_2_array2d" bound="10"/>
</corba:typeMapping>
```

Sequences

Logical mapping

Because CORBA sequences are an extension of arrays, sequences are described in Artix contracts similarly. Like arrays, sequences are described in the logical type section of the contract using `<xsd:sequence>` elements. Unlike arrays, the `minOccurs` and `maxOccurs` attributes do not have the same value. `minOccurs` is set to `0` and `maxOccurs` is set to the upper limit of the sequence. If the sequence is unbounded, `maxOccurs` is set to `unbounded`.

For example, the two sequences defined in Example 130, `longSeq` and `charSeq`:

Example 130: *IDL Sequences*

```
\\ IDL
typedef sequence<long> longSeq;
typedef sequence<char, 10> charSeq;
```

are described in the logical section of the contract with entries similar to those shown in Example 131.

Example 131: *Logical Description of Sequences*

```
<xsd:complexType name="longSeq">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:int" minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="charSeq">
  <xsd:sequence>
    <xsd:element name="item" type="xsd:byte" minOccurs="0" maxOccurs="10"/>
  </xsd:sequence>
</xsd:complexType>
```

CORBA type mapping

In the CORBA type map, sequences are described using a `<corba:sequence>` element. A `<corba:sequence>` has five required attributes.

`name`	A unique identifier used to reference the CORBA type in the binding.
`repositoryID`	The fully specified repository ID for the CORBA type.

`type`	The logical type the structure is mapping.
`elemtype`	The IDL type of the sequence's elements. This type can be either a primitive type or another complex type that is defined within the type map.
`bound`	The size of the sequence.

For example, the sequences described in Example 131 has a CORBA type map description similar to that shown in Example 132:

Example 132:*CORBA type map for Sequences*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
    <corba:sequence name="longSeq" repositoryID="IDL:longSeq:1.0" type="xsd1:longSeq"
   elemtype="corba:long" bound="0"/>
    <corba:sequence name="charSeq" repositoryID="IDL:charSeq:1.0" type="xsd1:charSeq"
   elemtype="corba:char" bound="10"/>
  </corba:typeMapping>
```

Exceptions

Mapping

Because exceptions typically return more than one piece of information, they require both an abstract type description and a CORBA type map entry. In the abstract type description, exceptions are described much like structures. In the CORBA type map, exceptions are described using `<corba:exception>` elements. A `<corba:exception>` element has three required attributes:

`name`	A unique identifier used to reference the CORBA type in the binding.
`type`	The logical type the structure is mapping.
`repositoryID`	The fully specified repository ID for the CORBA type.

The pieces of data returned with the exception are described by a series of `<corba:member>` elements. The elements must be declared in the same order as in the IDL representation of the exception. A `<corba:member>` has two required attributes:

`name`	The name of the element
`idltype`	The IDL type of the element. This type can be either a primitive type or another complex type that is defined within the type map.

Example

For example, the exception defined in Example 133, `idNotFound`,

Example 133: *idNotFound Exception*

```
\\IDL
exception idNotFound
{
  short id;
};
```

would be described in the logical type section of the contract, with an entry similar to that shown in Example 134:

Example 134:*idNotFound logical structure*

```
<xsd:complexType name="idNotFound">
  <xsd:sequence>
    <xsd:element name="id" type="xsd:short"/>
  </xsd:sequence>
</xsd:complexType>
```

In the CORBA type map portion of the contract, `idNotFound` is described by an entry similar to that shown in Example 135:

Example 135:*CORBA Type Map for idNotFound*

```
<corba:typeMapping targetNamespace="http://schemas.iona.com/bindings/corba/typemap">
...
  <corba:exception name="idNotFound" type="xsd1:idNotFound" repositoryID="IDL:idNotFound:1.0">
    <corba:member name="id" idltype="corba:short" />
  </corba:exception>
</corba:typeMapping>
```

Recursive Type Mapping

Overview

Both CORBA IDL and XMLSchema allow you define recursive data types. Because both type definition schemes support recursion, Artix directly maps recursive types between IDL and XMLSchema. The CORBA typemap generated by Artix to support the CORBA binding is straightforward and directly reflects the recursive nature of the data types.

Defining recursive types in XMLSchema

Recursive data types are defined in XMLSchema as complex types using the `<complexType>` element. XMLSchema supports two means of defining a recursive type. The first is to have an element of a complex type be of a type that includes an element of the type being defined. Example 136 shows a recursive complex type XMLSchema type, `allAboutMe`, defined using a named type.

Example 136: *Recursive XMLSchema Type*

```
<complexType name="allAboutMe">
  <sequence>
    <element name="shoeSize" type="xsd:int" />
    <element name="mated" type="xsd:boolean" />
    <element name="conversation" type="tns:moreMe" />
  </sequence>
</complexType>
<complexType name="moreMe">
  <sequence>
    <element name="item" type="tns:allAboutMe"
             maxOccurs="unbounded" />
  </sequence>
</complexType>
```

XMLSchema also supports the definition of recursive types using anonymous types. However, Artix does not support this style of defining recursive types.

CORBA typemap

As shown in Example 137, Artix maps recursive types into the CORBA typemap section of the Artix contract as it would non-recursive types, except that it maps the recursive element, which is a sequence in this case, to an

anonymous type using the <corba:anonsequence> element. The <corba:anonsequence> specifies that when IDL is generated from this binding the associated sequence will not generate a new type for itself.

Example 137: *Recursive CORBA Typemap*

```
<corba:anonsequence name="moreMe" bound="0"
                    elemtype="ns1:allAboutMe" type="xsd1:me" />
<corba:struct name="allAboutMe"
              repositoryID="IDL:allAboutMe:1.0"
              type="tns:allAboutMe">
  <corba:member name="shoeSize" idltype="corba:long"/>
  <corba:member name="mated" idltype="corba:boolean"/>
  <corba:member name="conversation" idltype="ns1:moreMe"/>
</corba:struct>
```

Generated IDL

While the XML in the CORBA typemap does not explicitly retain the recursive nature of recursive XMLSchema types, the IDL generated from the typemap restores the recursion in the IDL type. The IDL generated from the typemap in Example 137 on page 464 defines allAboutMe using recursion. Example 138 shows the generated IDL.

Example 138: *IDL for a Recursive Data Type*

```
\\IDL
struct allAboutMe
{
  long shoeSize;
  boolean mated;
  sequence<allAboutMe> conversation;
};
```

Mapping XMLSchema Features that are not Native to IDL

Overview

There are a number of data types that you can describe in your Artix contract using XMLSchema that are not native to IDL. Artix can map these data types into legal IDL so that your CORBA systems can interoperate with applications that use these data type descriptions in their contracts.

These features include:

- Binary Types
- Attributes
- Nested Choices
- Inheritance
- Nillable

Binary Types

Overview

There are three binary types defined in XMLSchema that have direct correlation to IDL data-types. These types are:

- `xsd:base64Binary`
- `xsd:hexBinary`
- `soapenc:base64`

These types are all mapped to octet sequences in CORBA.

Example

For example, the schema type, `joeBinary`, described in Example 139 results in the CORBA typemap description shown in Example 140.

Example 139: *joeBinary schema description*

```
<xsd:element name="joeBinary type="xsd:hexBinary" />
```

The resulting IDL for `joeBinary` is shown in Example 141.

Example 140: *joeBinary CORBA typemap*

```
<corba:sequence name="joeBinary" bound="0"
 elemtype="corba:octet" repositoryID="IDL:joeBinary:1.0"
 type="xsd:hexBinary" />
```

The mappings for `xsd:base64Binary` and `soapenc:base64` would be similar except that the `type` attribute in the CORBA typemap would specify the appropriate type.

Example 141: *joeBinary IDL*

```
\\IDL
typedef sequence<octet> joeBinary;
```

Attributes

Mapping

Required XMLSchema attributes are treated as normal elements in a CORBA structure.

Note: Attributes are not supported for complex types defined with `<choice>`.

Example

For example, the complex type, `madAttr`, described in Example 142 contains two attributes, `material` and `size`.

Example 142: *madAttr XMLSchema*

```
<complexType name="madAttr">
  <sequence>
    <element name="style" type="xsd:string" />
    <element name="gender" type="xsd:byte" />
  </sequence>
  <attribute name="size" type="xsd:int" />
  <attribute name="material" />
    <simpleType>
      <restriction base="xsg:string">
        <maxLength value="3" />
      </restriction>
    </simpleType>
  </attribute>
<complexType>
```

`madAttr` would generate the CORBA typemap shown in Example 143. Notice that `size` and `material` are simply incorporated into the `madAttr` structure in the CORBA typemap.

Example 143: *madAttr CORBA typemap*

```
<corba:annonstring bound="3" name="materialType" type="tns:material" />
<corba:struct name="madAttr" repositoryID="IDL:madAttr:1.0" type="typens:madAttr">
  <corba:member name="style" idltype="corba:string"/>
  <corba:member name="gender" idltype="corba:char"/>
  <corba:member name="size" idltype="corba:long"/>
  <corba:member name="material" idltype="ns1:materialType"/>
</corba:struct>
```

Similarly, in the IDL generated using a contract containing `madAttr`, the attributes are made elements of the structure and are placed in the order in which they are listed in the contract. The resulting IDL structure is shown in Example 144.

Example 144: *madAttr IDL*

```
\\IDL
struct madAttr
{
  string style;
  char gender;
  long size;
  string<3> material;
}
```

Nested Choices

Mapping

When mapping complex types containing nested `xsd:choice` elements into CORBA, Artix will break the nested `xsd:choice` elements into separate `unions` in CORBA. The resulting union will have the name of the original complex type with `ChoiceType` appended to it. So, if the original complex type was named *`joe`*, the union representing the nested choice would be named `joeChoiceType`.

The nested choice in the original complex type will be replaced by an element of the new union created to represent the nested choice. This element will have the name of the new union with `_f` appended. So if the original structure was named `carla`, the replacement element will be named `carlaChoiceType_f`.

The original type description will not be changed, the break out will only appear in the CORBA typemap and in the resulting IDL.

Example

For example, the complex type `details`, shown in Example 145, contains a nested `choice`.

Example 145:*details XMLSchema*

```
<complexType name="Details">
  <sequence>
    <element name="name" type="xsd:string"/>
    <element name="address" type="xsd:string"/>
    <choice>
      <element name="employer" type="xsd:string"/>
      <element name="unemploymentNumber" type="xsd:int"/>
    </choice>
  </sequence>
</complexType>
```

The resulting CORBA typemap, shown in Example 146, contains a new union, `detailsChoiceType`, to describe the nested choice. Note that the `type` attribute for both `details` and `detailsChoiceType` has the name of the

original complex type defined in the schema. The nested choice is represented in the original structure as a member of type `detailsChoiceType`.

Example 146: *details CORBA typemap*

```
<corba:struct name="details" repositoryID="IDL:details:1.0" type="xsd1:details">
  <corba:member idltype="corba:string" name="name"/>
  <corba:member idltype="corba:string" name="address"/>
  <corba:member idltype="ns1:detailsChoiceType" name="detailsChoiceType_f"/>
</corba:struct>
<corba:union discriminator="corba:long" name="detailsChoiceType"
             repositoryID="IDL:detailsChoiceType:1.0" type="xsd1:details">
  <corba:unionbranch idltype="corba:string" name="employer">
    <corba:case label="0"/>
  </corba:unionbranch>
  <corba:unionbranch idltype="corba:long" name="unemploymentNumber">
    <corba:case label="1"/>
  </corba:unionbranch>
</corba:union>
```

The resulting IDL is shown in Example 147.

Example 147: *details IDL*

```
\\IDL
union detailsChoiceType switch(long)
{
  case 0:
    string employer;
  case 1:
    long unemploymentNumber;
};
struct details
{
  string name;
  string address;
  detailsChoiceType DetailsChoiceType_f;
};
```

Inheritance

Mapping

XMLSchema describes inheritance using the `<complexContent>` tag and the `<extension>` tag. For example the complex type `seaKayak`, described in Example 148, extends the complex type `kayak` by including two new fields.

Example 148: *seaKayak XMLSchema*

```
<complexType name="kayak">
  <sequence>
    <element name="length" type="xsd:int" />
    <element name="width" type="xsd:int" />
    <element name="material" type="xsd:string" />
  </sequence>
</complexType>
<complexType name="seaKayak">
  <complexContent>
    <extension base="kayak">
      <sequence>
        <element name="chines" type="xsd:string" />
        <element name="cockpitStyle" type="xsd:string" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

When complex types using `<complexContent>` are mapped into CORBA types, Artix creates generates an intermediate type to represent the complex data defined within the `<complexContent>` element. The intermediate type is named by appending an identifier describing the complex content to the new type's name. Table 20 shows the complex content identifiers used appended to the intermediate type name.

Table 20: *Complex Content Identifiers in CORBA Typemap*

XMLSchema Type	Typemap Identifier
`<sequence>`	`SequenceStruct`
`<all>`	`AllStruct`
`<choice>`	`ChoiceType`

The CORBA type generated to represent the XMLSchema type generated to represent the type derived by extension will have an element of the type that it extends, named *baseType_f* and an element of the intermediate type, named *intermediateType_f*. Any attributes that are defined in the extended type are then mapped into the new CORBA type following the rules for mapping XMLSchema attributes into CORBA types.

Example

Example 149 shows how Artix maps the complex types defined in Example 148 on page 471 into a CORBA type map.

Example 149: *seaKayak CORBA type map*

```
<corba:struct name="kayak" repositoryID="IDL:kayak:1.0" type="tns:kayak">
  <corba:element name="length" idltype="corba:long" />
  <corba:element name="width" idltype="corba:long" />
  <corba:element name="material" idltype="corba:string" />
</corba:struct>
<corba:struct name="seaKayak" repositoryID="IDL:seaKayak:1.0" type="tns:seaKayak">
  <corba:element name="kayak_f" idltype="ns1:kayak" />
  <corba:element name="seaKayakSequenceStruct_f" idltype="ns1:seaKayakSequenceStruct" />
</corba:struct>
<corba:struct name="seaKayakSequenceStruct" repositoryID="IDL:seaKayakSequenceStruct:1.0"
              type="tns:seaKayakSequenceStruct">
  <corba:element name="chines" idltype="corba:string" />
  <corba:element name="cockpitStyle" idltype="corba:string" />
</corba:struct>
```

The IDL generated by Artix for the types defined in Example 148 on page 471 is shown in Example 150.

Example 150: *seaKayak IDL*

```
\\ IDL
struct kayak
{
  long length;
  long width;
  string material;
};
struct seaKayakSequenceStruct
{
  string chines;
  string cockpitStyle;
};
```

Example 150:*seaKayak IDL*

```
struct seaKayak
{
  kayak kayak_f;
  seaKayakSequenceStruct seqKayakSequenceStruct_f;
};
```

Nillable

Mapping

XMLSchema supports an optional attribute, `nillable`, that specifies that an element can be `nil`. Setting an element to `nil` is different than omitting an element whose `minOccurs` attribute is set to `0`; the element must be included as part of the data sent in the message.

Elements that have `nillable="true"` set in their logical description are mapped to a CORBA union with a single case, `TRUE`, that holds the value of the element if it is not set to `nil`.

Example

For example, imagine a service that maintains a database of information on people who download software from a web site. The only required piece of information the visitor needs to supply is their zip code. Optionally, visitors can supply their name and e-mail address. The data is stored in a data structure, `webData`, shown in Example 151.

Example 151: *webData XMLSchema*

```
<complexType name="webData">
  <sequence>
    <element name="zipCode" type="xsd:int" />
    <element name="name" type="xsd:string" nillable="true" />
    <element name="emailAddress" type="xsd:string"
             nillable="true" />
  </sequence>
</complexType>
```

When `webData` is mapped to a CORBA binding, it will generate a union, `string_nil`, to provide for the mapping of the two nillable elements, name and `emailAddress`. Example 152 shows the CORBA typemap for `webData`.

Example 152: *webData CORBA Typemap*

```
<corba:typemapping ...>
  <corba:struct name="webData" repositoryID="IDL:webData:1.0" type="xsd1:webData">
    <corba:member idltype="corba:long" name="zipCode"/>
    <corba:member idltype="ns1:string_nil" name="name"/>
    <corba:member idltype="ns1:string_nil" name="emailAddress"/>
  </corba:struct>
  <corba:union discriminator="corba:boolean" name="string_nil" repositoryID="IDL:string_nil:1.0"
               type="xsd1:emailAddress">
    <corba:unionbranch idltype="corba:string" name="value">
      <corba:case label="TRUE"/>
    </corba:unionbranch>
  </corba:union>
</corba:typeMapping>
```

The type assigned to the union, `string_nil`, does not matter as long as the type assigned maps back to an `xsd:string`. This is true for all nillable element types.

Example 153 shows the IDL for `webData`.

Example 153: *webData IDL*

```
\\IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
struct webData {
    long zipCode;
    string_nil name;
    string_nil emailAddress;
};
```

Optional Attributes

Overview

Attributes defined as optional in XMLSchema are mapped similar to `nillable` elements. Attributes that do not have `use="required"` set in their logical description are mapped to a CORBA union with a single case, `TRUE`, that holds the value of the element if it is set.

Note: By default attributes are optional if `use` is not set to `required`.

For example, you could define the complex type in Example 151 using attributes instead of a sequence. The data description for `webData` defined with attributes is shown in Example 154.

Example 154: *webData XMLSchema Using Attributes*

```
<complexType name="webData">
  <attribute name="zipCode" type="xsd:int" use="required"/>
  <attribute name="name" type="xsd:string"/>
  <attribute name="emailAddress" type="xsd:string"/>
</complexType>
```

CORBA type mapping

When `webData` is mapped to a CORBA binding, it will generate a union, `string_nil`, to provide for the mapping of the two nillable elements, name and `emailAddress`. Example 155 shows the CORBA typemap for `webData`.

Example 155: *webData CORBA Typemap*

```
<corba:typemapping ...>
  <corba:union discriminator="corba:boolean" name="string_nil" repositoryID="IDL:string_nil:1.0"
               type="xsd1:emailAddress">
    <corba:unionbranch idltype="corba:string" name="value">
      <corba:case label="TRUE"/>
    </corba:unionbranch>
  </corba:union>
  <corba:struct name="webData" repositoryID="IDL:webData:1.0" type="xsd1:webData">
    <corba:member idltype="corba:long" name="zipCode"/>
    <corba:member idltype="ns1:string_nil" name="name"/>
    <corba:member idltype="ns1:string_nil" name="emailAddress"/>
  </corba:struct>
</corba:typeMapping>
```

The type assigned to the union, `string_nil`, does not matter as long as the type assigned maps back to an `xsd:string`. This is true for all optional attributes.

Example 156 shows the IDL for `webData`.

Example 156:*webData IDL*

```
\\IDL
union string_nil switch(boolean) {
    case TRUE:
        string value;
};
struct webData {
    long zipCode;
    string_nil name;
    string_nil emailAddress;
};
```

Artix References

Overview

Artix references provide a means of passing a reference to a service between two operations. Because Artix services are Web services, their references are very different than references used in CORBA. Artix does, however, provide a mechanism for passing Artix references to CORBA applications over the Artix CORBA transport. This functionality allows CORBA applications to make calls on Artix services that return references to other Artix services.

For a detailed discussion of Artix references read *Developing Artix Applications in C++*.

Specifying references to map to CORBA

Artix references are mapped into a CORBA in one of two ways. The simplest way is to just specify your reference types as you would for an Artix service using SOAP. In this case, the Artix references are mapped into generic CORBA `Objects`.

The second method allows you to generate type-specific CORBA references, but requires some planning in the creation of your XMLSchema type definitions. When creating a reference type, you can specify the name of the CORBA binding that describes the interface in the physical section of the contract using an `<xsd:annotation>` element. Example 157 shows the syntax for specifying the binding in the type definition.

Example 157: *Reference Binding Specification*

```
<xsd:element name="typeName" type="references:Reference">
  <xsd:annotation>
    <xsd:appinfo>corba:binding=CORBABindingName</xsd:appinfo>
  </xsd:annotation>
</xsd:element>
```

When you specify a reference using the annotation, the CORBA binding generator and the IDL generator will inspect the specified binding and create a type-specific reference in the IDL generated for the contract that allows you to make use of the reference.

Note: Before you can generate a type-specific reference you need to generate the CORBA binding of the referenced interface.

CORBA typemap representation

Artix references are mapped to `<corba:object>` elements in the CORBA typemap section of an Artix contract. `<corba:object>` elements have four attributes:

`binding`	Specifies the binding to which the object refers. If the annotation element is left off the reference declaration in the schema, this attribute will be blank.
`name`	Specifies the name of the CORBA type. If the annotation element is left off the reference declaration in the schema, this attribute will be `Object`. If the annotation is used and the binding can be found, this attribute will be set to the name of the interface that the binding represents.
`repositoryID`	Specifies the repository ID of the generated IDL type. If the annotation element is left off the reference declaration in the schema, this attribute will be set to `IDL:omg.org/CORBA/Object/1.0`. If the annotation is used and the binding can be found, this attribute will be set to a properly formed repository ID based on the interface name.
`type`	Specifies the schema type from which the CORBA type is generated. This attribute is always set to `references:Reference`.

Example

Example 158 shows an Artix contract fragment that uses Artix references.

Example 158: *Reference Sample*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="bankService"
 targetNamespace="http://schemas.myBank.com/bankTypes"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.myBank.com/bankService"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://schemas.myBank.com/bankTypes"
 xmlns:corba="http://schemas.iona.com/bindings/corba"
 xmlns:corbatm="http://schemas.iona.com/typemap/corba/bank.idl"
 xmlns:references="http://schemas.iona.com/references">
```

Example 158:*Reference Sample*

```
  <types>
    <schema
     targetNamespace="http://schemas.myBank.com/bankTypes"
     xmlns="http://www.w3.org/2001/XMLSchema"
     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
      <xsd:import schemaLocation="./references.xsd"
                  namespace="http://schemas.iona.com/references"/>
...
      <xsd:element name="account" type="references:Reference">
        <xsd:annotation>
          <xsd:appinfo>
          corba:binding=AccountCORBABinding
          </xsd:appinfo>
        </xsd:annotation>
      </xsd:element>
  </schema>
</types>
...
  <message name="find_accountResponse">
    <part name="return" element="xsd1:account"/>
  </message>
  <message name="create_accountResponse">
    <part name="return" element="xsd1:account"/>
  </message>
```

Example 158:*Reference Sample*

```
  <portType name="Account">
    <operation name="account_id">
      <input message="tns:account_id" name="account_id"/>
      <output message="tns:account_idResponse"
              name="account_idResponse"/>
    </operation>
    <operation name="balance">
      <input message="tns:balance" name="balance"/>
      <output message="tns:balanceResponse"
              name="balanceResponse"/>
    </operation>
    <operation name="withdraw">
      <input message="tns:withdraw" name="withdraw"/>
      <output message="tns:withdrawResponse"
              name="withdrawResponse"/>
      <fault message="tns:InsufficientFundsException"
   name="InsufficientFunds"/>
    </operation>
    <operation name="deposit">
      <input message="tns:deposit" name="deposit"/>
      <output message="tns:depositResponse"
              name="depositResponse"/>
    </operation>
  </portType>
  <portType name="Bank">
    <operation name="find_account">
      <input message="tns:find_account" name="find_account"/>
      <output message="tns:find_accountResponse"
              name="find_accountResponse"/>
      <fault message="tns:AccountNotFound"
             name="AccountNotFound"/>
    </operation>
    <operation name="create_account">
      <input message="tns:create_account" name="create_account"/>
      <output message="tns:create_accountResponse"
              name="create_accountResponse"/>
      <fault message="tns:AccountAlreadyExistsException"
             name="AccountAlreadyExists"/>
    </operation>
  </portType>
</definitions>
```

The element named `account` is a reference to the interface defined by the `Account` port type and the `find_account` operation of `Bank` returns an element of type `account`. The annotation element in the definition of

account specifies the binding, AccountCORBABinding, of the interface to which the reference refers. Because you typically create the data types before you create the bindings, you must be sure that the generated binding name matches the name you specified. This can be controlled using the -b flag to wsdltocorba.

The first step to generating the Bank interface to use a type-specific reference to an Account is to generate the CORBA binding for the Account interface. You would do this by using the command **wsdltocorba -corba -i Account -b AccountCORBABinding** *wsdlName*.**wsdl** and replace *wsdlName* with the name of your contract. Once you have generated the CORBA binding for the Account interface, you can generate the CORBA binding and IDL for the Bank interface.

Example 159 shows the generated CORBA typemap resulting from generating both the Account and the Bank interfaces into the same contract.

Example 159:*CORBA Typemap with References*

```
<corba:typeMapping
  targetNamespace="http://schemas.myBank.com/bankService/corba/typemap/">
...
  <corba:object binding="" name="Object"
                repositoryID="IDL:omg.org/CORBA/Object/1.0" type="references:Reference"/>
  <corba:object binding="AccountCORBABinding" name="Account"
                repositoryID="IDL:Account:1.0" type="references:Reference"/>
</corba:typeMapping>
```

There are two entries because wsdltocorba was run twice on the same file. The first CORBA object is generated from the first pass of wsdltocorba to generate the CORBA binding for Account. Because wsdltocorba could not find the binding specified in the annotation, it generated a generic Object reference. The second CORBA object, Account, is generated by the second pass when the binding for Bank was generated. On that pass, wsldtocorba could inspect the binding for the Account interface and generate a type-specific object reference.

Example 160 shows the IDL generated for the Bank interface.

Example 160: *IDL Generated From Artix References*

```
//IDL
...
interface Account
{
  string account_id();

  float balance();

  void withdraw(in float amount)
    raises(::InsufficientFundsException);

  void deposit(in float amount);
};
interface Bank
{
  ::Account find_account(in string account_id)
    raises(::AccountNotFoundException);

  ::Account create_account(in string account_id,
                           in float initial_balance)
    raises(::AccountAlreadyExistsException);
};
```

WebSphere MQ Artix Extensions

Artix provides a number of proprietary WSDL extensions to configure a WebSphere MQ service.

Overview

To enable Artix to interoperate with WebSphere MQ, you must describe the WebSphere MQ port in the Artix contract defining the behavior of your Artix instance. Artix uses a number of proprietary WSDL extensions to specify all of the attributes that can be set on an WebSphere MQ port. The XMLSchema describing the extensions used for the WebSphere MQ port definition is included in the Artix installation under the `schemas` directory.

WebSphere MQ port attributes

Table 21 lists the attributes that are use to define the properties of a WebSphere MQ port. They are described in detail in the sections that follow the table.

Table 21: *WebSphere MQ Port Attributes*

Attributes	Description
`QueueManager`	Specifies the name of the queue manager.
`QueueName`	Specifies the name of the message queue.

Table 21: *WebSphere MQ Port Attributes*

Attributes	Description
ReplyQueueName	Specifies the name of the queue where response messages are received. This setting is ignored by WebSphere MQ servers when the client specifies the ReplyToQ in the request message's message descriptor.
ReplyQueueManager	Specifies the name of the reply queue manager. This setting is ignored by WebSphere MQ servers when the client specifies the ReplyToQMgr in the request message's message descriptor.
Server_Client	Specifies the type of WebSphere MQ installation is running on your applications host machine.
ModelQueueName	Specifies the name of the queue to be used as a model for creating dynamic queues.
AliasQueueName	Specifies the remote queue to which a server will put replies if its queue manager is not on the same host as the client's local queue manager.
ConnectionName	Specifies the name of the connection by which the adapter connects to the queue.
ConnectionReusable	Specifies if the connection can be used by more than one application.
ConnectionFastPath	Specifies if the queue manager will be loaded in process.
UsageStyle	Specifies if messages can be queued without expecting a response.
CorrelationStyle	Specifies what identifier is used to correlate request and response messages.
AccessMode	Specifies the level of access applications have to the queue.
Timeout	Specifies the amount of time within which the send and receive processing must begin before an error is generated.
MessageExpiry	Specifies the value of the MQ message descriptor's Expiry field.
MessagePriority	Specifies the value of the MQ message descriptor's Priority field.
Delivery	Specifies the value of the MQ message descriptor's Persistence field.
Transactional	Specifies if transaction operations must be performed on the messages.
ReportOption	Specifies the value of the MQ message descriptor's Report field.
Format	Specifies the value of the MQ message descriptor's Format field.

Table 21: *WebSphere MQ Port Attributes*

Attributes	Description
MessageId	Specifies the value for the MQ message descriptor's MsgId field..
CorrelationId	Specifies the value for the MQ message descriptor's CorrelId field.
ApplicationData	Specifies optional information to be associated with the message.
AccountingToken	Specifies the value for the MQ message decscriptor's AccountingToken field.
Convert	Specifies in the messages in the queue need to be converted to the system's native encoding.
ApplicationIdData	Specifies the value for the MQ message descriptor's ApplIdentityData field.
ApplicationOriginData	Specifies the value for the MQ message descriptor's ApplOriginData field.
UserIdentification	Specifies the value for the MQ message descriptor's UserIdentifier field.

QueueManager

Overview

`QueueManager` specifies the name of the WebSphere MQ queue manager used for request messages. Client applications will use this queue manager to place requests and server applications will use this queue manager to listen for request messages. You must provide this information when configuring a WebSphere MQ port.

Example

Example 161 shows a simple WebSphere MQ server port configuration for servers that listen for requests using a queue manager called `leo`.

Example 161: *MQ Port Definition*

```
<mq:server QueueManager="leo" QueueName="requestQ" />
```

QueueName

Overview

`QueueName` is a required attribute for a WebSphere MQ port. It specifies the request message queue. Client applications place request messages into this queue. Server applications take requests from this queue. The queue must be configured under the specified queue manager before it can be used.

Example

Example 162 shows a definition of a simple WebSphere MQ client that places oneway requests onto a queue called `ether`.

Example 162: *WebSphere MQ QueueName example*

```
<mq:client QueueManager="Qmgr" QueueName="ether" />
```

ReplyQueueName

Overview

`ReplyQueueName` is mapped to the MQ message descriptor's `ReplyToQ` field. It specifies the name of the reply message queue used by the port. When configuring an MQ client port this attribute is required if the clients expect replies to their requests. When configuring an MQ server port you can leave this attribute unset if you are sure that all clients are populating the `ReplyToQ` field in the message descriptor of their requests.

Server handling of ReplyQueueName

When a WebSphere MQ server receives a request, it first looks at the request's message descriptor's `ReplyToQ` field. If the request's message descriptor has `ReplyToQ` set, the server uses the reply queue specified in the message descriptor and ignores the `ReplyQueueName` setting. If the `ReplyToQ` field in the message descriptor is not set, the server will use the `ReplyQueueName` to determine where to send reply messages.

Example

Example 163 shows a WebSphere MQ server port that defaults to placing reply messages onto the queue `outbox`.

Example 163: *MQ Server with ReplyQueueName Set*

```
<mq:server QueueName="ether" QueueManager="leo"
           ReplyQueueName="outbox" ReplyQueueManager="pager" />
```

ReplyQueueManager

Overview

`ReplyQueueManager` is mapped to the MQ message descriptor's `ReplyToQMgr` field. It specifies the name of the WebSphere MQ queue manager that controls the reply message queue. When configuring an MQ client port this attribute is required if the clients expect replys to their requests. When configuring an MQ server port you can leave this attribute unset if you are sure that all clients are populating the `ReplyToQMgr` field in the message descriptor of their requests.

Server handling of ReplyQueueManager

When a WebSphere MQ server receives a request, it first looks at the request's message descriptor's `ReplyToQMgr` field. If the request's message descriptor has `ReplyToQ`Mgr set, the server uses the reply queue specified in the message descriptor and ignores the `ReplyQueueManager` setting. If the `ReplyToQMgr` field in the message descriptor is not set, the server will use the `ReplyQueueManager` to determine where to send reply messages.

Example

Example 164 shows a WebSphere MQ client port that is configured to receive replies from the server defined in Example 163 on page 490.

Example 164: *MQ Client with ReplyQueueName Set*

```
<mq:client QueueName="ether" QueueManager="leo"
           ReplyQueueName="outbox" ReplyQueueManager="pager" />
```

Server_Client

Overview

`Server_Client` specifies the type of WebSphere MQ installation on an application's host machine. If your application's host machine has a WebSphere MQ client installation, you must set this attribute to `client`. If the application's host machine has a WebSphere MQ server installation you do not need to set this attribute.

ModelQueueName

Overview

`ModelQueueName` is only needed if you are using dynamically created queues. It specifies the name of the queue from which the dynamically created queues are created.

AliasQueueName

Overview

When interoperating between WebSphere MQ applications whose queue managers are on different hosts, Artix requires that you specify the name of the remote queue to which the server will post reply messages. This ensures that the server will put the replies on the proper queue. Otherwise, the server will receive a request message with the `ReplyToQ` field set to a queue that is managed by a queue manager on a remote host and will be unable to send the reply.

You specify this server's local reply queue name in the WebSphere MQ client's `AliasQueueName` attribute when you define it in an Artix contract.

Effect of AliasQueueName

When you specify a value for `AliasQueueName` in a WebSphere MQ client port definition, you are altering how Artix populates the request message's `ReplyToQ` field and `ReplyToQMgr` field. Typically, Artix populates the reply queue information in the request message's message descriptor with the values specified in `ReplyQueueManager` and `ReplyQueueName`. Setting `AliasQueueName` causes Artix to leave `ReplytoQMgr` empty, and to set `ReplyToQ` to the value of `AliasQueueName`. When the `ReplyToQMgr` field of the message descriptor is left empty, the sending queue manager inspects the queue named in the `ReplyToQ` field to determine who its queue manager is and uses that value for `ReplyToQMgr`. The server puts the message on the remote queue that is configured as a proxy for the client's local reply queue.

Example

If you had a system defined similar to that shown in Figure 152, you would need to use the `AliasQueueName` attribute setting when configuring your WebSphere MQ client. In this set up the client is running on a host with a local queue manager `QMgrA`. `QMgrA` has two queues configured. `RqA` is a remote queue that is a proxy for `RqB` and `RplyA` is a local queue. The server is running on a different machine whose local queue manager is `QMgrB`.

QMgrB also has two queues. RqB is a local queue and RplyB is a remote queue that is a proxy for RplyA. The client places its request on RqA and expects replies to arrive on RplyA.

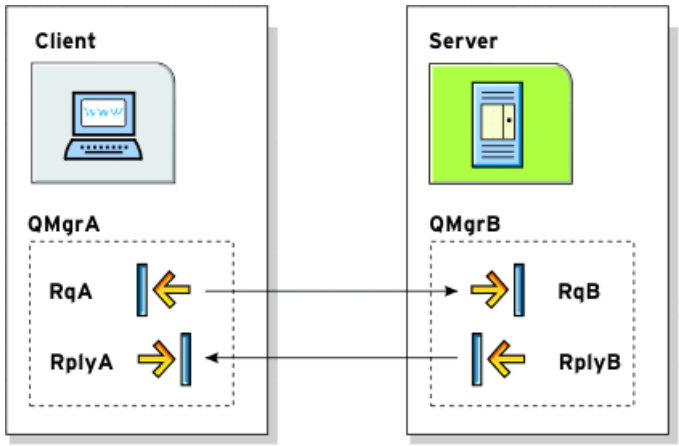

Figure 152:*MQ Remote Queues*

The Artix WebSphere MQ port definitions for the client and server for this deployment are shown in Example 165. AliasQueueName is set to RplyB because that is the remote queue proxying for the reply queue in server's local queue manager. ReplyQueueManager and ReplyQueueName are set to the client's local queue manager so that it knows where to listen for responses. In this example, the server's ReplyQueueManager and ReplyQueueName do not need to be set because you are assured that the client is populating the request's message descriptor with the needed information for the server to determine where replies are sent.

Example 165:*Setting Up WebSphere MQ Ports for Intercommunication*

```
<mq:client QueueManager="QMgrA" QueueName="RqA"
           ReplyQueueManager="QMgrA" ReplyQueueName="RplyA"
           AliasQueueName="RplyB"
           Format="string" Convert="true" />
<mq:server QueueManager="QMgrB" QueueName="RqB"
           Format="String" Convert="true" />
```

ConnectionName

Overview

`ConnectionName` specifies the name of the connection Artix uses to connect to its queue.

Note: If you set `CorrelationStyle` to `messageID copy` and specify a value for `ConnectionName` your system will not work as expected.

ConnectionReusable

Overview

`ConnectionReusable` specifies if the connection named in the `ConnectionName` field can be used by more than one application. Valid entries are `true` and `false`. Defaults to `false`.

ConnectionFastPath

Overview

`ConnectionFastPath` specifies if you want to load the request queue manager in process. Valid entries are `true` and `false`. Defaults to `false`.

Example

Example 166 shows a WebSphere MQ client port that loads its request queue manager in process.

Example 166: *WebSphere Client Port using ConnectionFastPath*

```
<mq:client QueueName="gate" QueueManager="dhd"
           ReplyQueueName="inbound" ReplyQueueManager="flipside"
           ConnectionFastPath="true" />
```

UsageStyle

Overview

`UsageStyle` specifies if a message can be queued without expecting a response. Valid entries are `peer`, `requester`, and `responder`. The default value is `peer`.

Attribute settings

The behavior of each setting is described in Table 22.

Table 22: *UsageStyle Settings*

Attribute Setting	Description
`peer`	Specifies that messages can be queued without expecting any response.
`requester`	Specifies that the message sender expects a response message.
`responder`	Specifies that the response message must contain enough information to facilitate correlation of the response with the original message.

Example

In Example 167, the WebSphere MQ client wants a response from the server and needs to be able to associate the response with the request that generated it. Setting the `UsageStyle` to responder ensures that the server's response will properly populate the response message descriptor's `CorrelID` field according to the defined correlation style. In this case, the correlation style is set to `correlationId`.

Example 167:*MQ Client with UsageStyle Set*

```
<mq:client QueueManager="postmaster" QueueName="eddie"
           ReplyQueueManager="postmaster" ReplyQueueName="fred"
           UsageStyle="responder"
           CorrelationStyle="correlationId" />
```

CorrelationStyle

Overview

`CorrelationStyle` determines how WebSphere MQ matches both the message identifier and the correlation identifier to select a particular message to be retrieved from the queue (this is accomplished by setting the corresponding `MQMO_MATCH_MSG_ID` and `MQMO_MATCH_CORREL_ID` in the `MatchOptions` field in `MQGMO` to indicate that those fields should be used as selection criteria).

The valid correlation styles for an Artix WebSphere MQ port are `messageId`, `correlationId`, and `messageId copy`.

Note: When a value is specified for `ConnectionName`, you cannot use `messageID copy` as the correlation style.

Attribute settings

Table 23 shows the actions of `MQGET` and `MQPUT` when receiving a message using a WSDL specified message ID and a WSDL specified correlation ID.

Table 23: *MQGET and MQPUT Actions*

Artix Port Setting	Action for MQGET	Action for MQPUT
`messageId`	Set the `CorrelId` of the message descriptor to `MessageID`.	Copy `MessageID` onto the message descriptor's `CorrelId`.
`correlationId`	Set `CorrelId` of the message descriptor to `CorrelationID`.	Copy `CorrelationID` onto message descriptor's `CorrelId`.
`messageId copy`	Set `MsgId` of the message descriptor to `messageID`.	Copy `MessageID` onto message descriptor's `MsgId`.

Example

Example 168 shows a WebSphere MQ client application that wants to correlate messages using the `messageID copy` setting.

Example 168:*MQ Client using messageID copy*

```
<mq:client QueueManager="grub" QueueName="gnome"
           ReplyQueueManager="lilo" ReplyQueueName="kde"
           CorrelationStyle="messageId copy" />
```

AccessMode

Overview

AccessMode controls the action of MQOPEN in the Artix WebSphere MQ transport. Its values can be peek, send, receive, receive exclusive, and receive shared. Each setting mapping corresponds to a WebSphere MQ setting for the MQOPEN. The default is receive.

Attribute settings

Table 24 describes the correlation between the Artix attribute settings and the MQOPEN settings.

Table 24: *Artix WebSphere MQ Access Modes*

Attribute Setting	Description
peek	Equivalent to MQOO_BROWSE. peek opens a queue to browse messages. This setting is not valid for remote queues.
send	Equivalent to MQOO_OUTPUT. send opens a queue to put messages into. The queue is opened for use with subsequent MQPUT calls.
receive (default)	Equivalent to MQOO_INPUT_AS_Q_DEF. receive opens a queue to get messages using a queue-defined default. The default value depends on the DefInputOpenOption queue attribute (MQOO_INPUT_EXCLUSIVE or MQOO_INPUT_SHARED).
receive exclusive	Equivalent to MQOO_INPUT_EXCLUSIVE. receive exclusive opens a queue to get messages with exclusive access. The queue is opened for use with subsequent MQGET calls. The call fails with reason code MQRC_OBJECT_IN_USE if the queue is currently open (by this or another application) for input of any type.

Table 24: *Artix WebSphere MQ Access Modes*

Attribute Setting	Description
`receive shared`	Equivalent to `MQOO_INPUT_SHARED`. `receive shared` opens queue to get messages with shared access. The queue is opened for use with subsequent `MQGET` calls. The call can succeed if the queue is currently open by this or another application with `MQOO_INPUT_SHARED`.

Example

Example 169 shows the settings for a WebSphere MQ server port that is set-up so that only one application at a time can access the queue.

Example 169: *WebSphere MQ Server setting AccessMode*

```
<mq:server QueueManager="welk" QueueName="anacani"
           ReplyQueueManager="severinsen" ReplyQueueName="johnny"
            AccessMode="recieve exclusive" />
```

Timeout

Overview

`Timeout` specifies the amount of time, in milliseconds, between a request and the corresponding reply before an error message is generated. If the reply to a particular request has not arrived after the specified period, it is treated as an error.

Example

Example 170 shows the settings for a MQ client port where replies are required in at most 3 minutes.

Example 170: *WebSphere MQ Client Port with a 3 Minute Timeout*

```
<mq:client QueueManager="jpl" QueueName="appollo"
           ReplyQueueManager="jpl" ReplyQueueName="mercury"
           Timeout="180000" />
```

MessageExpiry

Overview

`MessageExpiry` is mapped to the MQ message descriptor's `Expiry` field. It specifies message lifetime, expressed in tenths of a second. It is set by the Artix endpoint that puts the message onto the queue. The message becomes eligible to be discarded if it has not been removed from the destination queue before this period of time elapses.

The value is decremented to reflect the time the message spends on the destination queue, and also on any intermediate transmission queues if the put is to a remote queue. It may also be decremented by message channel agents to reflect transmission times, if these are significant.

`MessageExpiry` can also be set to `INFINITE` which indicates that the messages have unlimited lifetime and will never be eligible for deletion. If `MessageExpiry` is not specified, it defaults to `INFINITE` lifetime.

Example

Example 171 shows the settings for a WebSphere MQ client port where the messages sent from applications using this port have a lifetime of 30 minutes.

Example 171: *Client Port with a 3 Minute Message Lifetime*

```
<mq:client QueueManager="domino" QueueName="dot"
           ReplyQueueManager="domino" ReplyQueueName="cash"
           MessageExpiry="18000" />
```

MessagePriority

Overview

`MessagePriority` is mapped to the MQ message descriptor's `Priority` fileld. It specifies the message's priority. Its value must be greater than or equal to zero; zero is the lowest priority. If not specified, this field defaults to `priority normal`, which is 5. The special values for `MessagePriority` include `highest` (9), `high` (7), `medium` (5), `low` (3) and `lowest` (0).

Delivery

Overview

`Delivery` can be `persistent` or `not persistent`. `persistent` means that the message survives both system failures and restarts of the queue manager. Internally, this sets the `MQMD`'s `Persistence` field to `MQPER_PERSISTENT` or `MQPER_NOT_PERSISTENT`. The default value is `not persistent`. To support transactional messaging, you must make the messages `persistent`.

Example

Example 172 shows the settings for a WebSphere MQ port that sends persistent oneway messages.

Example 172:*Persistent WebSphere MQ Port*

```
<mq:client QueueManager="mointor" QueueName="msgQ"
           Delivery="persistent" />
```

Transactional

Overview

`Transactional` controls how messages participate in transactions and what role Webshpere MQ plays in the transactions.

Attribute settings

The values of this attribute are explained in Table 25.

Table 25: *Transactional Attribute Settings*

Attribute Setting	Description
`none (Default)`	The messages are not part of a transaction. No rollback actions will be taken if errors occur.
`internal`	The messages are part of a transaction with WebSphere MQ serving as the transaction manager.
`xa`	The messages are part of a transaction with WebSphere MQ serving as the resource manager.

Example

Example 173 shows the settings for a WebSphere MQ client port whose requests will be part of transactions managed by WebSphere MQ. Note that the `Delivery` attribute must be set to `persistent` when using transactions.

Example 173: *MQ Client setup to use Transactions*

```
<mq:client QueueManager="herman" QueueName="eddie"
           ReplyQueueManager="gomez" ReplyQueueName="lurch"
           UsageStyle="responder" Delivery="persistent"
           CorrelationStyle="correlationId"
           Transactional="internal" />
```

ReportOption

Overview

`ReportOption` is mapped to the MQ message descriptor's `Report` field. It enables the application sending the original message to specify which report messages are required, whether the application message data is to be included in them, and how the message and correlation identifiers in the report or reply message are to be set. Artix only allows you to specify one `ReportOption` per Artix port. Setting more than one will result in unpredictable behavior.

Attribute settings

The values of this attribute are explained in Table 26.

Table 26: *ReportOption Attribute Settings*

Attribute Setting	Description
`none (Default)`	Corresponds to `MQRO_NONE`. `none` specifies that no reports are required. You should never specifically set `ReportOption` to `none`; it will create validation errors in the contract.
`coa`	Corresponds to `MQRO_COA`. `coa` specifies that confirm-on-arrival reports are required. This type of report is generated by the queue manager that owns the destination queue, when the message is placed on the destination queue.
`cod`	Corresponds to `MQRO_COD`. `cod` specifies that confirm-on-delivery reports are required. This type of report is generated by the queue manager when an application retrieves the message from the destination queue in a way that causes the message to be deleted from the queue.

Table 26: *ReportOption Attribute Settings*

Attribute Setting	Description
`exception`	Corresponds to `MQRO_EXCEPTION`. `exception` specifies that exception reports are required. This type of report can be generated by a message channel agent when a message is sent to another queue manager and the message cannot be delivered to the specified destination queue. For example, the destination queue or an intermediate transmission queue might be full, or the message might be too big for the queue.
`expiration`	Corresponds to `MQRO_EXPIRATION`. `expiration` specifies that expiration reports are required. This type of report is generated by the queue manager if the message is discarded prior to delivery to an application because its expiration time has passed.
`discard`	Corresponds to `MQRO_DISCARD_MSG`. `discard` indicates that the message should be discarded if it cannot be delivered to the destination queue. An exception report message is generated if one was requested by the sender

Example

Example 174 shows the settings for a WebSphere MQ client that wants to be notified if any of its messages expire before they are delivered.

Example 174:*MQ Client Setup to Receive Expiration Reports*

```
<mq:client QueueManager="herman" QueueName="eddie"
           ReplyQueueManager="gomez" ReplyQueueName="lurch"
           ReportOption="expiration" />
```

Format

Overview

`Format` is mapped to the MQ message descriptor's `Format` field. It specifies an optional format name to indicate to the receiver the nature of the data in the message. The name may contain any character in the queue manager's character set, but it is recommended that the name be restricted to the following:

- Uppercase A through Z
- Numeric digits 0 through 9

Special values

`FormatType` can take the special values `none`, `string`, `event`, `programmable command`, and `unicode`. These settings are described in Table 27.

Table 27: *FormatType Attribute Settings*

Attribute Setting	Description
`none (Default)`	Corresponds to `MQFMT_NONE`. No format name is specified.
`string`	Corresponds to `MQFMT_STRING`. `string` specifies that the message consists entirely of character data. The message data may be either single-byte characters or double-byte characters.
`unicode`	Corresponds to `MQFMT_STRING`. `unicode` specifies that the message consists entirely of Unicode characters. (Unicode is not supported in Artix at this time.)
`event`	Corresponds to `MQFMT_EVENT`. `event` specifies that the message reports the occurrence of an WebSphere MQ event. Event messages have the same structure as programmable commands.

Table 27: *FormatType Attribute Settings*

Attribute Setting	Description
`programmable command`	Corresponds to `MQFMT_PCF`. `programmable command` specifies that the messages are user-defined messages that conform to the structure of a programmable command format (PCF) message. For more information, consult the IBM Programmable Command Formats and Administration Interfaces documentation at http://publibfp.boulder.ibm.com/epubs/html/csqzac03/csqzac030d.htm#Header_12.

When you are interoperating with WebSphere MQ applications host on a mainframe and the data needs to be converted into the systems native data format, you should set `Format` to `string`. Not doing so will result in the mainframe receiving corrupted data.

Example

Example 175 shows a WebSphere MQ client port used for making requests against a server on a mainframe system. Note that the `Convert` attribute is set to `true` signifying that WebSphere will convert the data into the mainframes native data mapping.

Example 175: *WebSphere MQ Client Talking to the Mainframe*

```
<mq:client QueueManager="hunter" QueueName="bigGuy"
           ReplyQueueManager="slate" ReplyQueueName="rusty"
           Format="string" Convert="true"/>
```

MessageId

Overview

`MessageId` is is mapped to the MQ message descriptor's `MsgId` field. It is an alphanumeric string of up to 20 bytes in length. Depending on the setting of `CorrelationStyle`, this string may be used to correlate request and response messages with each other. A value must be specified in this attribute if `CorrelationStyle` is set to `none`.

Example

Example 176 shows the settings for a WebSphere MQ client that wants to use message IDs to correlate response and request messages.

Example 176: *WebSphere MQ Client using MessageID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           CorrelationStyle="messageId" MessageID="foo"/>
```

CorrelationId

Overview

`CorrelationId` is mapped to the MQ message descriptor's `CorrelId` field. It is an alphanumeric string of up to 20 bytes in length. Depending on the setting of `CorrelationStyle`, this string will be used to correlate request and response messages with each other. A value must be specified in this attribute if `CorrelationStyle` is set to `none`.

Example

Example 177 shows the settings for a WebSphere MQ client that wants to use correlation Ids to correlate response and request messages.

Example 177: *WebSphere MQ Client using CorrelationID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
          CorrelationStyle="correlationId" CorrelationID="foo"/>
```

ApplicationData

Overview

`ApplicationData` specifies any application-specific information that needs to be set in the message header.

AccountingToken

Overview

`AccountingToken` is mapped to the MQ message descriptor's `AccountingToken` field. It specifies application-specific information used for accounting purposes.

Example

Example 178 shows the settings for a WebSphere MQ client used for making requests against a server on a mainframe system that keeps tracks of what department is using its resources.

Example 178: *WebSphere MQ Client Sending Accounting Token*

```
<mq:client QueueManager="hunter" QueueName="bigGuy"
           ReplyQueueManager="slate" ReplyQueueName="rusty"
           Format="string" Convert="true"
           AccountingToken="darkHorse" />
```

Convert

Overview

`Convert` specifies if messages are to be converted to the receiving system's native data format. Valid values are `true` and `false`. Default is `false`.

Note: The WebSphere MQ transport will always attempt to convert string data and always ignore non-string data. This setting is ignored.

Example

Example 179 shows a WebSphere MQ client port used for making requests against a server on a Unix system.

Example 179: *WebSphere MQ Client using Convert*

```
<mq:client QueueManager="atm5" QueueName="ReqQ"
           ReplyQueueManager="hpux1" ReplyQueueName="RepQ"
           Format="string" Convert="true"/>
```

ApplicationIdData

Overview

`ApplicationIdData` is mapped to the MQ message descriptor's `ApplIdentityData` field. It is application-specific string data that can be used to provide additional information about the message or the applciation from which it originated. This attribute is only valid when defining WebSphere MQ clients using an `<mq:client>` element.

ApplicationOriginData

Overview

`ApplicationOriginData` is mapped to the MQ message descriptor's `ApplOriginData` field. It is application-specific string data that can be used to provide additional information about the origin of the message.

Example

Example 180 shows the settings for a WebSphere MQ client that wants to identify itself to the server.

Example 180: *WebSphere MQ Client Sending Origin Data*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           ApplicationOriginData="SSLclient" />
```

UserIdentification

Overview

`UserIdentification` is mapped to the MQ message descriptor's `UserIdentifier` field. It is a string that represents the User ID of the application from which the message originated. This attribute is only valid when defining Websphere MQ clients using an `<mq:client>` element.

Example

Example 181 shows the settings for a WebSphere MQ client that needs to specify the User that is making the request.

Example 181: *WebSphere MQ Client Sending UserID*

```
<mq:client QueueManager="QM" QueueName="reqQueue"
           ReplyQueueManager="RQM" ReplyQueueName="RepQueue"
           UserIdentification="tux" />
```

Tibco Transport Extensions

Artix provides a number of attributes used in defining a TIB/RV service.

Port attributes

Table 28 lists the Artix contract elements used to describe a TIB/RV port.

Table 28: *TIB/RV Transport Properties*

Attribute	Explanation
`tibrv:port`	Indicates that the port uses the TIB/RV transport.
`tibrv:port@serverSubject`	A required element that specifies the subject to which the server listens. This parameter must be the same between client and server.
`tibrv:port@clientSubject`	Specifies the subject that the client listens to. The default is to use the transport inbox name. This parameter only affects clients.
`tibrv:port@bindingType`	Specifies the message binding type.
`tibrv:port@callbackLevel`	Specifies the server-side callback level when TIB/RV system advisory messages are received.
`tibrv:port@responseDispatchTimeout`	Specifies the client-side response receive dispatch timeout.

Table 28: *TIB/RV Transport Properties*

Attribute	Explanation
tibrv:port@transportService	Specifies the UDP service name or port for TibrvNetTransport.
tibrv:port@transportNetwork	Specifies the binding network addresses for TibrvNetTransport.
tibrv:port@transportDaemon	Specifies the TCP daemon port for the TibrvNetTransport.
tibrv:port@transportBatchMode	Specifies if the TIB/RV transport uses batch mode to send messages.
tibrv:port@cmSupport	Specifies if Certified Message Delivery support is enabled.
tibrv:port@cmTransportServerName	Specifies the server's TibrvCmTransport correspondent name.
tibrv:port@cmTransportClientName	Specifies the client TibrvCmTransport correspondent name.
tibrv:port@cmTransportRequestOld	Specifies if the endpoint can request old messages on start-up.
tibrv:port@cmTransportLedgerName	Specifies the TibrvCmTransport ledger file.
tibrv:port@cmTransportSyncLedger	Specifies if the endpoint uses a synchronous ledger.
tibrv:port@cmTransportRelayAgent	Specifies the endpoint's TibrvCmTransport relay agent.
tibrv:port@cmTransportDefaultTimeLimit	Specifies the default time limit for a Certified Message to be delivered.
tibrv:port@cmListenerCancelAgreements	Specifies if Certified Message agreements are canceled when the endpoint disconnects.
tibrv:port@cmQueueTransportServerName	Specifies the server's TibrvCmQueueTransport correspondent name.
tibrv:port@cmQueueTransportClientName	Specifies the client's TibrvCmQueueTransport correspondent name.

Table 28: *TIB/RV Transport Properties*

Attribute	Explanation
tibrv:port@cmQueueTransportWorkerWeight	Specifies the endpoint's TibrvCmQueueTransport `worker weight`.
tibrv:port@cmQueueTransportWorkerTasks	Specifies the endpoint's TibrvCmQueueTransport `worker tasks` parameter.
tibrv:port@cmQueueTransportSchedulerWeight	Specifies the TibrvCmQueueTransport `scheduler weight` parameter.
tibrv:port@cmQueueTransportSchedulerHeartbeat	Specifies the endpoint's TibrvCmQueueTransport `scheduler heartbeat` parameter.
tibrv:port@cmQueueTransportSchedulerActivation	Specifies the TibrvCmQueueTransport `scheduler activation` parameter.
tibrv:port@cmQueueTransportCompleteTime	Specifies the TibrvCmQueueTransport `complete time` parameter.

tibrv:port@bindingType

`tibrv:port@bindingType` specifies the message binding type. TIB/RV Artix ports support three types of payload formats as described in Table 29.

Table 29: *TIB/RV Supported Payload formats*

Setting	Payload Formats	TIB/RV Message Implications
`msg`	TibrvMsg	The top-level messages will have fields of type `TIBRVMSG_STRING`. The value of each field is the name of a WSDL part name from the corresponding WSDL message. If the WSDL part is a primitive type then the value of this type is put against the name of the WSDL part. If the WSDL part is a complex type then a nested TibrvMsg is created and added against the WSDL part name.
`xml`	SOAP, tagged data	The message data is encapsulated in a field of `TIBRVMSG_XML` with a null name and an ID of `0`.
`opaque`	fixed record length data, variable record length data	The message data is encapsulated in a field of `TIBRVMSG_OPAQUE` with a null name and an ID of `0`.

tibrv:port@callbackLevel

`tibrv:port@callbackLevel` specifies the server-side callback level when TIB/RV system advisory messages are received. It has three settings:

- `INFO`
- `WARN`
- `ERROR` (default)

This parameter only affects servers.

tibrv:port@responseDispatchTimeout

`tibrv:port@responseDispatchTimeout` specifies the client-side response receive dispatch timeout. The default is `TIBRV_WAIT_FOREVER`. Note that if only the TibrvNetTransport is used and there is no server return response for a request, then not setting a timeout value causes the client to block forever. This is because client has no way to know whether any server is processing on the sending subject or not. In this case, we recommend that `responseDispatchTimeout` is set.

tibrv:port@transportService

`tibrv:port@transportService` specifies the UDP service name or port for TibrvNetTransport. If empty or omitted, the default is `rendezvous`. If no corresponding entry exists in `/etc/services`, `7500` for the `TRDP` daemon, or `7550` for the `PGM` daemon will be used. This parameter must be the same for both client and server.

tibrv:port@transportNetwork

`tibrv:port@transportNetwork` specifies the binding network addresses for TibrvNetTransport. The default is to use the interface IP address of the host for the `TRDP` daemon, `224.0.1.78` for the `PGM` daemon. This parameter must be interoperable between the client and the server.

tibrv:port@transportDaemon

`tibrv:port@transportDaemon` specifies the TCP daemon port for TibrvNetTransport. The default is to use `7500` for the `TRDP` daemon, or `7550` for the `PGM` daemon.

tibrv:port@transportBatchMode

`tibrv:port@transportBatchMode` specifies if the TIB/RV transport uses batch mode to send messages. The default is `false` which specifies that the endpoint will send messages as soon as they are ready. When set to `true`, the endpoint will send its messages in timed batches.

tibrv:port@cmSupport

`tibrv:port@cmSupport` specifies if Certified Message Delivery support is enabled. The default is `false` which disables CM support. Set this parameter to `true` to enable CM support.

Note: When CM support is disabled all other CM properties are ignored.

tibrv:port@cmTransportServerName

`tibrv:port@cmTransportServerName` specifies the server's TibrvCmTransport correspondent name. The default is to use a transient correspondent name. This parameter must be the same for both client and server if the client also uses Certified Message Delivery.

tibrv:port@cmTransportClientName

`tibrv:port@cmTransportClientName` specifes the client's TibrvCMTransport correspondent name. The default is to use a transient correspondent name.

tibrv:port@cmTransportRequestOld

`tibrv:port@cmTransportRequestOld` specifies if the endpoint can request old messages on start-up. requestOld parameter. The default is `false` which disables the endpoint's ability to request old messages when it starts up. Setting this property to `true` enables the ability to request old messages.

tibrv:port@cmTransportLedgerName

`tibrv:port@cmTransportLedgerName` specifes the file name of the endpoint's TibrvCMTrasnport ledger. The default is to use an in-process ledger that is stored in memory.

tibrv:port@cmTransportSyncLedger

`tibrv:port@cmTransportSyncLedger` Specifies if the endpoint uses a synchronous ledger. `true` specifies that the endpoint uses a synchronous ledger. The default is `false`.

tibrv:port@cmTransportRelayAgent

`tibrv:port@cmTransportRelayAgent` Specifies the endpoint's TibrvCmTransport relay agent. If this property is not set, the endpoint does not use a relay agent.

tibrv:port@cmTransportDefaultTimeLimit

`tibrv:port@cmTransportDefaultTimeLimit` specifies TibrvCmTransport message default time limit. The default is that no message time limit will be set.

tibrv:port@cmListenerCancelAgreements

`tibrv:port@cmListenerCancelAgreements` specifies if the TibrvCmListener cancels Certified Message agreements when the endpoint disconnects. parameter. If set to `true`, CM agreements are cancelled when the endpoint disconnects. The default is `false`.

tibrv:port@cmQueueTransportServerName

`tibrv:port@cmQueueTransportServerName` specifies the server's TibrvCmQueueTransport correspondent name. If this property is set, the server listener joins to the distributed queue of the specified name. This parameter must be the same among the server queue members.

tibrv:port@cmQueueTransportClientName

`tibrv:port@cmQueueTransportClientName` specifies the client's TibrvCmQueueTransport correspondent name. If this property is set, the client listener joins to the distributed queue of the specifies name. This parameter must be the same among all client queue members.

Note: If distributed queue is enabled on the client side, the transport does not handle any request-response semantics. This is for load-balanced polling-type clients, e.g. one client in the distributed queue periodically invokes an operation that only has outputs and no input, and one listener in the group processes the response.

tibrv:port@cmQueueTransportWorkerWeight

`tibrv:port@cmQueueTransportWorkerWeight` specifies the endpoint's TibrvCmQueueTransport `worker weight`. The default is `TIBRVCM_DEFAULT_WORKER_WEIGHT`.

tibrv:port@cmQueueTransportWorkerTasks

`tibrv:port@cmQueueTransportWorkerTasks` specifies the endpoint's TibrvCmQueueTransport `worker tasks` parameter. The default is `TIBRVCM_DEFAULT_WORKER_TASKS`.

tibrv:port@cmQueueTransportSchedulerWeight

`tibrv:port@cmQueueTransportSchedulerWeight` specifies the TibrvCmQueueTransport `scheduler weight` parameter. The default is `TIBRVCM_DEFAULT_SCHEDULER_WEIGHT`.

tibrv:port@cmQueueTransportSchedulerHeartbeat

`tibrv:port@cmQueueTransportSchedulerHeartbeat` specifies the TibrvCmQueueTransport `scheduler heartbeat` parameter. The default is `TIBRVCM_DEFAULT_SCHEDULER_HB`.

tibrv:port@cmQueueTransportSchedulerActivation	`tibrv:port@cmQueueTransportSchedulerActivation` Specifies the TibrvCmQueueTransport `scheduler activation` parameter. The default is `TIBRVCM_DEFAULT_SCHEDULER_ACTIVE`.
tibrv:port@cmQueueTransportCompleteTime	`tibrv:port@cmQueueTransportCompleteTime` specifies the TibrvCmQueueTransport `complete time` parameter. The default is `0`.

Glossary

B

Binding

A binding associates a specific protocol and data format to 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. A collection 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 (binding) information for that interface. In the context of the Artix Designer, this contract is referred to as a *Resource*.

A contract contains two components: logical and physical. The logical contract defines things that are independent of the underlying transport and wire format: 'portType', 'Operation', 'Message', 'Type', and 'Schema.'

The physical contract defines the wire format, middleware transport, and service groupings, as well as the mapping between the `portType` 'operations' and wire formats, and the buffer layout for fixed formats and extensors, The physical contract defines: 'Port,' 'Binding' and 'Service.'

CORBA

CORBA (Common Object Request Broker Architecture) defines standards for interoperability and portability among distributed objects, independently of the language in which those objects are written. It is a robust, industry-accepted standard from the OMG (Object Management Group), deployed in thousands of mission critical systems.

CORBA also specifies an extensive set of services for creating and managing distributed objects, accessing them by name, storing them in persistent stores, externalizing their state, and defining ad hoc relationships between them. An ORB is the core element of the wider OMG framework for developing and deploying distributed components.

E

End-point

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 end-point can be compiled into an application).

Extensible Style Sheet Transformation

A set of extensions to the XML style sheet language that describes transformations between XML documents. For more information see the XSLT specification.

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 port type, which allows for a logical contract to have multiple bindings and thus multiple wire message formats for the same contract.

Message

A WSDL message is an abstract definition of the data being communicated. Each part of a message is associated with defined types. A WSDL message is analogous to a parameter in object-oriented programming.

O

Operation

A WSDL operation is an abstract definition of the action supported by the service. It is defined in terms of input and output messages. An operation is loosely analogous to a function or method in object-oriented programming, or a message queue or business process.

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 file using the binding definition.

Port Type

A WSDL port type is a collection of abstract operations, supported by one or more endpoints. A port type is loosely analogous to a class in object-oriented programming. A port type can be mapped to multiple transports using multiple bindings.

Protocol

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

R

Resource

A resource can be one of two things:

- A WSDL file that defines the interface of your Artix solution
- A Schema that defines one or more types. This schema can be a stand alone resource or it can define the types within a WSDL contract.

Resources are contained within collections. There can be one or more resources in a collection, and the resources can either be specific to that collection, or shared across several collections (shared resources).

Resources are created either from scratch using the Resource Editor wizards and dialogs to define them, or are based on an existing files. For example, you can use the Designer to convert an IDL file into WSDL.

Resource Editor

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

Routing

The redirection of a message from one WSDL binding to another. Routing rules apply to an end-point, and the specification of routing rules is required for a some Artix services. Artix supports topic-, subject- and content-based routing. Topic- and subject-based routing rules can be fully expressed in the WSDL contract. However, content-based routing rules may need to be placed in custom handlers (C plug-ins). Content-based routing handler plug-ins are dynamically loaded.

Router

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

S

Service

An Artix service is instance of an Artix runtime deployed with one or more contracts, but no generated language bindings (contrast this with end-point). The service acts as a daemon that 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.

SOAP

SOAP is an XML-based messaging framework specifically designed for exchanging formatted data across the Internet. It can be used for sending request and reply messages or for sending entire XML documents. As a protocol, SOAP is simple, easy to use, and completely neutral with respect to operating system, programming language, or distributed computing platform.

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

The implementation of an Artix WSDL service contract.

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' property in of an Artix contract.

Type

A WSDL data type is a container for data type definitions that is used to describe messages (for example an XML schema).

W

Web Services Description Language

An XML based specification for defining Web services. For more information see the WSDL specification.

Workspace

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

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

WSDL

WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data binding formats. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. Hence, a WSDL document uses the following elements in the definition of network services:

- **Types**—a container for data type definitions using some type system.
- **Message—**an abstract, typed definition of the data being communicated.
- **Operation—**an abstract definition of an action supported by the service.
- **Port Type—**an abstract set of operations supported by one or more endpoints.
- **Binding**—a concrete protocol and data format specification for a particular port type.
- **Port**—a single endpoint defined as a combination of a binding and a network address.
- **Service—**a collection of related endpoints.

Source: Web Services Description Language (WSDL) 1.1. W3C Note 15 March 2001. (http://www.w3.org/TR/wsdl)

X

XML

XML is a simpler but restricted form of SGML (Standard General Markup Language). The markup describes the meaning of the text. XML enables the separation of content from data. XML was created so that richly structured documents could be used over the web.

XSD

XML Schema Definition (XSD) is the language used to define an XML Schema. The XML Schema defines the structure of an XML document.

In Artix, a schema can be a standalone resource within a collection, or it can be used as an import do define the types within a WSDL contract.

Index

T

U

V

W

X