Artix™

Artix Connect User's Guide

Version 3.0, June 2005

Making Software Work Together™

Updated: 15-Jun-2005

Contents

List of Figures

Preface

Artix Connect is a .NET custom remoting channel that enables transparent communication between clients that are running in a Microsoft .NET environment and servers using any of the transports and protocols supported by Artix, including:

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

In addition, Artix Connect supports all of the bindings (marshalling schemes) supported by Artix, including

- SOAP
- CORBA Common Data Representation (CDR)
- Pure XML
- Fixed record length (FRL)*
- Tagged (variable record length)*
- TibrvMsg (a TIBCO Rendevous format)*
- Tuxedo Field Manipulation Language (FML)*

Note: To use any of the transports, protocols and bindings marked with a *, you must have a license for Artix Advanced.

Artix Connect is designed to allow .NET programmers to use any .NET language (for example, Visual Basic .NET, C#, J#, and so on) to easily access services running in Windows, UNIX, or OS/390 environments that have been described in Artix WSDL contracts. It enables .NET programmers to use the tools familiar to them to build heterogeneous systems that use both .NET and any of the middleware platforms supported by Artix.

What is Covered in this Guide

This book describes how to use Artix Connect in a .NET environment.

Who Should Read this Guide

This guide is intended for .NET application programmers who want to use Artix Connect to develop and deploy distributed applications that can communicate with any of the middleware platforms supported by Artix.

This guide assumes that the reader already has a working knowledge of .NET-based tools, such as Visual Basic .NET and C#.

The reader does not need an in-depth knowledge of Artix or WSDL concepts to use Artix Connect. However, some knowledge would help, particularly with more complex WSDL contracts. The following Artix guides are a good place to start learning:

- Getting Started with Artix
- Designing Artix Solutions

In addition, the following may provide useful background information:

- *Understanding Web Services: XML, WSDL, SOAP, and UDDI*, written by Eric Newcomer, published by Addison Wesley, ISBN 0-201-75081-3.
- *Understanding SOA with Web Services*, written by Eric Newcomer and Greg Lomow, published by Addison Wesley, ISBN 0-321-18086-0.
- The W3C XML Schema page at: www.w3.org/XML/Schema.
- The W3C WSDL specification at: www.w3.org/TR/wsdl.

Required Versions

To use Artix Connect, you need at least Microsoft .NET Framework 1.1 and Microsoft Visual Studio .NET 2003 installed on your machine.

Organization of this Guide

This guide is divided as follows:

Chapter 1, "Introduction to Artix Connect"

This chapter introduces Artix Connect, its system components and some usage models.

Chapter 2, "Getting Started"

This chapter gets you up and running quickly with Artix Connect by walking you though a simple demo application.

Chapter 3, "Developing .NET Clients"

This chapter helps to get you up and running quickly with application programming with Artix Connect. It explains the basics you need to know to develop a simple .NET client, written in C#, which can invoke on an existing Web service.

Chapter 4, "Client Callbacks"

.NET clients can implement some of the functionality associated with servers, and all servers can act as clients. A callback invocation is a programming technique that takes advantage of this. This chapter describes how to implement client callbacks.

Chapter 5, "Development Support Tools"

This chapter describes the Artix Connect Web service wizard and the `wsdltodotnet` command-line utility.

Chapter 6, "Deploying an Artix Connect Application"

This chapter provides an overview of the deployment model you can adopt when deploying a distributed application with Artix Connect. It also describes the steps you must follow to deploy a distributed Artix Connect application.

Chapter 7, "Introduction to WSDL"

Although you do not need to understand WSDL in any great detail to use Artix Connect, understanding the basics can help. This chapter introduces basic WSDL concepts.

Chapter 8, "WSDL to .NET Mapping"

WSDL types are defined in XML, and .NET types are defined in Microsoft Intermediate Language (MSIL). To allow interworking between .NET clients and Web services, .NET clients must be presented with metadata that

describes the interfaces exposed by the Web service. When using .NET Remoting, the .NET types must use the .NET Common Type System (CTS). This chapter outlines how Artix Connect maps WSDL-to-.NET CTS.

Chapter 9, "Configuration"

This chapter describes the environment variables that are specific to Artix Connect, and their associated values.

Additional Resources

Knowledge base

The IONA knowledge base (http://www.iona.com/support/knowledge_base/index.xml) contains helpful articles, written by IONA experts, about Artix Connect and other IONA products.

Update center

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

Support

If you need help with Artix Connect or any other IONA product, contact IONA at: support@iona.com.

Documentation feedback

Comments on IONA documentation can be sent to: docs-support@iona.com.

Newsgroup

The IONA newsgroup and discussion forums provide feedback and answers to questions about IONA products:

http://www.iona.com/products/newsgroups.htm

Typographical conventions

This book uses the following typographical and keying conventions:

`Fixed width`	Fixed 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>`
`Fixed width italic`	Fixed width italic words or characters in code and commands represent variable values that you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`YourUserName`*
Italic	Italic words in normal text represent *emphasis* and *new terms*.
Bold	Bold words in normal text represent graphical user interface components such as menu commands and dialog boxes (for example, the **User Preferences** dialog.)

Keying conventions

This guide may use the following keying conventions:

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

CHAPTER 1

Introduction to Artix Connect

Artix Connect is a custom .NET remoting channel that enables transparent communication between clients that are running in a Microsoft .NET environment and services deployed on any of the middleware platforms supported by Artix.

In this chapter

This chapter discusses the following topics:

Artix Connect Overview

Overview

This section provides an introductory overview of Artix Connect in terms of how it facilitates communication between .NET clients and any of the middleware platforms supported by Artix.

In this section

The following topics are discussed:

- What is Artix Connect?
- Graphical Overview of Role
- WSDL contract
- Supported Transports, Protocols, and Bindings

What is Artix Connect?

The Artix Connect is a custom .NET remoting channel, referred to as `Artix.Remoting`. Its purpose is to support application integration across network boundaries, different operating systems, and different programming languages. Specifically, it provides a high performance bridge that enables .NET clients to communicate with servers using any of the transports, protocols, and bindings (marshalling schemes) supported by Artix.

Graphical Overview of Role

Figure 1 provides a conceptual overview of how Artix Connect facilitates the integration of .NET clients and the middleware platforms supported by Artix:

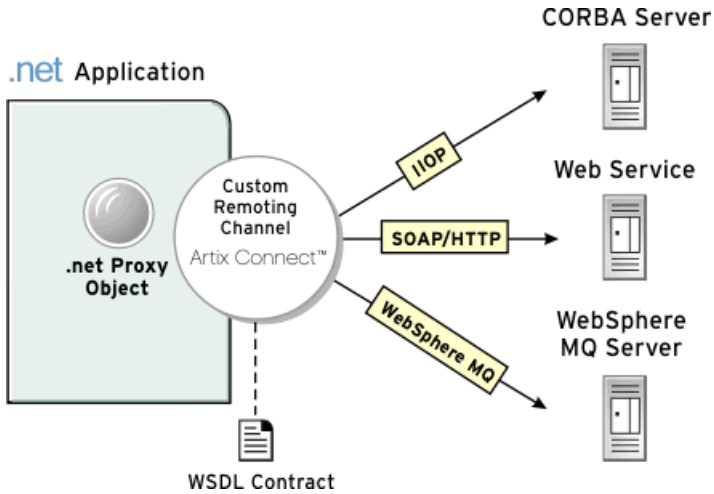

Figure 1: *Artix Connect Overview*

WSDL contract

To connect your .NET client to any of the middleware platforms supported by Artix, all Artix Connect requires is the WSDL contract for that service.

Artix uses Web Services Description Language (WSDL) contracts to express the logical interaction between services. With Artix, IONA has taken WSDL beyond simple SOAP over HTTP Web services by extending the features of WSDL to model diverse enterprise systems in a technology neutral way.

It separates the service from its underlying middleware mechanism, and allows the service to be invoked over an optimized connection using existing transport mechanisms such as WebSphere MQ (previously known as MQSeries) and Tuxedo.

The main elements of an Artix WSDL contract are as follows:

- *Port types*—a port type defines remotely callable operations that have parameters and return values.
- *Types*—user defined data types used to describe messages.

- *Binding*—a binding describes how to encode all of the operations and data types associated with a particular port type. A binding is specific to a particular protocol; for example, SOAP or CORBA.
- *Port definitions*—a port contains endpoint data that enables clients to locate and connect to a remote server; for example, a CORBA port might contain a stringified IOR.

For a basic introduction to WSDL, see "Introduction to WSDL" on page 69.

For more information about Artix and WSDL, see the Artix 3.0 documentation, available online at:
http://www.iona.com/support/docs/artix/3.0/index.xml

Supported Transports, Protocols, and Bindings

A key feature of Artix Connect is that it supports all of the transports, protocols that Artix supports, including:

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

In addition, Artix Connect supports all of the bindings (marshalling schemes) supported by Artix, including

- SOAP
- CORBA Common Data Representation (CDR)
- Pure XML
- Fixed record length (FRL)*
- Tagged (variable record length)*
- TibrvMsg (a TIBCO Rendevous format)*
- Tuxedo Field Manipulation Language (FML)*

The same binding can be used by multiple protocols or a binding can be used by only one protocol.

Note: To use any of the transports, protocols and bindings marked with a *, you must have a license for Artix Advanced.

Artix Connect System Components

Overview

This section describes the various components that comprise an Artix Connect system. The following topics are discussed:

- Bridge
- .NET client
- Artix service

Bridge

The bridge is a synonym for Artix Connect itself. It is implemented as a custom .NET remoting channel, referred to as `Artix.Remoting`. It is implemented in a mixture of managed and unmanaged DLLs. This channel uses a dynamic marshaller and the WSDL contract to formulate dynamic requests that can be invoked on the service defined in the WSDL contract. The bridge provides the mappings and performs the necessary translation between .NET common type system (CTS) and WSDL types.

The bridge is used in conjunction with the Artix Connect Wizard, which generates .NET metadata from a WSDL contract, from within the Microsoft Visual Studio .NET 2003 development environment.

.NET client

A .NET client can use Artix Connect to communicate with any service described in an Artix WSDL contract. This client can be written in any language compatible with .NET, including Visual Basic .NET, Visual C++, C#, J#, and Jscript.

Artix service

Any service that has been defined in an Artix WSDL contract can be contacted by .NET clients, using Artix Connect.

Artix Connect Usage Scenarios

Overview

Artix Connect can be used to connect .NET clients to any middleware platform supported by Artix, once the back-end service is defined in a WSDL contract.

In this section

This section gives an overview of two such scenarios:

.NET Client Invoking on Web service using SOAP over HTTP

Overview

This subsection describes a scenario in which Artix Connect connects a .NET client to a Web service using SOAP over HTTP. It discusses the following topics:

- Graphical overview
- Web service
- WSDL contract
- .NET client and Artix Connect
- Using a transport other than SOAP over HTTP
- Demo

Graphical overview

Figure 2 is a graphical overview of this usage model:

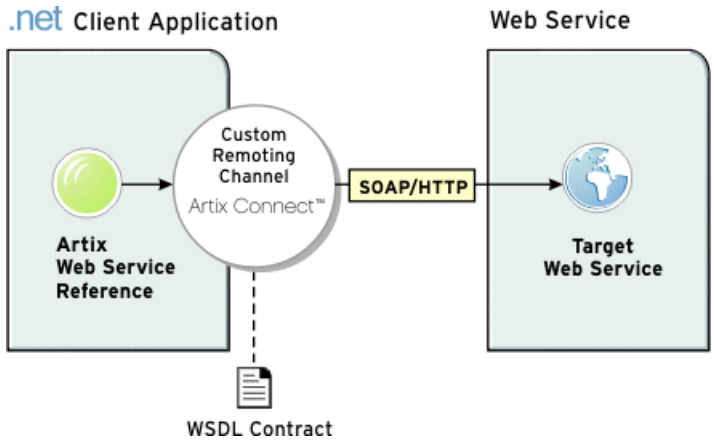

Figure 2: *.NET client invoking on SOAP over HTTP Web Service*

Web service

The Web service can be any SOAP over HTTP Web service. In this case, it is implemented in C++, using Artix. The advantage of using Artix is that clients can use the enhanced quality of services that it provides; for example, callbacks.

For more detail on using Artix to develop a SOAP over HTTP Web service, see the Artix documentation on the IONA documentation website.

WSDL contract

The types and protocols that can be used to contact the Web service are contained in its WSDL contract. In this case, the Artix Designer, which is part of the Artix product, is used to design the WSDL contract.

For more details on using Artix to design WSDL contracts, see the Designing Artix Solutions guide.

.NET client and Artix Connect

Artix Connect provides a dynamic bridge for .NET in the form of a custom remoting channel, referred to as `Artix.Remoting`. The .NET client loads this bridge in-process (that is, in the client's address space). Artix Connect uses the transport and protocol details contained in the WSDL file to communicate between the .NET client machine and SOAP over HTTP Web service. The WSDL file is the only thing required by Artix Connect to enable the .NET client to successfully invoke on the Web service. No changes are required on the server side.

The .NET client registers the `Artix.Remoting` custom remoting channel. The .NET client then creates a proxy for the remote service. The .NET client can subsequently make calls on this proxy as if it were a local .NET object. The proxy uses the `Artix.Remoting` channel to make a corresponding call on the target Web service.

Artix Connect provides a Web service wizard that generates .NET metadata from the WSDL contract from within the Microsoft Visual Studio .NET 2003 development environment. The `Artix.Remoting` channel exposes the mapped .NET types as metadata contained in a .NET assembly, allowing automatic mapping of .NET object references to the interfaces and object references defined in the WSDL file at runtime.

The client does not need to know that the target object is, for example, a SOAP over HTTP Web service. A .NET client can be written in Visual Basic, C#, J#, C++ or any language that supported by .NET.

Using a transport other than SOAP over HTTP

If required, the deployed .NET client can use different transports and protocols; for example, if the SOAP over HTTP transport preforms too slowly in a deployed system. You can simply change the WSDL file to reflect the new transport details and Artix Connect takes care of the rest. You do not need to make any changes to the client.

Demo

Artix Connect includes a demo that illustrates a .NET client invoking on a SOAP over HTTP Web service. It is located in:

```
ArtixConnectInstallDir/artix/Version/demos/dotnet/hello_world
```

For details on how to run this demo, see the `README.txt` file in the demo directory.

.NET Client Invoking on a CORBA Server using IIOP

Overview

This subsection describes a scenario in which Artix Connect connects a .NET client to a CORBA server. It discusses the following topics:

- Graphical overview
- CORBA server
- WSDL contract
- .NET client and Artix Connect
- Demo

Graphical overview

Figure 3 is a graphical overview of this usage model:

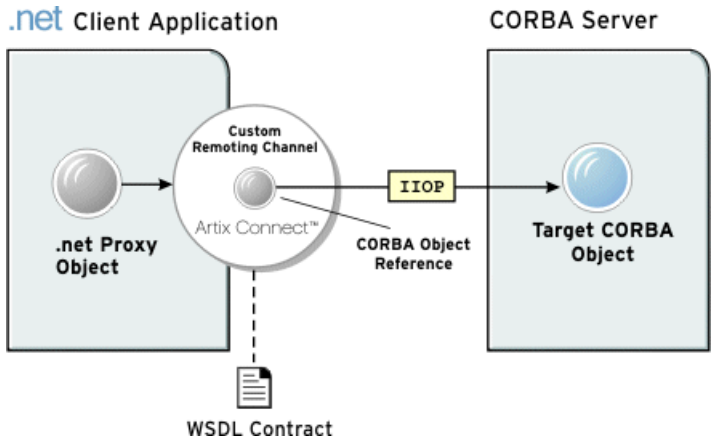

Figure 3: *.NET client invoking on a CORBA server over IIOP*

CORBA server

The server can be any CORBA-compliant server. In this case it is implemented in C++ using Orbix. No changes are required on the server side.

For more detail on CORBA and Orbix, see the Orbix documentation, available on the IONA documentation website.

WSDL contract

The CORBA server's interface is specified in a CORBA IDL file. The Artix Designer, which is part of the Artix product, is used to generate an Artix WSDL contract from the IDL file. The WSDL contract specifies that clients should communicate with the server using IIOP. In addition, the WSDL contract contains details of the CORBA server's location (IOR, `corbaname` or `corbaloc`).

For more detail on how to use Artix to expose a CORBA service as a Web service, see the Artix for CORBA guide.

.NET client and Artix Connect

Artix Connect provides a dynamic bridge for .NET in the form of a custom remoting channel, referred to as `Artix.Remoting`. The .NET client loads this remoting channel in-process (that is, in the client's address space). Artix Connect uses the transport and protocol details contained in the WSDL contract to communicate between the .NET client machine and the CORBA server. The WSDL file is the only thing required by Artix Connect to enable the .NET client to successfully invoke on the CORBA server. No changes are required on the server side.

The .NET client registers the `Artix.Remoting` custom remoting channel and creates a proxy for the remote object. The .NET client can subsequently make calls on this proxy as if it were a local .NET object. The proxy uses the `Artix.Remoting` channel to make a corresponding call on the target object.

Artix Connect provides a Web service wizard that generates .NET metadata from the WSDL contract from within the Microsoft Visual Studio .NET 2003 development environment. The `Artix.Remoting` channel exposes the mapped .NET types as metadata contained in a .NET assembly, allowing automatic mapping of .NET object references to the interfaces and object references defined in the WSDL file at runtime.

The client does not need to know that the target object is, for example, a CORBA object. A .NET client can be written in Visual Basic, C#, J#, C++ or any language supported by .NET.

Demo

Artix Connect includes a demo that illustrates a .NET client invoking on a CORBA server. It is located in:

```
ArtixConnectInstallDir/artix/Version/demos/dotnet/corba_grid
```

For details on how to run this demo, see the `README.txt` file in the demo directory.

CHAPTER 2

Getting Started

This chapter focuses on getting started with Artix Connect. It walks you through a simple Hello World demo that shows you how a Web service can be invoked from a standard C# .NET client using Artix Connect.

In this chapter

This chapter contains the following sections:

Introduction

Overview

This chapter is based on running Artix Connect `Hello World` demo. It shows how you use Artix Connect to connect a .NET client to a SOAP over HTTP Artix Web service.

In this section

This section gives details of the prerequisites to running the demo and provides some basic details. The following topics are covered:

- Prerequisites
- Demo location
- Running from the command line

Prerequisites

The Artix Connect demos are designed to run on Windows only.

In addition, you must have Microsoft Visual Studio .NET 2003 installed into the default location on your Windows system.

Demo location

The demo can be found in:

```
ArtixConnectInstallDir\artix\Version\demos\dotnet\hello_world
```

Running from the command line

This chapter details how you can build and run the demo from within the Visual Studio .NET 2003 development environment. You can, however, also build and run the demo from the command line. For details, see the `README.txt` file in the demo directory.

Running the Hello World Demo

Overview

To run the `Hello World` demo from within the Microsoft Visual Studio .NET 2003 development environment, complete the following steps:

Step	Action
1	Set Artix Connect environment
2	Select the Artix Connect Demos
3	Build the demo
4	Run the server
5	Run the client

Set Artix Connect environment

The Artix Connect installer sets the environment variables for you. If, however, you chose not set the environment variables while installing the product, you must set them manually before building and running the demo. See "Configuration" on page 111 for more detail.

Select the Artix Connect Demos

From the Windows **Start** menu, select the Artix Connect 3.0 **Demos**, as shown in Figure 4:

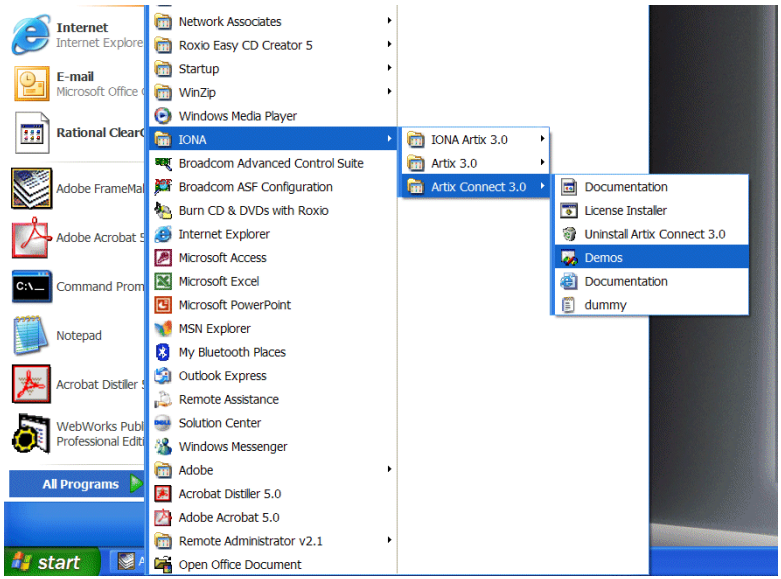

Figure 4: *Selecting Artix Connect Demos*

The demos load into the Visual Studio .NET 2003 development environment as shown in Figure 5. In the example shown the `README_DOTNET.txt` file is selected. This is a high-level readme that comes with the demos.

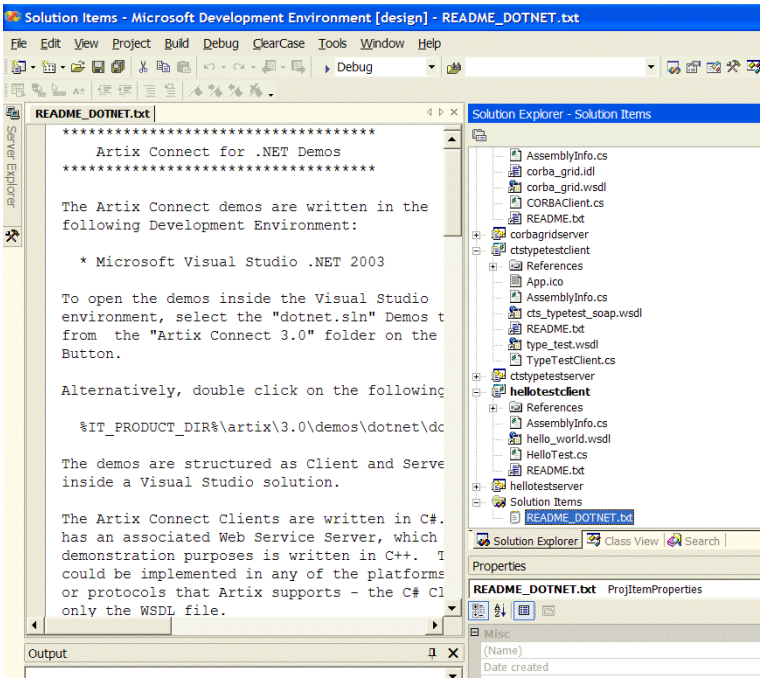

Figure 5: *Artix Connect Demos Loaded into Visual Studio .NET 2003*

Build the demo

To build the demos, select **Build|Build Solution**, as shown in Figure 6:

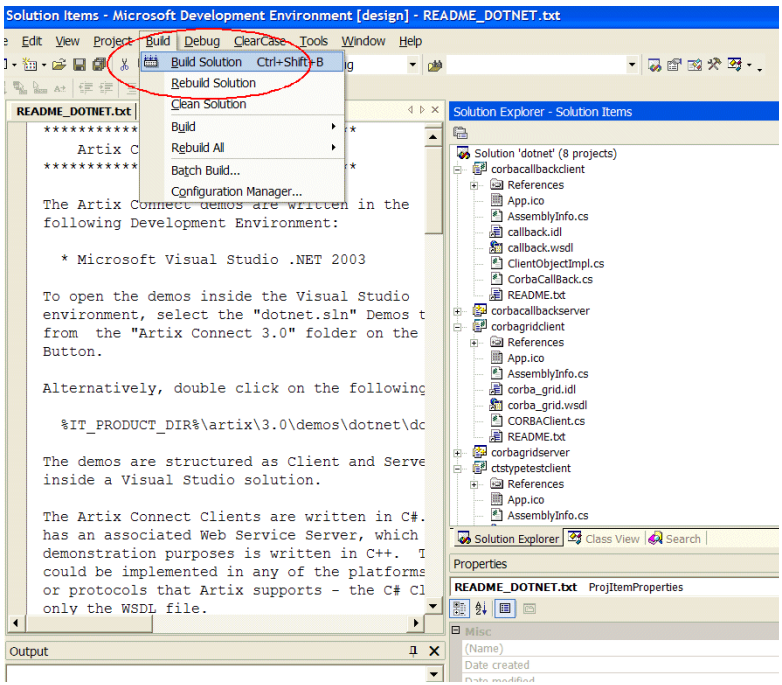

Figure 6: *Building Demos from Visual Studio .NET 2003*

Run the server

To run the server, complete the following steps:

1. Right-click on the `hellotestserver` icon and select **Set as StartUp Project**, as shown in Figure 7:

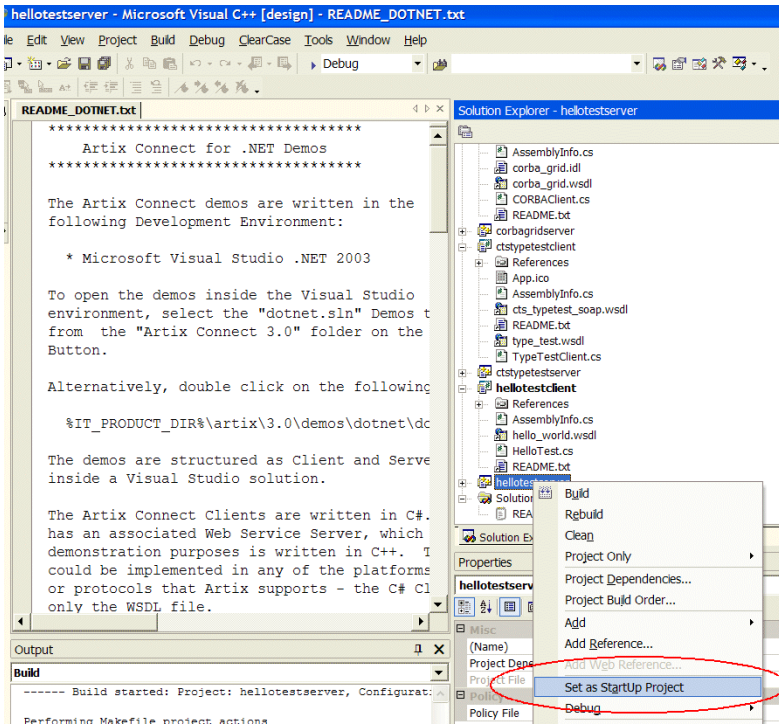

Figure 7: *Running the Hello World Server—Set as StartUp Project*

2. Select **Debug|Start Without Debugging**, as shown in Figure 8:

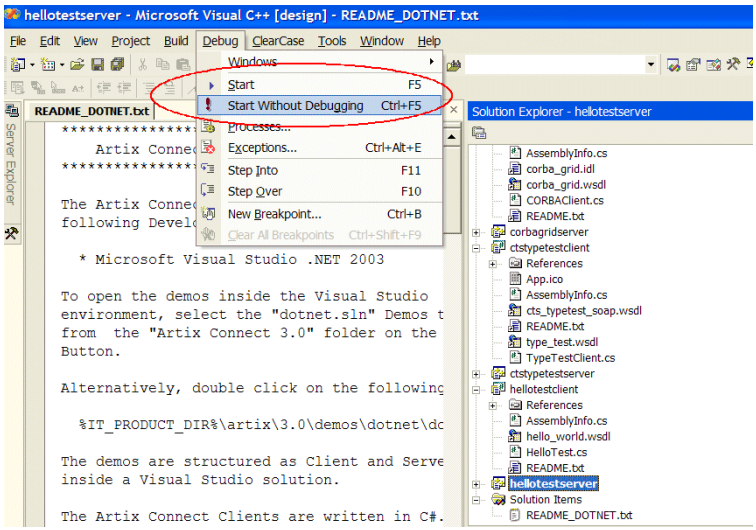

Figure 8: *Running the Hello World Server—Start Without Debugging*

The server will open in a new DOS command window and output Server Ready to the screen.

Run the client

To run the client, complete the following steps:

1. Right-click on the `hellotestclient` icon and select **Set as StartUp Project**, as shown in Figure 9:

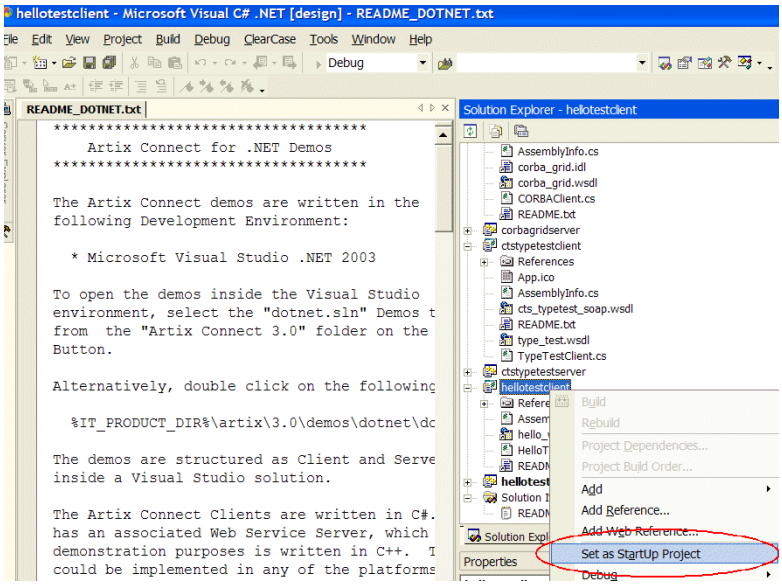

Figure 9: *Running Hello World Client—Set as StartUp Project*

2. Select **Debug|Start Without Debugging**, as shown in Figure 10:

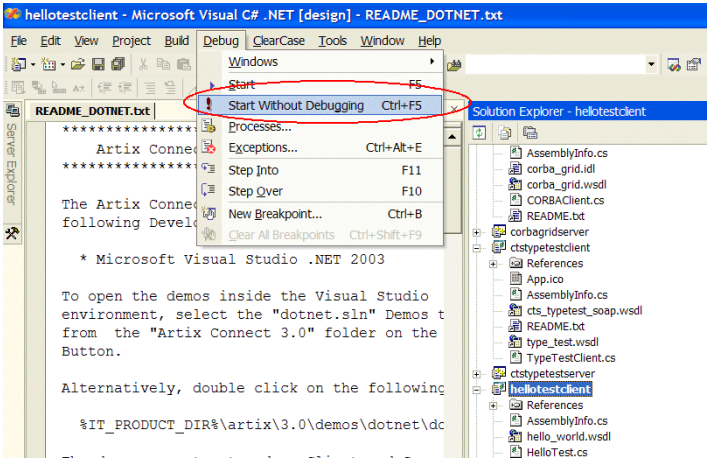

Figure 10: *Running the Hello World Client—Start Without Debugging*

The client starts in a new DOS command window, invokes on the server and outputs `Hello .NET Connector` to the screen.

Background Information

Overview

This section describes what happens when the demo runs and provides some background information on the `Hello World` demo files. The following topics are covered:

- What happens when the demo runs
- Server
- Client
- WSDL contract
- Using other transports and protocols

What happens when the demo runs

When the `Hello World` server process starts, it starts to listen for SOAP over HTTP requests and outputs `Server Ready` to the screen. When the `Hello World` client application starts, it reads the `hello_world.wsdl` contract, which is located in:

```
ArtixConnectInstallDir\artix\Version\demos\dotnet\hello_world\
etc
```

The WSDL contract contains details of the types and protocols that can be used by the client to contact the Web service, as well as details of the location of the Web service.

Server

The Web service is implemented in C++ and was developed using Artix.

For more information on Artix development, see the Artix 3.0 library.

Client

The Artix Connect Web service wizard was used to generate the type information required by the .NET client to invoke on the Web service. All it required was the WSDL contract; in this case, `hello_world.wsdl`. It generated a `Greeter.dll` .NET assembly, which contains the type information, and client starting point code in a `Greeter.cs` file. Application logic was added to the `Greeter.cs` file.

For more information on developing .NET clients, see "Developing .NET Clients" on page 27.

WSDL contract

The `hello_world.wsdl` contract contains all the information required by the .NET C# client to invoke on the Web service successfully. It is located in:

```
ArtixConnectInstallDir\artix\Version\demos\dotnet\hello_world\
etc
```

It was designed using the Artix Designer, which is a GUI that ships with Artix. The WSDL file specifies that clients should communicate with the server using SOAP/HTTP in the following XML fragment:

```
...
<wsdl:service name="SOAPService">
   <wsdl:port binding="tns:Greeter_SOAPBinding" name="SoapPort">
      <soap:address location="http://localhost:9000"/>
        <http-conf:client/>
        <http-conf:server/>
   </wsdl:port>
</wsdl:service>
```

For more information on designing Artix WSDL contracts, see the Designing Artix Solutions guide.

Using other transports and protocols

The .NET C# client can use any of the transports and protocols supported by Artix, including:

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

Note: To use any of the transports and protocols marked with a *, you must have a valid Artix Advanced license.

The .NET client only requires the WSDL contract. Therefore, by simply editing the contents of the WSDL file if, for example, the SOAP/HTTP transport performed too slowly in a deployed system, or the enterprise qualities of service features provided by a different transport are required

and it proves necessary to change the server, the transports and protocols used by deployed C# clients can be changed by simply changing the contents of the WSDL contract.

CHAPTER 3

Developing .NET Clients

This chapter explains how to develop a simple .NET client, written in C#, which can invoke on an existing Artix Web service using SOAP over HTTP.

In this chapter

This chapter discusses the following topics:

Prerequisites

Overview

This section describes the prerequisites to starting application development with Artix Connect. The following topics are discussed:

- Required versions
- Client-side requirements
- Server-side requirements
- Adding Artix Connect to the Global Assembly Cache

Required versions

To use the Artix Connect runtime, you need at least Microsoft .NET Framework 1.1 installed on your machine. To use Artix Connect for development, you need Microsoft Visual Studio .NET 2003 installed on your machine.

Client-side requirements

Ensure that Artix Connect is installed and configured correctly. See the Artix Connect Installation Guide for details.

Server-side requirements

Artix Connect requires no changes to existing services. All it needs is access to the WSDL contract that defines the service.

This guide assumes that you do not have to design the WSDL contract. It is assumed that the WSDL contract is provided for you. If, however, you need to know how to design an Artix WSDL contract for a new or existing service, see the Designing Artix Solutions guide.

Adding Artix Connect to the Global Assembly Cache

Artix Connect is implemented as a custom remoting channel in managed C++. This custom remoting channel is called `Artix.Remoting` and is contained in the `Artix.Remoting.dll` assembly. To use the `Artix.Remoting` channel, the .NET framework must be able to obtain and access the `Artix.Remoting.dll` assembly from either of the following:

- The directory from which the client program is run.
- The Global Assembly Cache (GAC).

By default, `Artix.Remoting` is registered with the GAC during the installation of Artix Connect.

Developing .NET Clients

Overview

This section describes how to develop a .NET client that can invoke on Artix service using Artix Connect. The `Hello World` demo is used as an example application. The `Hello World` demo shows a C# .NET client invoking on an Artix Web service, using SOAP over HTTP. It is located in:

```
ArtixConnectInstallDir/artix/Version/demos/dotnet/hello_world
```

In this section

This section discusses the steps that you must complete to develop a .NET client that can connect to an Artix Web service. The steps are:

Generating .NET Metadata from a WSDL file Using the GUI

Overview

The first task in implementing a .NET client that can communicate with a server that supports any of the transports and protocols supported by Artix, is to generate the .NET metadata that describes the target service interface. .NET metadata is required so that .NET applications that are to make invocations on remote objects can be compiled, and to allow .NET to create proxy objects.

Ordinarily, when .NET applications are communicating with each other, the metadata for .NET objects can be found as part of the .NET assembly. However, this is not the case for Artix services. Artix Connect includes a GUI, the Artix Connect Wizard, which enables you to generate .NET metadata and client starting point code from an Artix WSDL contract from within the Microsoft Visual Studio .NET 2003 development environment.

In this section

This section walks you through the steps to generating .NET metadata and client starting point code from a WSDL contract using the Artix Connect Wizard.

Note: This guide assumes that the WSDL contract already exists and that you have been provided with it as a starting point.

For more information on using Artix to develop WSDL contracts, see the Designing Artix Solutions guide.

Using the Artix Connect Wizard

To generate .NET metadata from within the Microsoft Visual Studio .NET 2003 development environment, using the Artix Connect Wizard, do the following:

1. Select **File | New | Project** to start a new project as shown in Figure 11:

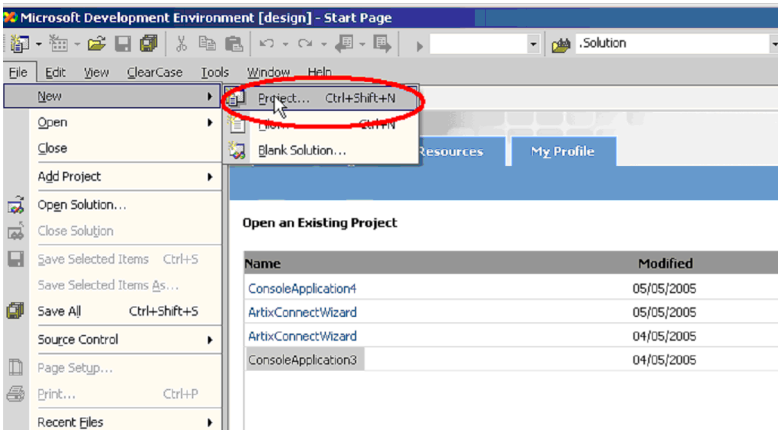

Figure 11: *Creating a New Project*

2. The **New Project** dialog box appears as shown in Figure 12. Select the project type that you want to create—in this case, a Visual C# project using the Empty Project template:

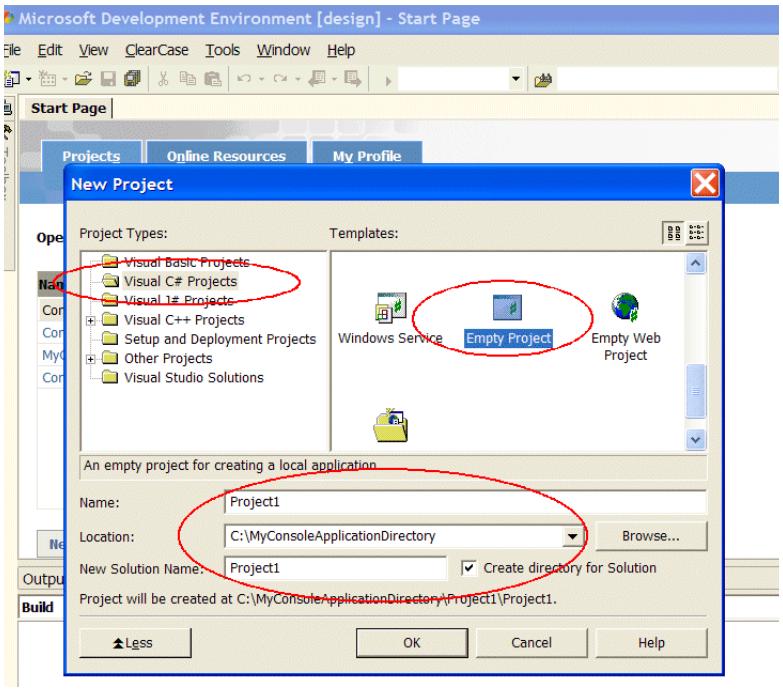

Figure 12: *Starting a New Project*

Note: The Artix Connect GUI supports C# console projects only. For projects that do not use the console or use other languages, you should use the `wsdltodotnet` command-line utility to generate the .NET metadata for you. See "wsdltodotnet Command-line Utility" on page 63 for more detail.

3. Enter a name for your project and a directory into which you want your project to be stored.

4. Click **OK**. The Visual Studio .NET 2003 Development Environment creates a C# project, as shown in Figure 13:

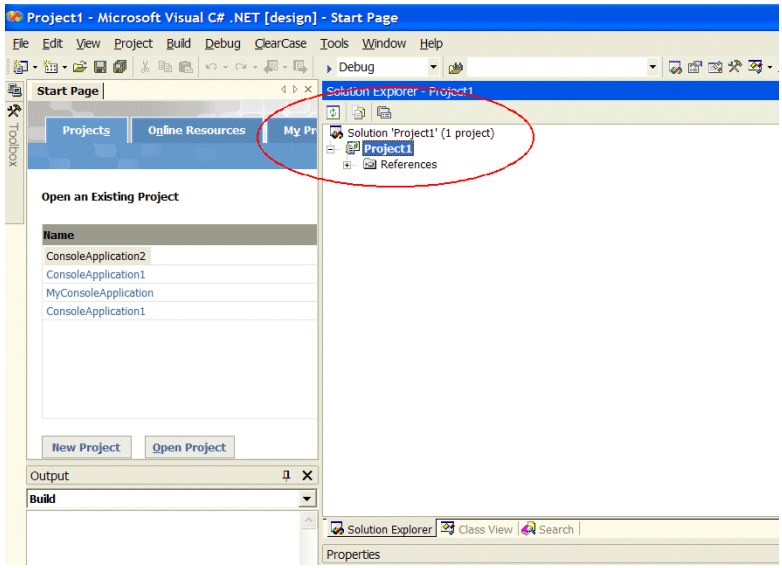

Figure 13: *C# Project*

5. Next you need to add the server WSDL file to the project. To do this select **File | Add New Item**, as shown in Figure 14, to launch the **Add New Item** dialog box:

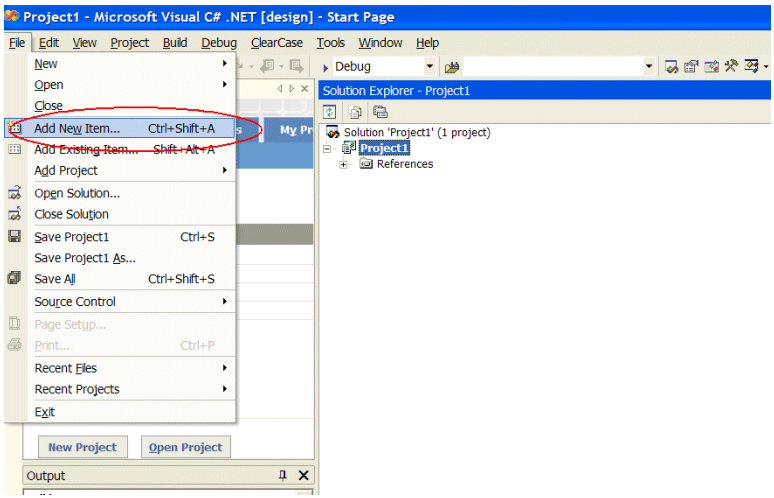

Figure 14: *Launching the Add New Item Dialog Box*

6. The **Add New Item** dialog box appears as shown in Figure 15. Select the **IONA Artix Web Service** wizard and click **Open**:

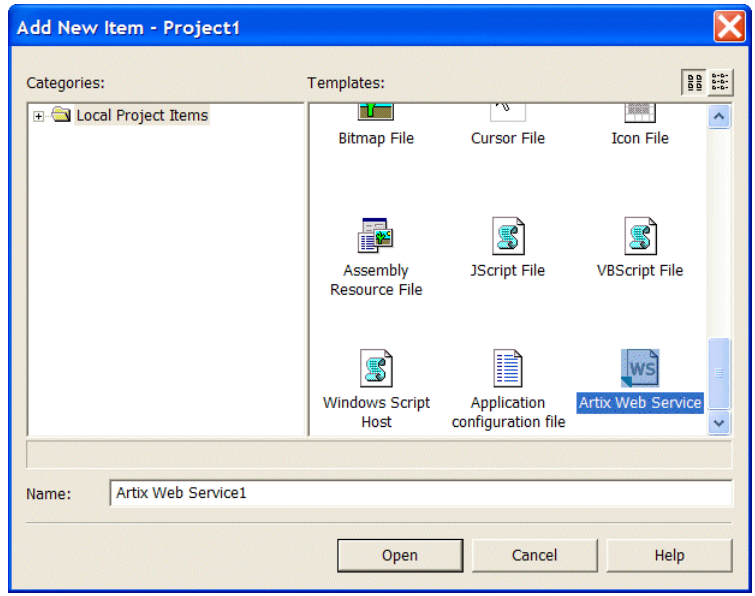

Figure 15: *Launching the Artix Connect Wizard*

7. The **Artix Connect Wizard** appears as shown in Figure 16. Click the **Select** button and browse for the WSDL contract associated with the Artix service to which you want the client to connect. In this example, select the `hello_world.wsdl` file, located in *`ArtixInstallDir`*`\Artix\`*`Version`*`\demos\dotnet\hello_world\etc`

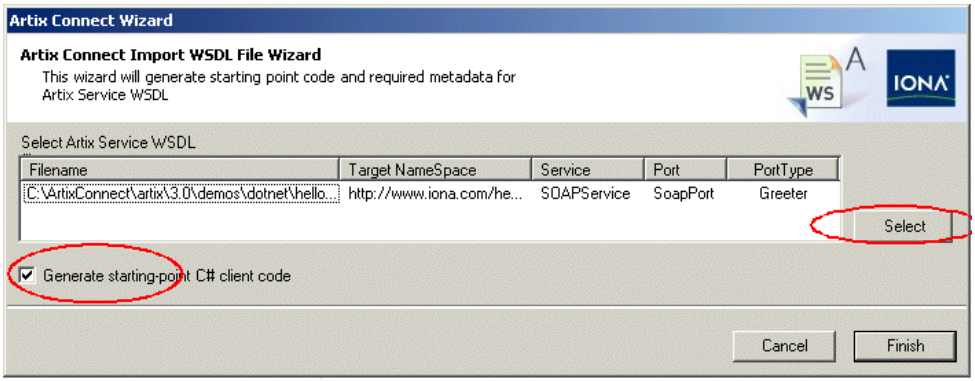

Figure 16: *Selecting WSDL File Using Artix Connect Wizard*

The **Artix Connect Wizard** fills in the `Filename`, `Target NameSpace`, `Service`, `Port`, and `PortType` fields with values taken from the WSDL contract. You should verify that the service selected is the one you want. The **Generate starting-point C# client code** check box is selected by default.

8. Click **Finish** to import the WSDL file and generate client starting point code for this service.

The **Artix Connect Wizard** adds three required items to the client project, as shown in Figure 17):

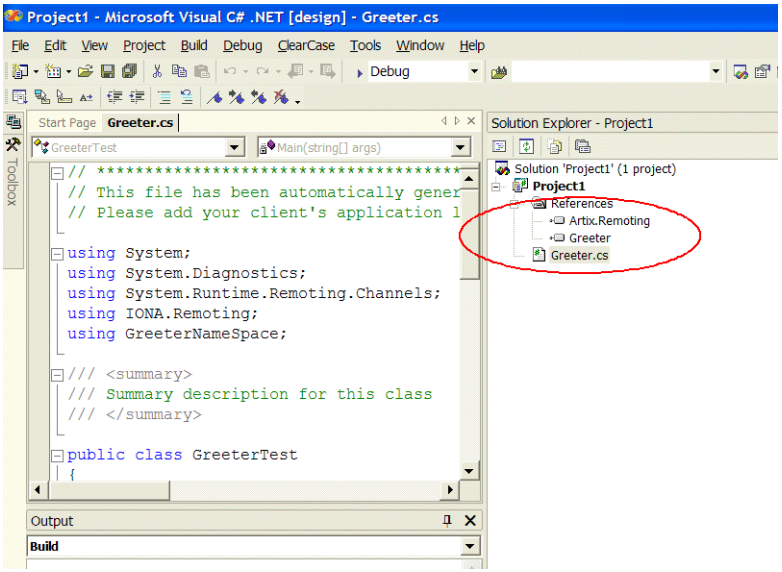

Figure 17: *Required Files Added to Project by Artix Connect Wizard*

It adds the following references:

- The `Artix.Remoting` assembly, which is required at runtime by all Artix Connect clients.
- The *`PortType_Name`*`.dll` metadata assembly, which has been generated by the `wsdltodotnet` command-line tool, and contains the type information for the server. In this example, the file in called `Greeter.dll`.

And the following file:

- Client starting point code in a `.cs` file—in this case, `Greeter.cs`. This is where you add your client application code.

Writing a C# Client

Overview

The next task in implementing a .NET client that can communicate with an Artix Web service is to write the C# client. As shown in the previous subsection, the Artix Connect Wizard generates a client mainline with starting point code. In this example, the file is called `Greeter.cs` and is shown in Figure 18. You simply uncomment the relevant line of client application code and add the client logic.

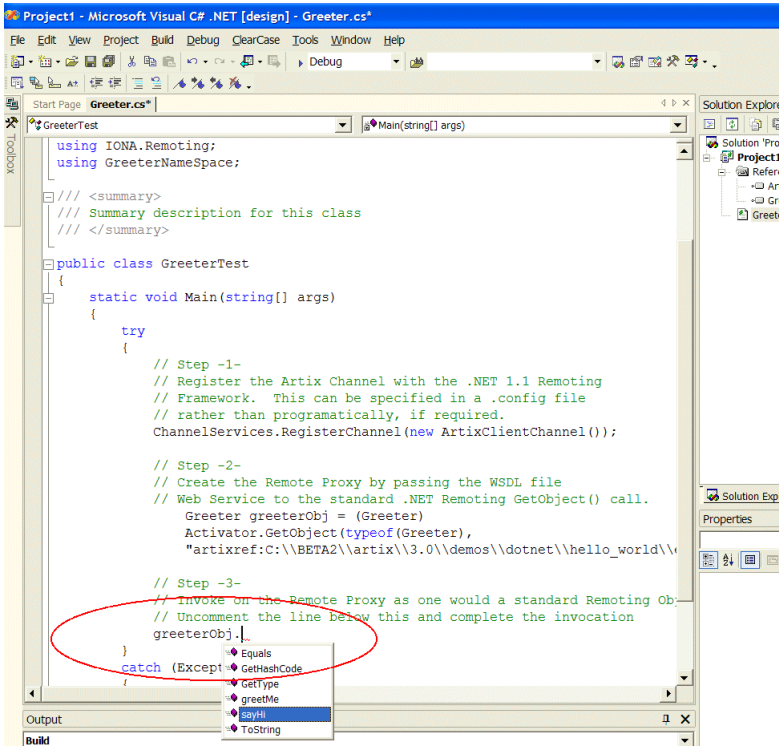

Figure 18: *Greeter.cs*

In this subsection

This subsection walks you through the code, which:

1. Registers the remoting channel
2. Creates a remote proxy
3. Invokes on remote proxy

Registers the remoting channel

The following line registers the remoting channel that the client wants to use. The custom remoting channel should be registered in the same way as any other .NET remoting channel.

```
// C#
ChannelServices.RegisterChannel(new ArtixClientChannel());
```

The preceding code tells the .NET application that when it is attempting to access an object outside of its application domain, it should use the `ArtixClientChannel` remoting channel.

Note: If you use the `wsdltodotnet` command-line utility to generate the .NET metadata, you must add the `Artix.Remoting.dll` and the *`PortType_Name`*`.dll` metadata assembly, which contains the type information for the server, to your project. You can do this by right-clicking on your project and selecting the Add References option. Select the `Artix.Remoting.dll` from the list that appears and select the generated *`PortType_Name`*`.dll` by browsing to the location where you have it stored.

Creates a remote proxy

The following code creates a proxy instance of the remote target object in the client's address space:

Example 1: *Creating a remote proxy*

```
//C#
//GetObject() call.
Greeter greeterObj = (Greeter),
   Activator.GetObject(typeof (Greeter),
  "artixref:C:\\Program Files\\artix\\3.0\\
   demos\\dotnet\\hello_world\\etc\\hello_world.wsdl
   http://www.iona.com/hello_world_soap_http
   SOAPService SoapPort");
```

1. The call to `GetObject()` specifies the .NET type that corresponds to the name of the target object to which the client wants to connect (in this case, `Greeter`).
2. It also specifies an Artix reference, which points the client to the WSDL contract that defines the service that it wants to connect to. It is made up of four parts, each separated by a space and all specified on one line. The parts are:
   i. The location and name of the WSDL contract—in this example, the `hello_world.wsdl`, which is located in `ArtixInstallDir\artix\Version\demos\dotnet\hello_world\` etc.
   ii. The target namespace—in this example, `http://www.iona.com/hello_world_soap_http`. This is taken from the WSDL contract.
   iii. The name of the service that the client wants to use—in this example, `SOAPService`. This is taken from the WSDL contract.
   iv. The name of the port that the client wants to use—in this example, `Greeter`. This is taken from the WSDL contract.

Invokes on remote proxy

To complete the client you need to uncomment the code that invokes on the remote proxy—in this case, `greeterObj`—and add the client logic. For example, you can have the client invoke on the remote proxy `greetMe()` operation and have the client print the response to the screen by adding the code shown below:

```
// C#
     String response;
     response = greeterObj.greetMe(".NET Connector");
     Console.WriteLine(response);
```

Building and Running the Client

Overview

This subsection describes how to build the client that you wrote in the previous subsection.

Building the client

To build the client, select **Build | Build Solution**, as shown in Figure 19:

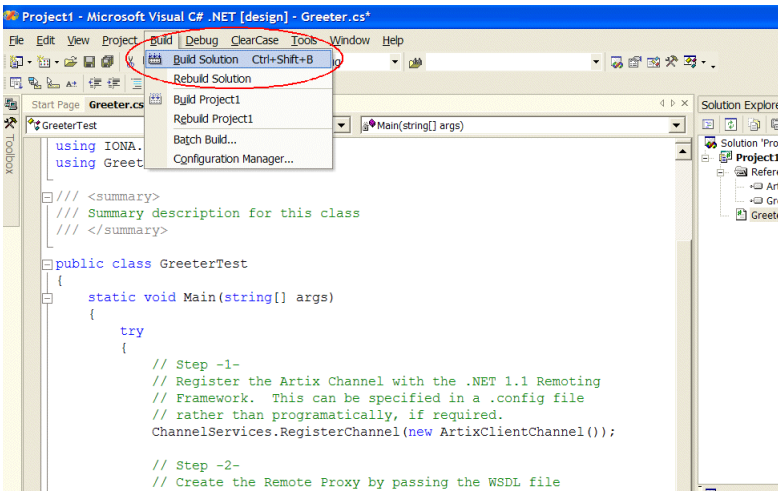

Figure 19: *Building the Client*

Running the client

To run the client successfully, you must:

1. Start the server. In this case you can use the server that is provided with the `Hello World` demo. To open the demo solution, from the **File** menu select **Open Solution**, as shown in Figure 20:

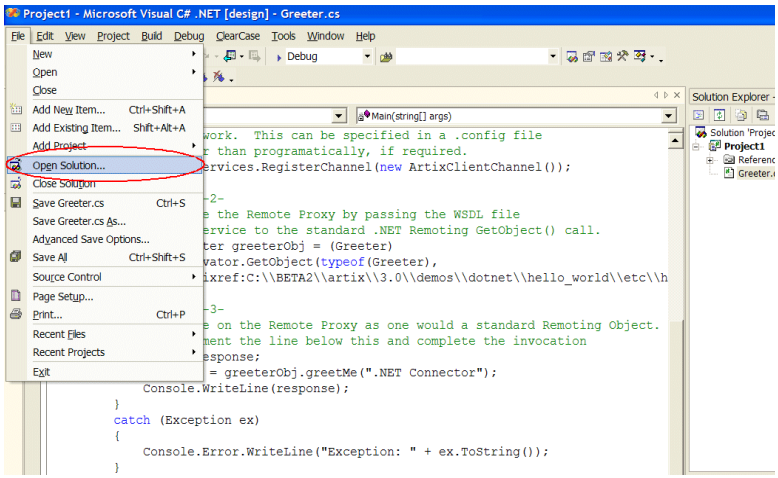

Figure 20: *Opening the Hello World Demo Solution*

2. The Open Solution dialog box appears as shown in Figure 21:

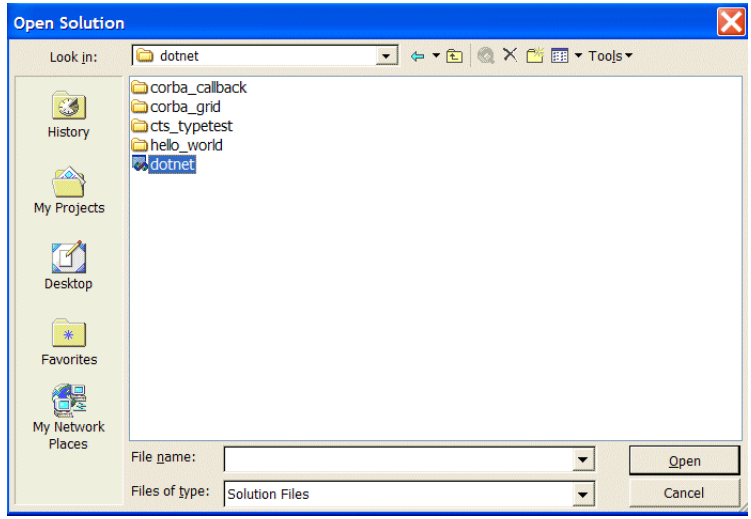

Figure 21: *Opening Demo Solution*

3. Select the `dotnot` solution file, as shown in Figure 21, and click **Open**.
4. Follow the instructions for running the server in "Run the server" on page 19.

5. Reopen your client project and run the client by selecting **Debug|Start Without Debugging**, as shown in Figure 22:

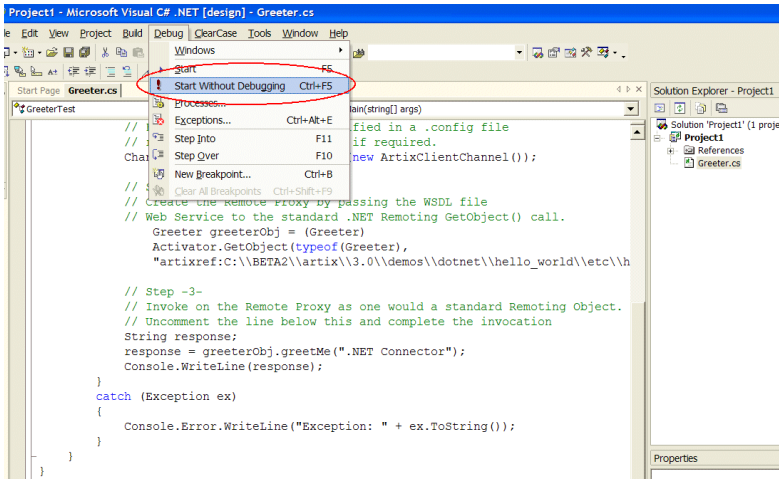

Figure 22: *Running the Client*

6. The client starts in a new DOS command window, invokes on the server and prints `Hello .NET Connector` to the screen.

CHAPTER 4

Client Callbacks

.NET clients can implement some of the functionality associated with servers, and all servers can act as clients. A callback invocation is a programming technique that takes advantage of this. This chapter describes how to implement client callbacks.

In this chapter

This chapter discusses the following topics:

Introduction to Callbacks

Overview

This section introduces the concept of client callbacks. The following topics are discussed:

- What is a callback?
- Typical use

What is a callback?

A callback is an operation invocation made from a server to an object that is implemented in a client. A callback allows a server to send information to clients without forcing clients to explicitly request the information.

Typical use

Callbacks are typically used to allow a server to notify a client to update itself. For example, in a banking application, clients might maintain a local cache to hold the balance of accounts for which they hold references. Each client that uses the server's account object maintains a local copy of its balance. If the client accesses the balance attribute, the local value is returned if the cache is valid. If the cache is invalid, the remote balance is accessed and returned to the client, and the local cache is updated.

When a client makes a deposit to, or withdrawal from, an account, it invalidates the cached balance in the remaining clients that hold a reference to that account. These clients must be informed that their cached value is invalid. To do this, the real account object in the server must notify (that is, call back) its clients whenever its balance changes.

Implementing Callbacks

Overview

This section describes how to implement callbacks using Artix Connect. Artix Connect supports callbacks on any of the middleware platforms supported by Artix.

In this section

This section discusses the following topics:

Callback Demonstration

Overview

The callback example described in this section is based on the CORBA Callback demonstration, which is located in:

```
ArtixConnectInstallDir/artix/Version/demos/dotnet/corba_callback
```

For details on how to run this demo, see the README.txt file in the demo directory.

Graphical view

Example 23 illustrates how the callback proceeds:

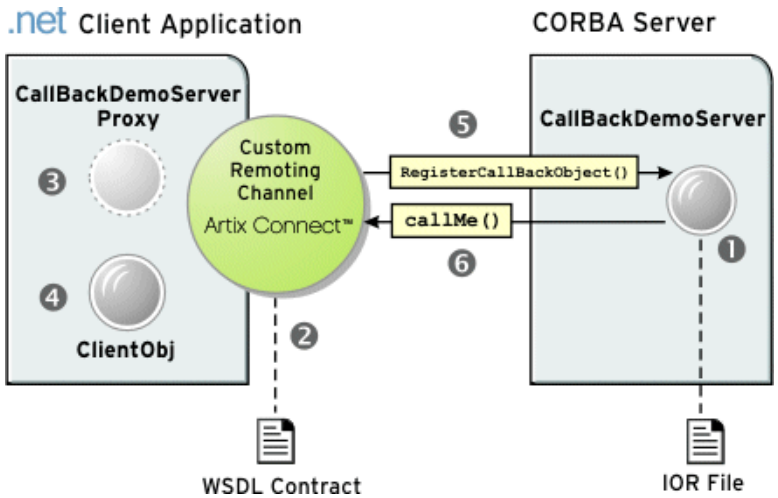

Figure 23: *Callback in Progress*

Example 23 can be explained as follows:

1. When the CORBA server process starts, it creates a CORBA object, `CallBackDemoServer`, and writes a reference to the object to a file, `callback_corba_service.ior`. It then starts to listen for communications from the client over the Internet Inter-ORB Protocol (IIOP).
2. When the client starts, it reads the WSDL contract. The WSDL contract contains details of the types and protocols that can be used to contact the CORBA server. It also contains details of the location of the `callback_corba_service.ior` file, which the client uses to locate the server.
3. The client creates a proxy of the target CORBA server.
4. The client creates a native .NET object, `clientObj`, of type `ClientObjectImpl`, which in turn inherits and implements the `ClientCallbackObject` interface.
5. The client calls `RegisterCallBackObject()` on the CORBA server and passes it a reference to `clientObj`. This notifies the server of the callback service.
6. When the server receives the callback reference, it calls back to the client by invoking on the client's `callMe()` operation.

Callback WSDL Contract

Overview

The first step in implementing client callback functionality is to define the client and server in a WSDL contract. The WSDL contract is the only thing required by the .NET client to invoke on the CORBA server.

In this subsection

This subsection describes the WSDL contract that defines the interaction between the client and the server in the `CORBA Callback` demonstration. It was automatically generated from the CORBA server's IDL file using the Artix Designer, which is available in Artix 3.0.

> **Note:** This guide assumes that the WSDL contract already exists.
>
> For more information on using Artix to develop WSDL contracts, see the Designing Artix Solutions guide. For more information on using Artix to expose CORBA servers as Web services, including generating WSDL from IDL, see the Artix for CORBA guide.

WSDL contract

Example 2 shows the WSDL contract, `callback.wsdl`, used in the `CORBA Callback` demonstration. It is located in:

```
ArtixInstallDir/artix/Version/demos/dotnet/corba_callback/etc
```

Example 2: *Example Callback WSDL Contract*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions
 targetNamespace="http://schemas.iona.com/idl/callback.idl"
 xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:tns="http://schemas.iona.com/idl/callback.idl"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:xsd1="http://schemas.iona.com/idltypes/callback.idl"
 xmlns:corba="http://schemas.iona.com/bindings/corba"
 xmlns:corbatm="http://schemas.iona.com/typemap/corba/
 callback.idl"
 xmlns:references="http://schemas.iona.com/references">
  <types>
    <schema targetNamespace=
     "http://schemas.iona.com/idltypes/callback.idl"
     xmlns="http://www.w3.org/2001/XMLSchema"
```

Example 2: *Example Callback WSDL Contract*

```
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
     <xsd:import schemaLocation=
     "http://schemas.iona.com/references/references.xsd"
     namespace="http://schemas.iona.com/references"/>
     <xsd:element name="ClientCallbackObject.callMe">
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="s" type="xsd:string"/>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
     <xsd:element
      name="CallBackDemoServer.RegisterCallBackObject">
       <xsd:complexType>
         <xsd:sequence>
           <xsd:element name="obj" type="references:Reference"/>
         </xsd:sequence>
       </xsd:complexType>
     </xsd:element>
   </schema>
</types>
<message name="ClientCallbackObject.callMe">
   <part name="parameters"
    element="xsd1:ClientCallbackObject.callMe"/>
</message>
<message name="CallBackDemoServer.RegisterCallBackObject">
   <part name="parameters"
    element="xsd1:CallBackDemoServer.RegisterCallBackObject"/>
</message>

1  <portType name="ClientCallbackObject">
     <operation name="callMe">
       <input message="tns:ClientCallbackObject.callMe"
        name="callMe"/>
     </operation>
   </portType>
2  <portType name="CallBackDemoServer">
     <operation name="RegisterCallBackObject">
       <input message=
       "tns:CallBackDemoServer.RegisterCallBackObject"
        name="RegisterCallBackObject"/>
     </operation>
   </portType>
```

Example 2: *Example Callback WSDL Contract*

```
  <binding name="ClientCallbackObjectCORBABinding"
   type="tns:ClientCallbackObject">
    <corba:binding repositoryID="IDL:ClientCallbackObject:1.0"/>
    <operation name="callMe">
      <corba:operation name="callMe">
        <corba:param name="s" mode="in" idltype="corba:string"/>
      </corba:operation>
      <input/>
    </operation>
  </binding>
  <binding name="CallBackDemoServerCORBABinding"
   type="tns:CallBackDemoServer">
    <corba:binding repositoryID="IDL:CallBackDemoServer:1.0"/>
    <operation name="RegisterCallBackObject">
      <corba:operation name="RegisterCallBackObject">
        <corba:param name="obj" mode="in"
         idltype="corbatm:ClientCallbackObject"/>
      </corba:operation>
      <input/>
    </operation>
  </binding>

3 <service name="ClientCallbackObjectCORBAService">
    <port name="ClientCallbackObjectCORBAPort"
     binding="tns:ClientCallbackObjectCORBABinding">
      <corba:address location="ior:"/>
    </port>
  </service>
4 <service name="CallBackDemoServerCORBAService">
    <port name="CallBackDemoServerCORBAPort"
     binding="tns:CallBackDemoServerCORBABinding">
5     <corba:address location=
       "file:..\..\etc\callback_corba_service.ior"/>
    </port>
  </service>

  <corba:typeMapping targetNamespace=
   "http://schemas.iona.com/typemap/corba/callback.idl">
    <corba:object name="ClientCallbackObject"
     type="references:Reference"
     repositoryID="IDL:ClientCallbackObject:1.0"
     binding="tns:ClientCallbackObjectCORBABinding"/>
  </corba:typeMapping>
</definitions>
```

The WSDL definitions shown in the preceding example, `callback.wsdl`, can be explained as follows:

1. The `ClientCallbackObject` port type is implemented on the client side. It contains a `callMe` operation that takes a single string argument. The server calls back on this operation after it receives a reference to the client's service.
2. The `CallBackDemoServer` port type is implemented on the server side and supports a single WSDL operation—`RegisterCallBackObject`. The `RegisterCallBackObject` operation takes a single Artix reference argument, which is used to pass a reference to the client callback object.
3. Specifies that the client callback object receives messages via IIOP. The client callback address, `ior:`, acts as a placeholder for the address generated dynamically at runtime.
4. Specifies that clients should communicate with the server using IIOP.
5. When the CORBA server process starts, it creates a CORBA object and writes a reference to the object to a file. The server's address is contained in that file—**`file:..\..\etc\callback_corba_service.ior`**.

Implementing the Client in C#

Overview

This subsection describes how to implement a client based on the WSDL contract shown "Callback WSDL Contract" on page 50. The client is an implementation of the `ClientObject` port type. The following topics are covered:

- Main client code.
- Client implementation code

Main client code

Example 3 shows code contained in the `CorbaCallback.cs` file. It contains the C# mainline code that invokes on the server:

Example 3: *CorbaCallback.cs*

```
...
1     ChannelServices.RegisterChannel(new ArtixClientChannel());
...
2     callBackSrvObj = (CallbackDemoNameSpace.CallBackDemoServer)
      Activator.GetObject(typeof(CallbackDemoNameSpace.CallBackDemo
      Server), "artixref:../../etc/callback.wsdl
      http://schemas.iona.com/idl/callback.idl
      CallBackDemoServerCORBAService CallBackDemoServerCORBAPort");

      // Test the callback, allow 30 secs for it to occur.
3     ClientObjectImpl clientObj = new ClientObjectImpl();
      Console.WriteLine("Registering the Callback object");
4     callBackSrvObj.RegisterCallBackObject(clientObj);
      Thread.Sleep(1000);
      int i = 0;
      while ((!clientObj.called) && (i < 30))
      {
          Thread.Sleep(1000);
          i++;
      }
...
```

The code shown in Example 3 can be explained as follows:

1. Registers the Artix remoting channel. This can be specified in an Artix configuration file rather than programmatically.
2. Creates a proxy of the target object in the client's address space. Specifies an Artix reference, which is made up of four parts:
   i. The location of the WSDL contract.
   ii. The target namespace. Each Web service requires a unique namespace that makes it possible for client applications to differentiate between Web services that might use the same method name. Although the namespace resembles a typical URL, do not assume that it is viewable in a Web browser—it is merely a unique identifier.
   iii. The name of the service that the clients should use; in this case, `CallBackDemoServerCORBAService`.
   iv. The name of the port that the client should use; in this case `CallBackDemoServerCORBAPort`.
3. Creates an implementation object, `clientObj`, of the `ClientObject` type.
4. Calls the `RegisterCallBackObject()` operation on the `callBackSrvObj` server object, and passes it a reference to its implementation object, `clientObj`. This allows the server to subsequently invoke operations on the client callback object.

Client implementation code

Example 4 shows code contained in the `ClientObjectImpl.cs` file. It implements the .NET object that receives the server callback:

Example 4: *ClientObjectImpl.cs*

```
using System;

1  [System.Web.Services.WebService(Name=
      "ClientCallbackObjectCORBAService",
      Namespace="http://schemas.iona.com/idl/callback.idl")]
2  public class ClientObjectImpl :
      CallbackDemoNameSpace.ClientCallbackObject
   {
3      public System.Boolean called;
       public ClientObjectImpl()
```

Example 4: *ClientObjectImpl.cs*

```
    {
        called = false;
    }
    #region ClientCallbackObject Members

4   public void callMe(string s)
    {
       Console.WriteLine("ClientObjectImpl::callMe(): called.");
       Console.WriteLine("    " + s);
       Console.WriteLine("ClientObjectImpl::callMe():
       returning.");
       called = true;
    }
    #endregion
}
```

1. Specifies Web service meta information for the class:
   i. The `Name` property specifies the name of the service, as defined in the WSDL contract.
   ii. The `Namespace` property specifies a unique namespace for the Web service, as defined in the WSDL contract.

Note: You do not need to include a `Description` property for the Web service attribute if the client and server port types are defined in the same WSDL contract. This is normally the case for callbacks. If, however, the client port type is defined in a different WSDL contract from the server port type, you must add a `Description` property that specifies the client WSDL contract; for example, `Description="../../etc/callback.wsdl"`

2. Specifies the name of the client's callback implementation class. You can use any name for this, but you must specify that it inherits from the `CallbackDemoNameSpace.ClientCallbackObject` base class, which is taken from the `PortType` element in the WSDL contract.
3. It is possible to add operations and properties to the client that are not defined in the WSDL contract. These can only be used by the client. Here, for example, the `called` property lets the client to know when the server has called back.
4. Implements the `callMe()` operation defined in the WSDL contract.

Implementing the Server

Overview

Artix Connect can communicate with any server that supports the transports and protocols supported by Artix, including SOAP over HTTP, CORBA, IIOP, BEA Tuxedo, IBM WebSphere MQ (formerly MQSeries), TIBCO Rendezvous, and the Java Messaging Service. To use Artix Connect, you do not have to make any changes to such servers. All that Artix Connect requires is the WSDL contract that defines the server.

In this subsection

This section describes the CORBA server that is used in the `CORBA Callback` demonstration. The steps used to implement it were:

- Step 1—Implementing the CallBackDemoServer port type
- Step 2—Invoking the callMe() operation on the client

Step 1—Implementing the CallBackDemoServer port type

An implementation class was provided for the `CallBackDemoServer` port type.

The implementation of the `RegisterCallBackObject()` operation receives a CORBA object reference from the client. When the client invokes the `RegisterCallBackObject()` operation on the server, a CORBA proxy object for the client's `ClientObject` object is created in the Artix Connect bridge. Artix Connect transforms the .NET object reference in the client code to a CORBA object reference, which it passes to the CORBA servant.

The server uses the CORBA proxy object to call back to the client. The implementation of the `RegisterCallBackObject()` operation stores the reference to the CORBA proxy for this purpose.

Step 2—Invoking the callMe() operation on the client

After the CORBA proxy object for the client's `ClientObject` object has been created in the Artix Connect bridge, the server can then invoke the `callMe()` operation on this proxy object.

CHAPTER 5

Development Support Tools

The first step in writing a .NET client that can communicate with an Artix Web service is to obtain .NET metadata, which describes the target service interfaces and types as .NET interfaces and types. Artix Connect includes a Web service wizard that generates the .NET metadata and client starting point code for you, from within the Visual Studio .NET 2003 development environment. All it requires is the Web service WSDL contract. In addition, Artix Connect includes a wsdltodotnet command-line utility that you can use, as an alternative to the wizard, to generate .NET metadata from a WSDL contract.

In this chapter

This chapter discusses the following topics:

Artix Connect Wizard

Overview

Artix Connect provides Web service wizard, **Artix Connect Wizard**, which you can use to generate .NET metadata, which describes the target service interfaces and types as .NET interfaces and types. You can use the wizard from within the Microsoft Visual Studio .NET 2003 development environment. It enables you to select the WSDL contract for the service to which you want the client to connect and, as well as producing the .NET metadata from the WSDL contract, the wizard produces client starting point code that you can use to develop your client application. The .NET metadata assembly is stored in a DLL file that is generated, behind the scenes, by the `wsdltodotnet` command-line utility.

In this section

This section describes the Artix Connect Wizard and points you to an example of using the wizard. The following topics are covered:

- Main screen
- Fields
- Example of using the Artix Connect Wizard

Main screen

Figure 24 shows the Artix Connect Wizard main screen:

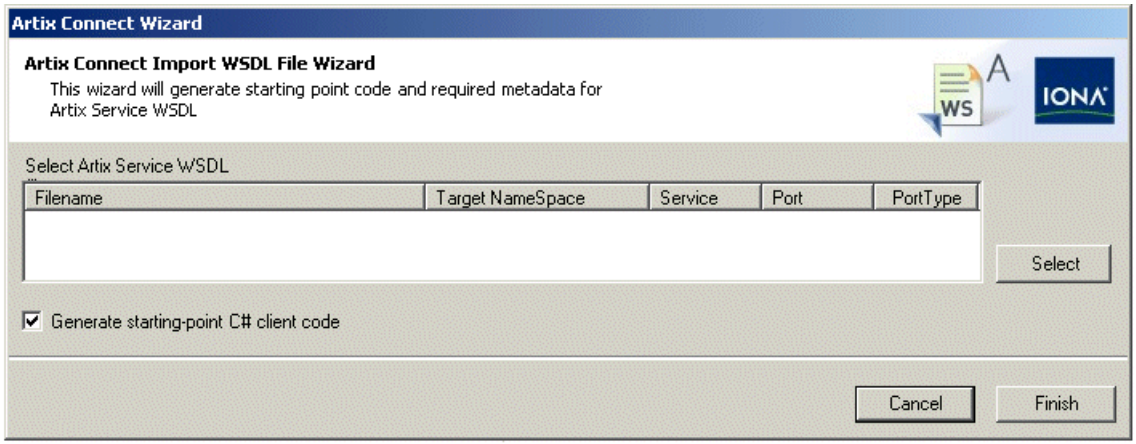

Figure 24: *Artix Connect Wizard*

Fields

The Artix Connect Wizard fields are described below. They are populated automatically when you select the WSDL contract for the service to which you want your client to connect. The values are taken directly from the WSDL contract

`Filename`	The WSDL filename and location.
`Target NameSpace`	Specifies the target namespace.
`Service`	Specifies the name of the service that the client wants to use.
`Port`	Specifies the name of the port that the client wants to use.
`PortType`	Specifies the port type of the server that the client wants to connect to.

Note: If the WSDL contract contains more than one service, the wizard selects the first service. If you want to select a different service, you must change the values in the generated starting point code. You cannot change the values in the wizard.

Example of using the Artix Connect Wizard

For an example of using the Artix Connect Wizard, see "Developing .NET Clients" on page 27.

wsdltodotnet Command-line Utility

Overview

Artix Connect provides an `wsdltodotnet` command-line utility that you can use to map WSDL types to .NET types. The .NET metadata assembly is stored in a DLL file that is generated by the `wsdltodotnet` utility. The `wsdltodotnet` command-line utility is provided as an alternative to using the Artix Connect Wizard and is useful if you want to view the C# files that are used to generate the type DLL file.

Note: If you use the `wsdltodotnet` command-line utility to generate the .NET metadata, you must add the `Artix.Remoting.dll` and the *`PortType_Name`*`.dll` metadata assembly, which contains the type information for the server, to your project. You can do this by right-clicking on your project and selecting the Add References option. Select the `Artix.Remoting.dll` from the list that appears and select the generated *`PortType_Name`*`.dll` by browsing to the location where you have it stored.

Generating metadata

You can generate metadata at the command line using the following command:

```
wsdltodotnet.exe [-source] [-quiet] [-verbose]
[ -namespace <C# NameSpace> ] [ -name <C# Assembly Name> ]
[-v] [-?] [<wsdlurl>]
```

You must specify the location of a valid WSDL contract file, `wsdlurl`, for the `wsdltodotnet` metadata generator to work. You can also supply the following optional parameters:

`-source`	Outputs C# source code as well as an assembly containing .NET metadata. This is not generated by default and is not required to build and run the demos. It is useful if you want to examine the type mapping.
`-quiet`	Specifies quiet mode.
`-verbose`	Specifies verbose mode.
`-namespace <C# NameSpace>`	Specifies the namespace to use for the generated code. If not specified the namespace defaults to `[<FirstPortTypeinWSDLfile>NameSpace]`

`-name <C# Assembly Name>`	Specifies the name of the assembly containing the .NET metadata. If not specified, the names defaults to `[<FirstPortTypeinWSDLfile>]`.
`-v`	Displays the version of the tool.
`-?`	Displays the `wsdltodotnet`'s usage message.

Usage examples

Example 1

The following command generates a .NET metadata assembly within a `Greeter.dll` file, based on the `Greeter` port type described in the `hello_world.wsdl` file in the Artix Connect `Hello World` demo. In this case, the command is being run from the directory in which the WSDL file exists; that is:
`ArtixConnectInstallDir``\artix\`*`Version`*`\demos\dotnet\hello_world\etc`:

```
wsdltodotnet hello_world.wsdl
```

Example 2

The following command generates a .NET metadata assembly called `TestGreeter` and the C# source file, `Greeter.cs`. Again, the command is being run from the directory in which the WSDL file is stored:

```
wsdltodotnet -source -name TestGreeter hello_world.wsdl
```

CHAPTER 6

Deploying an Artix Connect Application

This chapter provides an overview of the deployment model you can adopt when deploying a distributed application with Artix Connect. It also describes the steps you must follow to deploy a distributed Artix Connect application.

In This Chapter

This chapter discusses the following topics:

Deployment Model

Overview

Figure 25 provides a graphical overview of a typical deployment scenario. Although WebSphere MQ Server is chosen as the server in this example, any server that uses the transports and protocols supported by Artix can be used, including SOAP over HTTP, CORBA, IIOP, BEA Tuxedo, TIBCO Rendezvous, and Java Messaging Service.

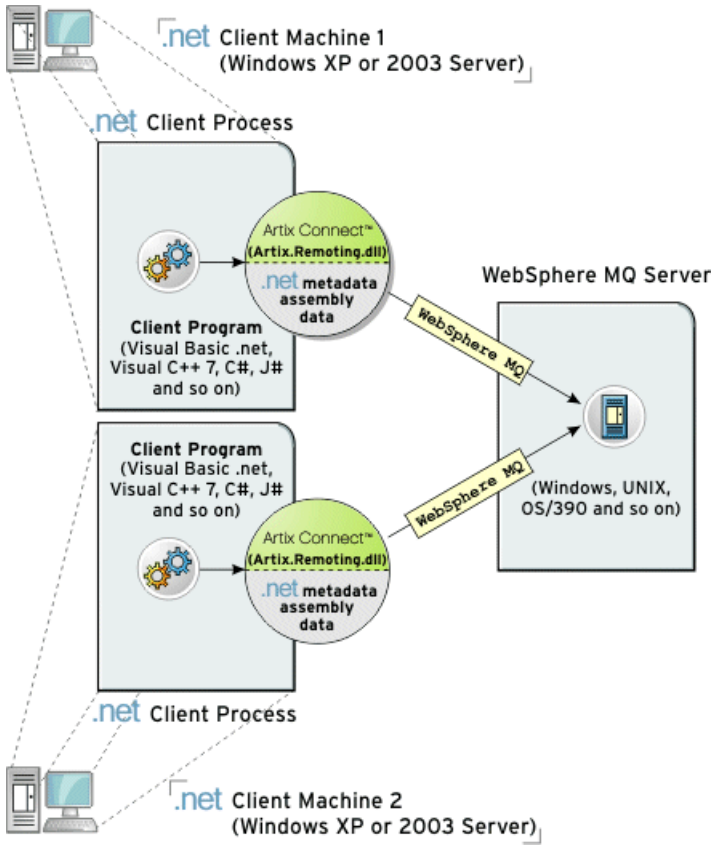

Figure 25: *Typical Deployment Scenario*

Explanation

The deployment scenario overview in Figure 25 can be outlined as follows:

- Each .NET client machine must be running on Windows 2000, NT, XP or 2003 Server.
- The Artix Connect bridge (that is, `Artix.Remoting` custom remoting channel) always runs in-process (that is, within the client process).
- The .NET metadata DLL file is also exposed within the client process.
- Each client machine uses the protocol specified in the WSDL file to communicate with the back-end server—in this case WebSphere MQ.
- The back-end server process can be running on any platform that is supported by Artix.

Deployment Steps

Overview

This section describes the steps involved in deploying an Artix Connect application.

Required components

Four components are required for successful deployment of an Artix Connect client:

- The .NET client executable.
- The .NET metadata assembly DLL.
- Artix Connect runtime installation.
- WSDL contract.

These must be copied from the development host to every deployment host.

Steps

The steps to deploy an Artix Connect client application are:

1. Install the Artix Connect runtime on the deployment host. The `Artix.Remoting` assembly must be in the client directory or in the GAC of the client machine. The Artix Connect installer places the `Artix.Remoting` assembly in the GAC by default.
2. Configure Artix Connect. The installer allows you to set the environment variables that Artix Connect requires during installation. If you choose not to set them during installation, you can either run the `artix_env.bat` script or set them manually later. See "Configuration" on page 111 for more details.
3. Copy the client executable and the .NET metadata DLL to the deployment host.
4. Copy the WSDL contract for the service to which you want to connect.

Repeat these steps as necessary for each deployment host on your system.

CHAPTER 7

Introduction to WSDL

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

In this chapter

This chapter discusses the following topics:

Note: This chapter is taken from the Getting Started with Artix guide. For more information, please refer to that guide.

WSDL Basics

Overview

Web Services Description Language (WSDL) is an XML document format used to describe services offered over the Web. WSDL is standardized by the World Wide Web Consortium (W3C) and is currently at revision 1.1. You can find the standard on the W3C website, www.w3.org.

Abstract operations

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

Port types

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

Concrete details

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

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

Namespaces and imported descriptions

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

Elements of a WSDL document

A WSDL document is made up of the following elements:

- `import`—allows you to import another WSDL or XSD file
- `types`—the definition of complex data types based on in-line type descriptions and/or external definitions such as those in an XML Schema (XSD).
- `message`—the abstract definition of the data being communicated.
- `operation`—the abstract description of an action.
- `portType`—the set of operations representing an 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

Example 5 shows a simple WSDL document. It defines a SOAP over HTTP service access point that returns the date.

Example 5: *Simple WSDL*

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

Example 5: *Simple WSDL (Continued)*

```
  <message name="DateResponse">
    <part name="date" element="xsd1:dateType"/>
  </message>
  <portType name="DatePortType">
    <operation name="sendDate">
      <output message="tns:DateResponse" name="sendDate"/>
    </operation>
  </portType>
  <binding name="DatePortBinding" type="tns:DatePortType">
    <soap:binding style="rpc"
   transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="sendDate">
      <soap:operation soapAction="" style="rpc"/>
      <output name="sendDate">
        <soap:body
   encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
   namespace="urn:dateservice" use="encoded"/>
      </output>
    </operation>
  </binding>
  <service name="DateService">
    <port binding="tns:DatePortBinding" name="DatePort">
      <soap:address location="http://www.iona.com/DatePort/"/>
    </port>
  </service>
</definitions>
```

Abstract Data Type Definitions

Overview

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

Complex type definitions

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

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

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

Example

The structure, `personalInfo`, defined in Example 6, contains a `string`, an `int`, and an `enum`. The `string` and the `int` both have equivalent XSD types and do not require special type mapping. The enumerated type `hairColorType`, however, does need to be described in XSD.

Example 6: *personalInfo*

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

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

Example 7 shows one mapping of `personalInfo` into XSD. This mapping is a direct representation of the data types defined in Example 6. `hairColorType` is described using a named `simpleType` because it does not have any child elements. `personalInfo` is defined as an `element` so that it can be used in messages later in the contract.

Example 7: *XSD type definition for personalInfo*

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

Another way to map personalInfo is to describe `hairColorType` in-line as shown in Example 8. WIth this mapping, however, you cannot reuse the description of `hairColorType`.

Example 8: *Alternate XSD mapping for personalInfo*

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

Example 8: *Alternate XSD mapping for personalInfo (Continued)*

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

Abstract Message Definitions

Overview

WSDL is designed to describe how data is passed over a network. It describes data that is exchanged between two endpoints in terms of abstract messages described in `message` elements. Each abstract message consists of one or more parts, defined in `part` elements. These abstract messages represent the parameters passed by the operations defined by the WSDL document and are mapped to concrete data formats in the WSDL document's `binding` elements.

Messages and parameter lists

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

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

Messages allow for concrete methods defined in programming languages like C++ to be mapped to abstract WSDL operations. Each message contains a number of `part` elements that represent one element in a parameter list. Therefore, all of the input parameters for a method call are defined in one message and all of the output parameters, including the operation's return value, would be mapped to another message.

Example

For example, imagine a server that stored personal information as defined in Example 6 on page 73 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 9.

Example 9: *personalInfo lookup method*

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

This method signature could be mapped to the WSDL fragment shown in Example 10.

Example 10: *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 you name messages in a way that shows whether they are input messages (requests) or output messages (responses).

Message parts

Message parts are the formal data elements of the abstract message. Each part is identified by a name and an attribute specifying its data type. The data type attributes are listed in Table 1

Table 1: *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`, which 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 11.

Example 11: *Reused part*

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

Example 11: *Reused part (Continued)*

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

Abstract Interface Definitions

Overview

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

Port types

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

Port types are described using the `portType` element in a WSDL document. Each port type in a WSDL document must have a unique name, specified using the `name` attribute, and is made up of a collection of operations, described in `operation` elements. A WSDL document can describe any number of port types.

Operations

Operations, described in `operation` elements in a WSDL document are an abstract description of an interaction between two endpoints. For example, a request for a checking account balance and an order for 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.

Elements of an operation

Each operation is made up of a set of elements. The elements represent the messages communicated between the endpoints to execute the operation. The elements that can describe an operation are listed in Table 2.

Table 2: *Operation Message Elements*

Element	Description
`input`	Specifies a message that is received from another endpoint. This element can occur at most once for each operation.

Table 2: *Operation Message Elements*

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

An operation is required to have at least one `input` or `output` element. The elements are defined by two attributes listed inTable 3.

Table 3: *Attributes of the Input and Output Elements*

Attribute	Description
`name`	Identifies the message so it can be referenced when mapping the operation to a concrete data format. The name must be unique within the enclosing port type.
`message`	Specifies the abstract message that describes the data being sent or received. The value of the `message` attribute must correspond to the `name` attribute of one of the abstract messages defined in the WSDL document.

It is not necessary to specify the `name` attribute for all input and output elements; WSDL provides a default naming scheme based on the enclosing operation's name. If only one element is used in the operation, the element name defaults to the name of the operation. If both an `input` and an `output` element are used, the element name defaults to the name of the operation with `Request` or `Response` respectively appended to the name.

Return values

Because the port type is an abstract definition of the data passed during an operation, WSDL does not provide for return values to be specified for an operation. If a method returns a value it will be mapped into the `output` message as the last `part` of that message. The concrete details of how the message parts are mapped into a physical representation are described in the binding section.

Example

For example, in implementing a server that stored personal information in the structure defined in Example 6 on page 73, you might use an interface similar to the one shown in Example 12.

Example 12: *personalInfo lookup interface*

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

This interface could be mapped to the port type in Example 13.

Example 13: *personalInfo lookup port type*

```
<types>
...
  <element name="idNotFound" type="idNotFoundType">
  <complexType name="idNotFoundType">
    <sequence>
      <element name="ErrorMsg" type="xsd:string"/>
      <element name="ErrorID" type="xsd:int"/>
    </sequence>
  </complexType>
</types>
<message name="personalLookupRequest">
  <part name="empId" type="xsd:int" />
</message>
<message name="personalLookupResponse">
  <part name="return" element="xsd1:personalInfo" />
</message>
<message name="idNotFoundException">
  <part name="exception" element="xsd1:idNotFound" />
</message>
<portType name="personalInfoLookup">
  <operation name="lookup">
    <input name="empID" message="personalLookupRequest" />
    <output name="return" message="personalLookupResponse" />
    <fault name="exception" message="idNotFoundException" />
  </operation>
</portType>
```

Mapping to the Concrete Details

Overview

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

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

Bindings

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

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

Services

The final piece of information needed to describe how to connect a remote service is the network information needed to locate it. This information is defined inside a `port` element. Each port specifies the address and configuration information for connecting the application to a network.

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

CHAPTER 8

WSDL to .NET Mapping

To enable interworking between .NET clients and services described in WSDL contracts, .NET clients must be presented with metadata that describes the interfaces exposed by the WSDL contract. When using .NET Remoting, the .NET types must use the .NET Common Type System (CTS). This chapter describes how Artix Connect maps WSDL types to .NET CTS types.

In this chapter

This chapter discusses the following topics:

Mapping a WSDL Contract to CTS

Overview

Artix Connect maps WSDL contracts into C# using the mapping described in this section.

In this section

This section contains the following subsections:

Port Types

Overview

A C# interface is generated for each `portType` element in an Artix WSDL contract. The name of the generated interface is taken from the name attribute of the `portType` element.

WSDL contract example

For example, the WSDL contract shown in Example 14 generates a C# interface called `sportsCenterPortType`. which contains one operation, called `update`. (see Example 15)

Example 14: *Segment of Sports Center WSDL Contract*

```
<message name="scoreRequest">
  <part name="teamName" type="xsd:string" />
</message>
<message name="scoreReply">
  <part name="score" type="xsd:int" />
</message>
<portType name="sportsCenterPortType">
  <operation name="update">
    <input message="scoreRequest" name="request" />
    <ouput message="scoreReply" name="reply" />
  </operation>
</portType>
<binding name="scoreBinding" type="tns:sportsCenterPortType">
...
<service name="sportsService">
  <port name="sportsCenterPort" binding="tns:scoreBinding">
...
```

CTS mapping

Example 15 shows how the preceding WSDL contract maps to a C# interface defined using the Common Type System:

Example 15: *C# Mapping for Sports Center WSDL Contract*

```
// C#
public interface sportsCenterPortType
{
    System.Int32 update(System.String teamName);
}
```

Operations

Overview

Every `operation` element contained in a WSDL contract generates a C# method within the interface defined for the `operation` element's `portType`. The generated method's name is taken from the `operation` element's name attribute.

WSDL contract example

Example 16 shows a WSDL contract that contains an operation called `greetMe`:

Example 16: *WSDL Contract containing greetMe Operation*

```
<wsdl:portType name="Greeter">
  <wsdl:operation name="sayHi">
    <wsdl:input message="tns:sayHiRequest" name="sayHiRequest"/>
    <wsdl:output message="tns:sayHiResponse"
    name="sayHiResponse"/>
  </wsdl:operation>
  <wsdl:operation name="greetMe">
    <wsdl:input message="tns:greetMeRequest"
     name="greetMeRequest"/>
    <wsdl:output message="tns:greetMeResponse"
     name="greetMeResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

CTS mapping

The WSDL contract shown in Example 16 maps to a C# interface defined using the Common Type System as follows:

```
public interface Greeter {
System.String sayHi();
System.String greetMe(System.String me);
}
```

Messages

Overview

The message parts of an operation's input and output elements are mapped as parameters in the generated method's signature. The parameter names are taken from the name attribute of the `part` element.

The order of the mapped parameters is based on the order in which they appear in the WSDL contract.

Input message parts are listed before output message parts. Message parts that are listed in both the input and output messages are considered `inout` parameters and are listed according to their position in the input message.

The first part in output messages are mapped to a return types. For the remaining message parts, each part is mapped to either `ref` parameter or an `out` parameter. If the message part is listed in both the input and output message, it is mapped to a `ref` parameter. If the message part is only listed in the output message, it is mapped to an `out` parameter.

WSDL contract example

For example, the WSDL contract fragment shown in Example 17 maps to a `SimpleTestPortType` interface that contains a `test_short` operation, which has a return type of `String` and a parameter list that contains two input parameters and two output parameters.

Example 17: *Segment of WSDL Contract*

```
<message name="test_short">
  <part name="x" element="s:short_x"/>
  <part name="y" element="s:short_y"/>
</message>
<message name="test_short_response">
  <part name="return" element="s:short_return"/>
  <part name="y" element="s:short_y"/>
  <part name="z" element="s:short_z"/>
</message>
 <portType name="SimpleTestPortType">
  <operation name="test_short">
   <input name="test_short" message="tns:test_short"/>
   <output name="test_short_response"
   message="tns:test_short_response"/>
 </operation>
</portType>
```

CTS mapping

Example 18 shows how the preceding WSDL contract maps to a C# interface defined using the Common Type System:

Example 18: *C# Mapping of SimpleTestPortType*

```
// C#
public interface SimpleTestPortType
{
System.Int16 test_short(System.Int16 x, ref System.Int16 y, out
   System.Int16 z);
}
```

Document/Literal Wrapped Style

Overview

This subsection describes the document/literal wrapped style for defining WSDL operations and parameters. The document/literal wrapped style is distinguished by the fact that it uses single-part messages. The single part is defined as a schema element that contains a sequence of elements, one for each parameter.

Request message

The request message in document/literal wrapped style must obey the following conventions:

- The single element that wraps the input parameters must have the same name as the WSDL operation, `OperationName`.
- The single part must have the name, `parameters`.

Reply message

The reply message in document/literal wrapped style must obey the following conventions:

- The single element that wraps the output parameters must have the form, `OperationNameResult`.
- The single part must have the name, `parameters`.

You can declare a WSDL operation in document/literal wrapped style as follows:

- In the `schema` section of the WSDL contract, define an `element` (the input part wrapping element) as a sequence type containing elements for each of the in and inout parameters.
- In the `schema` section of the WSDL contract, define another `element` (the output part wrapping element) as a sequence type containing elements for each of the inout and out parameters.
- Declare a single-part input message, including all of the in and inout parameters for the new operation.
- Declare a single-part output message, including all of the out and inout parameters for the operation.
- Within the scope of `portType`, declare a single operation that includes a single `input` message and a single `output` message.

Artix Connect automatically detects that document/literal wrapped style is being used, as long as the WSDL contract obeys the conventions outlined above. If document/literal wrapped style is detected, Artix Connect unwraps the operation parameters to generate a normal function signature in C#.

WSDL contract example

Example 19 shows how the WSDL contract shown in Example 17 could be expressed in WSDL using the document/literal style:

Example 19: *Segment of Sports Final WSDL Contract using Document/Literal Style*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ... >
    <wsdl:types>
        <schema targetNamespace="..."
                xmlns="http://www.w3.org/2001/XMLSchema">
            <element name="final">
                <complexType>
                    <sequence>
                       <element name="team1" type="xsd:string"/>
                       <element name="team2" type="xsd:string"/>
                    </sequence>
                </complexType>
            </element>
            <element name="finalResult">
                <complexType>
                    <sequence>
                        <element name="winTeam"
                         type="xsd:string"/>
                        <element name="team1score"
                         type="xsd:int"/>
                        <element name="team2score"
                         type="xsd:int"/>
                    </sequence>
                </complexType>
            </element>
        </schema>
    </wsdl:types>
    <message name="final">
        <part name="parameters" element="tns:final"/>
    </message>
    <message name="finalResult">
        <part name="parameters" element="tns:finalResult"/>
    </message>
```

Example 19: *Segment of Sports Final WSDL Contract using Document/Literal Style*

```
    <wsdl:portType name="sportsFinalPortType">
        <wsdl:operation name="final">
            <wsdl:input  message="tns:final"
                         name="final"/>
            <wsdl:output message="tns:finalResult"
                         name="finalResult"/>
        </wsdl:operation>
    </wsdl:portType>
    ...
<binding name="scoreBinding" type="tns:sportsFinalPortType">
...
<service name="sportsService">
  <port name="sportsFinalPort" binding="tns:scoreBinding">
...
</definitions>
```

CTS mapping

Example 20 shows how the preceding WSDL contract maps, for example, to a C# interface defined using the Common Type System:

Example 20: *C# Mapping for Sports Final WSDL Contract that uses Document/Literal style*

```
// C#
public interface sportsFinal
{
    System.String final(System.String team1, System.String team2,
                        out System.Int32 team1score,
                        out System.Int32 team2score);
}
```

Simple Types

Overview

This section describes the mapping of simple WSDL types to CTS.

In this section

This section includes the following subsections:

Atomic Types

Table of atomic types

Table 4 shows how the XSD schema atomic types map to .NET CTS types:

Table 4: *XSD Schema Simple Types Mapping to .NET CTS Types*

XSD Schema Type	CTS Type
xsd:anySimpleType	System.String
xsd:anyURI	System.String
xsd:base64Binary	System.Byte[]
xsd:boolean	System.Boolean
xsd:byte	System.SByte
xsd:unsignedByte	System.Byte
xsd:dateTime	System.DateTime
xsd:double	System.Double
xsd:decimal	System.Decimal
xsd:float	System.Single
xsd:gDay	System.String
xsd:gMonth	System.String
xsd:gMonthDay	System.String
xsd:gYear	System.String
xsd:gYearMonth	System.String
xsd:hexBinary	System.Byte[]
xsd:ID	System.String
xsd:int	System.Int32
xsd:unsignedInt	System.UInt32
xsd:integer	System.String

Table 4: *XSD Schema Simple Types Mapping to .NET CTS Types*

XSD Schema Type	CTS Type
xsd:long	System.Int64
xsd:unsignedLong	System.UInt64
xsd:negativeInteger	System.String
xsd:nonPositiveInteger	System.String
xsd:nonNegativeInteger	System.String
xsd:positiveInteger	System.String
xsd:QName	System.Xml.XmlQualifiedName
xsd:short	System.Int16
xsd:unsignedShort	System.UInt16
xsd:string	System.String
xsd:time	System.DateTime

Lists

Overview

XML schema supports a mechanism for defining data types that are a list of space separated simple types. Artix Connect maps these lists onto .NET arrays.

WSDL contract example

Example 21 shows a WSDL definition for a list of strings:

Example 21: *WSDL for List of Strings*

```
<types>
...
   <simpleType name="StringList">
      <list itemType="xsd:string"/>
   </simpleType>
   <element name="StringList_x" type="tns:StringList"/>
   <element name="StringList_y" type="tns:StringList"/>
   <element name="StringList_z" type="tns:StringList"/>
   <element name="StringList_return" type="tns:StringList"/>
...
</types>
 <message name="test_StringList">
   <part element="tns:StringList_x" name="x"/>
   <part element="tns:StringList_y" name="y"/>
 </message>
 <message name="test_StringList_response">
   <part element="tns:StringList_return" name="return"/>
   <part element="tns:StringList_y" name="y"/>
   <part element="tns:StringList_z" name="z"/>
 </message>
 <portType name="TypeTestPortType">
    <operation name="test_StringList">
        <input message="tns:test_StringList"
   name="test_StringList"/>
        <output message="tns:test_StringList_response"
   name="test_StringList_response"/>
      </operation>
</portType>
```

CTS mapping

The WSDL contract shown in Example 21 maps to a .NET array as shown in Example 22:

Example 22: *C# Mapping for StringList*

```
//C#:
System.String[] test_StringList(System.String[] x, ref
                System.String[] y, out System.String[] z);
```

Unsupported Simple Types

Overview

The following simple types are not supported:

- xsd:duration
- xsd:NOTATION
- xsd:IDREF
- xsd:IDREFS
- xsd:ENTITY
- xsd:ENTITIES
- xsd:anySimpleType
- xsd:simpleType/xs:union

Complex Types

Overview

This section describes the mapping of complex WSDL types to .NET CTS types.

In this section

This section contains the following subsections:

Sequence and All Complex Types

Overview

Complex types often describe basic structures that contain a number of fields or elements. XML schema provides two mechanisms for describing a structure. One method is to describe the structure inside of a sequence element. The other is to describe the structure inside of an `all` element. Both methods of describing a structure result in the same generated C# classes.

Difference between sequence and all

The difference between using a sequence and an `all` is in how the elements of the structure are passed on the wire. When a structure is described using a sequence, the elements are passed on the wire in the exact order that they are specified in the WSDL contract. When the structure is described using an `all` element, the elements of the structure can be passed on the wire in any order.

Mapping

Artix Connect maps WSDL sequence and `all` complex types to CTS classes with properties that represent each element.

WSDL contract example

Example 23 shows an XSD sequence type with three simple elements:

Example 23: *WSDL Definition for a Sequence Complex Type*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="SequenceType">
        <sequence>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
            <element name="varString" type="xsd:string"/>
        </sequence>
    </complexType>
    ...
</schema>
```

CTS mapping

Example 24 shows the result of mapping the SequenceType type (from the preceding Example 23) to C# defined using CTS:

Example 24: *C# Mapping for SequenceType*

```
// C#
[System.Serializable()]
public class SequenceType {

    private System.Single _varFloat;
    private System.Int32 _varInt;
    private System.String _varString;

    public virtual System.Single varFloat {
        get {
            return this._varFloat;
        }
        set {
            this._varFloat = value;
        }
    }

    public virtual System.Int32 varInt {
        get {
            return this._varInt;
        }
        set {
            this._varInt = value;
        }
    }

    public virtual System.String varString {
        get {
            return this._varString;
        }
        set {
            this._varString = value;
        }
    }
```

Arrays

Overview

If a sequence only includes one element and this element has `minOccurs` and `maxOccurs` attributes, then Artix Connect generates a class for this sequence, which includes the array properties. Unlike the other mappings listed in this chapter, this differs from the .NET `WSDL.exe` data mapping tool. The `WSDL.exe` tool will not generate a class for this sequence—it directly maps it to an array parameter in the method.

See also SOAP Arrays and Occurance Constraints.

WSDL contract example

Example 25 shows an example of such a sequence:

Example 25: *WSDL Definition for Sequence with one Element containing minOccurs and maxOccurs Attributes*

```
<complexType name="UnboundedArray">
   <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="item"
    type="xsd:string"/>
     </sequence>
</complexType>
<element name="UnboundedArray_x" type="s:UnboundedArray"/>
<element name="UnboundedArray_y" type="s:UnboundedArray"/>
<element name="UnboundedArray_z" type="s:UnboundedArray"/>
<element name="UnboundedArray_return" type="s:UnboundedArray"/>
...
<message name="test_UnboundedArray">
   <part element="s:UnboundedArray_x" name="x"/>
   <part element="s:UnboundedArray_y" name="y"/>
</message>
<message name="test_UnboundedArray_response">
   <part element="s:UnboundedArray_return" name="return"/>
   <part element="s:UnboundedArray_y" name="y"/>
   <part element="s:UnboundedArray_z" name="z"/>
</message>
<portType name="TypeTestPortType">
   <operation name="test_UnboundedArray">
      <input message="tns:test_UnboundedArray"
       name="test_UnboundedArray"/>
       <output message="tns:test_UnboundedArray_response"
        name="test_UnboundedArray_response"/>
    </operation>
```

Example 25: *WSDL Definition for Sequence with one Element containing minOccurs and maxOccurs Attributes*

```
</portType>
```

CTS mapping

Artix Connect maps the WSDL contract shown in Example 25 to C# as shown in Example 26:

Example 26: *Artix Connect C# Mapping for Sequence with one Element containing minOccurs and maxOccurs Attributes*

```
//C#
UnboundedArray test_UnboundedArray(UnboundedArray x, ref
UnboundedArray y, out UnboundedArray z);

public class UnboundedArray {
    private System.String[] _item;
    public virtual System.String[] item {
        get {
            return this._item;
        }
        set {
            this._item = value;
        }
   }
}
```

The .NET `WSDL.exe` tool maps the WSDL contract shown in Example 25 to C# as shown below:

```
public string[] test_UnboundedArray(string[] UnboundedArray_x,
   ref string[] UnboundedArray_y, out string[] UnboundedArray_z)
```

Choice Complex Type

Overview

The .NET CTS has no concept of a `choice` or `union` type. As a result, Artix Connect maps XML schema choice complex types to a generated C# class. Accessor and modifier functions are defined for each element in the choice complex type. The choice complex type is equivalent to a C++ union. Therefore, only one of the elements is accessible at a time.

WSDL contract example

Example 27 shows an XSD choice type with three elements:

Example 27: *WSDL Definition for a Choice Complex Type*

```
<schema targetNamespace="http://soapinterop.org/xsd"
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
    <complexType name="ChoiceType">
        <choice>
            <element name="varFloat" type="xsd:float"/>
            <element name="varInt" type="xsd:int"/>
            <element name="varString" type="xsd:string"/>
        </choice>
    </complexType>
    ...
</schema>
```

CTS mapping

Example 28 shows the result of mapping the `ChoiceType` (from the preceding Example 27) to C#:

Example 28: *C# Mapping of ChoiceType*

```
// C#
public class ChoiceType
{
   [System.Xml.Serialization.XmlElement("varFloat",
   Type=typeof(System.Single), DataType="float")]
   [System.Xml.Serialization.XmlElement("varInt",
   Type=typeof(System.Int32), DataType="int")]
   [System.Xml.Serialization.XmlElement("varString",
   Type=typeof(System.String), DataType="string")]
   private object _Item;
```

Example 28: *C# Mapping of ChoiceType*

```
    public virtual object Item {
         get {
             return this._Item;
         }
         set {
             this._Item = value;
         }
    }
}
```

Attributes

Overview

An attribute is mapped to a field by Artix Connect.

WSDL contract example

Example 29 shows a segment of a WSDL contract that includes an attribute, called `"varAttrString"`:

Example 29: *WSDL Definition including an Attribute*

```
<complexType name="SimpleStruct">
   <sequence>
      <element name="varFloat" type="xsd:float"/>
      <element name="varInt" type="xsd:int"/>
      <element name="varString" type="xsd:string"/>
   </sequence>
   <attribute name="varAttrString" type="xsd:string"/>
</complexType>
```

CTS mapping

The WSDL segment shown in Example 29 maps to C# as shown in Example 30:

Example 30: *C# Mapping for Attribute varAttrString*

```
public class SimpleStruct {
private System.Single _varFloat;
private System.Int32 _varInt;
private System.String _varString;
public System.String varAttrString;
public virtual System.Single varFloat {
   get {
         return this._varFloat;
         }

   set {
         this._varFloat = value;
        }
}
public virtual System.Int32 varInt {
    get {
     return this._varInt;
    }
```

Example 30: *C# Mapping for Attribute varAttrString*

```
    set {
     this._varInt = value;
     }
}

public virtual System.String varString {
    get {
        return this._varString;
    }

    set {
        this._varString = value;
    }
  }
}
```

Enumerations

Overview

Artix Connect maps enumerations defined in WSDL onto .NET enumerations.

WSDL contract example

Example 31 shows a WSDL definition for an enumeration, `DecimalEnum`:

Example 31: *WSDL Definition of Enumeration*

```
<simpleType name="DecimalEnum">
   <restriction base="xsd:decimal">
      <enumeration value="-10.34"/>
      <enumeration value="11.22"/>
      <enumeration value="14.55"/>
   </restriction>
</simpleType>
```

CTS mapping

This maps to a .NET enumeration as shown in Example 32

Example 32: *C# Mapping of DecimalEnum*

```
// C#
[System.Serializable()]
public enum DecimalEnum {

    [System.Xml.Serialization.XmlEnum(Name="-10.34")]
    Item1034,

    [System.Xml.Serialization.XmlEnum(Name="11.22")]
    Item1122,

    [System.Xml.Serialization.XmlEnum(Name="14.55")]
    Item1455,
}
```

Occurance Constraints

Overview

Certain XML schema tags—for example, `element`, `sequence`, `choice`, and `any`—can be declared to occur multiple times using occurrence constraints. The occurrence constraints are specified by assigning integer values (or the special value `unbounded`) to the `minOccurs` and `maxOccurs` attributes.

Currently, `minOccurs` and `maxOccurs` are only supported in sequence elements. If an element in a sequence has `minOccurs` and `maxOccurs` attributes, Artix Connect generates an array for that element.

WSDL contract example

Example 33 shows a WSDL sequence element with `minOccurs` and `maxOccurs` constraints:

Example 33: *WSDL Sequence with Occurrence Constraints*

```
<complexType name="FixedArray">
  <sequence>
  element maxOccurs="3" minOccurs="3" name="item"
  type="xsd:int"/>
  </sequence>
</complexType>
```

CTS mapping

Example 33 maps to C# as follows:}

Example 34: *C# Mapping of WSDL Sequence with Occurrence Constraints*

```
//C#
public class FixedArray {
   private System.Int32 _item;
   public virtual System.Int32 item {

     get {
         return this._item;
     }

     set {
         this._item = value;
     }
 }
```

SOAP Arrays

Overview

SOAP arrays have a relatively rich feature set, including support for sparse arrays and partially transmitted arrays. SOAP arrays map to .NET arrays.

WSDL contract example

Example 35 shows a WSDL definition of a SOAP array:

Example 35: *SOAP Array defined in WSDL*

```
<complexType name="ArrayOfInt">
  <complexContent>
   <restriction base="soap-enc:Array">
    <attribute ref="soap-enc:arrayType" wsdl:arrayType="int[]"/>
   </restriction>
  </complexContent>
</complexType>
...
<message name="echoIntArrayFaultRequest">
  <part name="param" type="ns2:ArrayOfInt"/>
</message>
...
<portType name="SimpleRpcEncPortType">
  <operation name="echoIntArrayFault" parameterOrder="param">
    <input message="tns:echoIntArrayFaultRequest"/>
    <output message="tns:echoFaultResponse"/>
  </operation>
</portType>
```

CTS mapping

The WSDL shown in Example 35 maps to C# as follows:

```
//C#
void echoIntArrayFault(System.Int32[] param);
```

CHAPTER 9

Configuration

This chapter describes the configuration variables that are specific to the Artix Connect, and their associated values.

In this chapter

This chapter discusses the following topics:

Overview

Configuration domains

Artix Connect configuration variables are stored in a configuration domain. An Artix Connect configuration domain is a collection of configuration information in an Artix Connect runtime environment. This information consists of configuration variables and their values. When you install Artix Connect, you are provided with a default configuration. The default Artix Connect configuration domain file is located in:

```
ArtixConnectInstallDir/artix/Version/etc/domains/artix.cfg
```

More information

See the Deploying and Managaing Artix Solutions guide for more detail on configuring Artix.

Environment Variables

Overview

The Artix Connect installer automatically sets the environment variables that are required by Artix Connect. If, however, you chose not to set the variables during installation, you must either run the `artix_env.bat` script or set the the variables manually.

In this section

This section gives details of the variables and how to set them if you have not already set them while installing the product. The following topics are covered:

- Artix Connect Environment variables
- Running the artix_env.bat script
- Setting manually

Artix Connect Environment variables

This section describes the environment variables used by Artix Connect. They include:

- IT_PRODUCT_DIR
- IT_LICENSE_FILE
- IT_CONFIG_DOMAINS_DIR
- IT_DOMAIN_NAME
- PATH
- JETVMPROP

Note: You do not have to manually set your environment variables. You can configure them during installation, or set them later by running the provided `artix_env.bat` script.

The environment variables are explained in Table 5:

Table 5: *Artix Connect Environment Variables*

Variable	Description
`IT_PRODUCT_DIR`	`IT_PRODUCT_DIR` points to the top level of your Artix Connect installation. For example, if you install Artix Connect into the `C:\Program Files\IONA` directory, `IT_PRODUCT_DIR` should be set to that directory. **Note:** If you have other IONA products installed and you choose not to install them into the same directory tree, you must reset `IT_PRODUCT_DIR` each time you switch IONA products.
`IT_LICENSE_FILE`	`IT_LICENSE_FILE` specifies the location of your Artix Connect license file. The default value is *`ArtixConnectInstallDir`*`\etc\licenses.txt`

Table 5:*Artix Connect Environment Variables*

Variable	Description
IT_DOMAIN_NAME	IT_DOMAIN_NAME specifies the name of the configuration domain used by Artix Connect to locate its configuration. This variable also specifies the name of the file in which the configuration is stored. It should be set to artix.
IT_CONFIG_DOMAINS_DIR	IT_CONFIG_DOMAINS_DIR specifies the directory where Artix Connect searches for its configuration file, artix.cfg. It should be set to: *ArtixConnectInstallDir*\artix*Version*\etc\domains For example: C:\iona\ArtixConnect\artix\3.0\etc\domains
PATH	The Artix bin directories are added to the PATH variable to ensure that the proper configuration files, libraries, and utility programs are used. The default bin directories are: %IT_PRODUCT_DIR%\artix*Version*\bin and %IT_PRODUCT_DIR%\bin
JETVMPROP	JETVMPROP specifies where the Artix Connect license file is stored. It is required for the Artix Connect wsdltodotnet metadata generator to work. The default value is: -Dcom.iona.artix.LicenseFile=*ArtixConnectInstallDir*\etc\licenses.txt For example: -Dcom.iona.artix.LicenseFile=C:\iona\ArtixConnect\etc\licenses.txt

Running the artix_env.bat script

The Artix Connect installation process creates a script named `artix_env.bat`, which captures the information required to set your host's environment variables. Running this script configures your system to use Artix Connect. The script is located in the Artix Connect `bin` directory:

```
ArtixConnectInstallDir\artix\Version\bin
```

The `artix_env.bat` script takes the following arguments. You must specify `-compiler vc71`. The rest of the arguments described are optional:

Table 6: *Options to artix_env Script*

Option	Description
`-compiler vc71`	Enables support for Microsoft Visual Studio .NET 2003. You must specify this option.

`-preserve`	Preserves the settings of any environment variables that have already been set. When this argument is specified, `artix_env.bat` does not overwrite the values of variables that are already set. This option applies to the following environment variables:
	`IT_PRODUCT_DIR`
	`IT_LICENSE_FILE`
	`IT_CONFIG_DOMAINS_DIR`
	`IT_DOMAIN_NAME`
	`CLASSPATH`
	`PATH`
	`JETVMPROP`
	For more detailed information, see "Artix Connect Environment variables" on page 114.
	Note: Before using the `-preserve` option, always ensure that the existing environment variable values are set correctly.

`-verbose`	`artix_env.bat` outputs an audit trail of all its actions to `stdout`.

Setting manually

To set the environment variables manually:

1. Right-click on the Windows **My Computer** desktop icon and select **View system information**. The System Properties dialog box appears as shown in Figure 26:

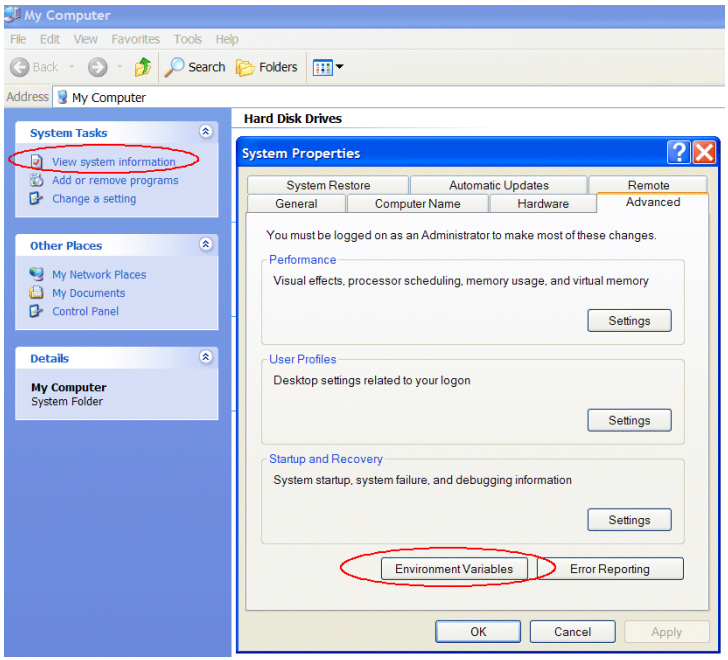

Figure 26: *Selecting My Computer*

2. Select the **Advanced** tab and cick **Environment Variables**, as shown in Figure 26. The **Environment Variables** dialog box appears as shown in Figure 27:

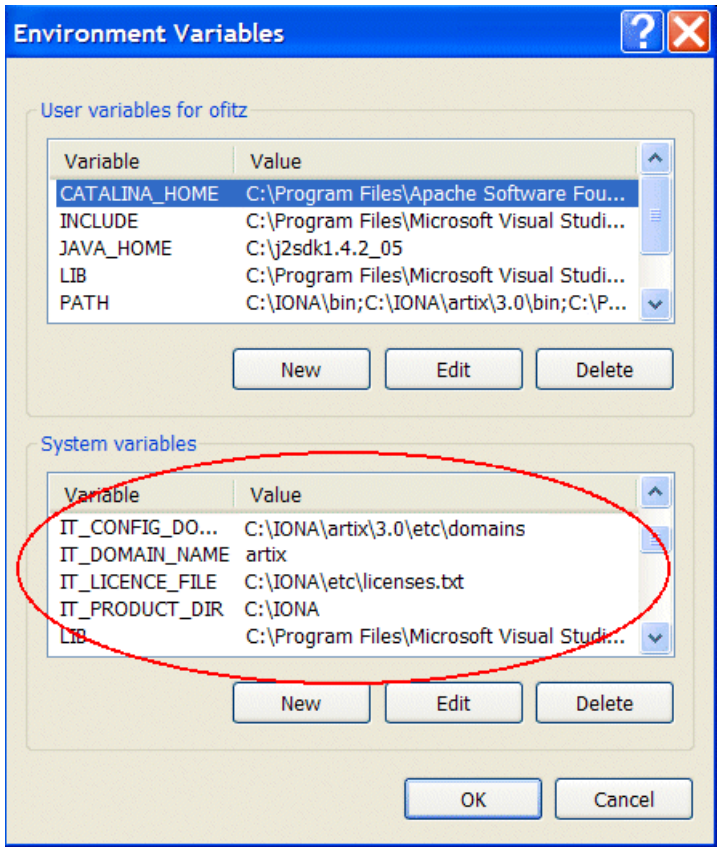

Figure 27: *Setting Environment Variables Manually*

3. Add each of the environment variables, including the correct value for your installation, as described in "Artix Connect Environment variables".

Note: The variables must be set at a system level for IIS.

Index

X