IONA®

Artix™

Artix Connect User's Guide

Version 4.0, August 2006

Making Software Work Together™

While the information in this publication is believed to be accurate, IONA Technologies PLC makes no warranty of any kind to this material including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. IONA shall not be liable for errors contained herein, or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Updated: October 18, 2006

Contents

List of Figures 7

Preface 11

Part I Introduction

Chapter 1 Introduction to Artix Connect 19
- Artix Connect Overview 20
- Artix Connect System Components 23
- Artix Connect Usage Scenarios 24
  - .NET Client Invoking on Web service using SOAP over HTTP 25
  - .NET Client Invoking on a CORBA Server using IIOP 27

Chapter 2 Getting Started 29
- Introduction 30
- Running the Hello World Demo 31
- Background Information 35

Part II Development and Deployment

Chapter 3 Development Support Tools 41
- Development Prerequisites 42
- Add Artix Reference Wizard 43
- wsdltodotnet Command-line Utility 45
- Add IDL Reference Wizard 47
- idltowsdl Command-line Utility 49

Chapter 4 Developing .NET Clients for Artix Services 51
- Generating .NET Metadata 52

Writing a Client 61
Building and Running your Client 63
Client Factory Class 65

Chapter 5 Developing .NET Clients for CORBA Services 67
Generating .NET Metadata for a CORBA Service 68
Writing a Client for a CORBA Service 78
Building and Running your Client 80
CORBA System Defined by Multiple IDL Files 82
CORBA Client-Side Runtime Memory Requirements 84
IDL Source Files 86

Chapter 6 Exposing a .NET Server as an Artix Service Endpoint 89
Generating .NET Metadata 90
Writing a C# Artix Server 97
Building and Running your Server 100

Chapter 7 Deploying an Artix Connect Application 101
Deployment Model 102
Deployment Steps 104

Part III Using Advanced Features

Chapter 8 Client Callbacks 107
Introduction to Callbacks 108
Implementing Callbacks 109
Callback Demonstration 110
Callback WSDL File 112
Implementing the Client in C# 116
Implementing the Server 119

Chapter 9 Using the Artix Locator 121
Overview of the Locator 122
Developing a .NET Client that Uses the Locator 124
Building and Running your Client 130

Chapter 10 Using the Artix Session Manager 135
Overview of the Session Manager 136
Developing a .NET Client that Uses the Session Manager 137
Building and Running your Client 143

Part IV Reference Material

Chapter 11 Introduction to WSDL 149
WSDL Basics 150
Abstract Data Type Definitions 153
Abstract Message Definitions 156
Abstract Interface Definitions 159
Mapping to the Concrete Details 162

Chapter 12 WSDL to .NET Mapping 163
Mapping a WSDL File to CTS 164
Port Types 165
Operations 167
Messages 168
Document/Literal Wrapped Style 170
Simple Types 173
Atomic Types 174
Lists 176
Unsupported Simple Types 178
Complex Types 179
Sequence and All Complex Types 180
Arrays 182
Choice Complex Type 184
Attributes 186
Attributes with In-line Data Descriptions 188
Enumerations 191
Deriving a Complex Type from a Simple Type 192
Occurrence Constraints 194
SOAP Arrays 196

Chapter 13 Configuration 197
Overview 198
Environment Variables 199

Index 205

List of Figures

Figure 1: Artix Connect Overview 21
Figure 2: .NET client invoking on SOAP over HTTP Web Service 25
Figure 3: .NET client invoking on a CORBA server over IIOP 27
Figure 4: Selecting Artix Connect Demos 32
Figure 5: Artix Connect Demos Loaded into Visual Studio .NET 2003 33
Figure 6: Running the Hello World Server 34
Figure 7: Add Artix Reference Wizard 43
Figure 8: Add IDL Reference Wizard 47
Figure 9: Selecting Artix Connect Demos 53
Figure 10: Adding a New Project to the Demo Solution 54
Figure 11: Saving New Project Details 54
Figure 12: C# Project 55
Figure 13: Launching the Add New Item Dialog Box 56
Figure 14: Launching the Add Artix Reference Wizard 57
Figure 15: Generating .NET Metadata and Client Helper Code 58
Figure 16: Required Files Added to Project 59
Figure 17: Class1.cs 61
Figure 18: Building your Client 63
Figure 19: Running the Hello World Server 64
Figure 20: Selecting Artix Connect Demos 69
Figure 21: Starting the CORBA Server—Set as StartUp Project 70
Figure 22: Starting the CORBA Server—Start Without Debugging 71
Figure 23: Adding a New Project to the dotnet Demo Solution 71
Figure 24: Saving New Project Details 72
Figure 25: C# Project 73
Figure 26: Launching the Add New Item Dialog Box 74

Figure 27: Launching the Add IDL Reference Wizard 75
Figure 28: Generating .NET Metadata and Client Helper Code for CORBA Service 76
Figure 29: Grid Client—Class1.cs 78
Figure 30: Building your Grid Client 80
Figure 31: Running your Grid Client 81
Figure 32: IDL Interfaces that Inherit fron Each Other 82
Figure 33: Selecting Artix Connect Demos 91
Figure 34: Adding a New Project to the Demo Solution 92
Figure 35: Selecting a Project Type 92
Figure 36: Launching the Add New Item Dialog Box 93
Figure 37: Launching the Add Artix Reference Wizard 94
Figure 38: *Generating .NET Metadata and Server Implementation Code* 95
Figure 39: Required Files Added to Project 96
Figure 40: Running your Server 100
Figure 41: Typical Deployment Scenario 102
Figure 42: Callback in Progress 110
Figure 43: Generating Metadata for Simple Service 125
Figure 44: Adding a Reference to your Project 126
Figure 45: Adding the Locator.dll to your Project 127
Figure 46: Creating a Post-build Event 131
Figure 47: Project Properties Dialog 132
Figure 48: Starting the Simple Service Server—Set as StartUp Project 133
Figure 49: Starting the Simple Service Server—Start Without Debugging 134
Figure 50: Generating Metadata for Session Management Service 138
Figure 51: Adding a Reference to your Project 139
Figure 52: Adding the SessionManager.dll to your Project 140
Figure 53: Creating a Post-build Event 144
Figure 54: Project Properties Dialog 145
Figure 55: Selecting My Computer 202

Figure 56: Setting Environment Variables Manually 203

Preface

Artix Connect is a .NET custom remoting channel that enables you to develop and deploy .NET clients and servers that can communicate using any of the transports and protocols 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. It 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 files. The following Artix guides are a good place to start learning:

- Getting Started with Artix
- Building Service-Oriented Architectures with Artix

In addition, the following 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:

- **Part I, Introduction,** which gives an overview of Artix Connect, its system components and some typical usage scenarios. It also describes how to get started with Artix Connect by running a simple hello world demo application.
- **Part II, Development and Deployment,** which helps to get you up and running quickly with application programming and deployment using Artix Connect. It explains the basics you need to know to develop:
    - A .NET client, written in C#, which can invoke on an existing Web service.
    - A .NET client, written in C#, which can invoke on an existing CORBA service.
    - A .NET server, written in C#, which can be invoked upon using any of the transports and protocols supported by Artix.

  It also includes a chapter on the deployment model that you can use and the steps that you should follow when deploying a distributed application with Artix Connect.
- **Part III, Using Advanced Features,** which describes how to implement client callbacks, and how to use the Artix locator and session manager services.
- **Part IV, Reference Information,** which:
    - Introduces basic WSDL concepts.
    - Gives details of the WSDL to .NET mapping used by Artix Connect.
    - Describes the environment variables that are specific to Artix Connect and their associated values.
    - Describes the client factory class that is generated by the Add Artix Reference wizard.
- **Glossary of Terms**, which explains the terminology used in this book.
- **Index**

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.

Part I

Introduction

In this part

This part contains the following chapters:

Introduction to Artix Connect	page 19
Getting Started	page 29

CHAPTER 1

Introduction to Artix Connect

Artix Connect is a custom .NET remoting channel that enables you to develop .NET clients and servers that can communicate using any of the transports and protocols supported by Artix.

In this chapter

This chapter discusses the following topics:

Artix Connect Overview	page 20
Artix Connect System Components	page 23
Artix Connect Usage Scenarios	page 24

Artix Connect Overview

What is Artix Connect?

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 and protocols supported by Artix.
- .NET servers to communicate with clients using any of the transports and protocols supported by Artix.

How does Artix Connect differ from standard .NET remoting channels?

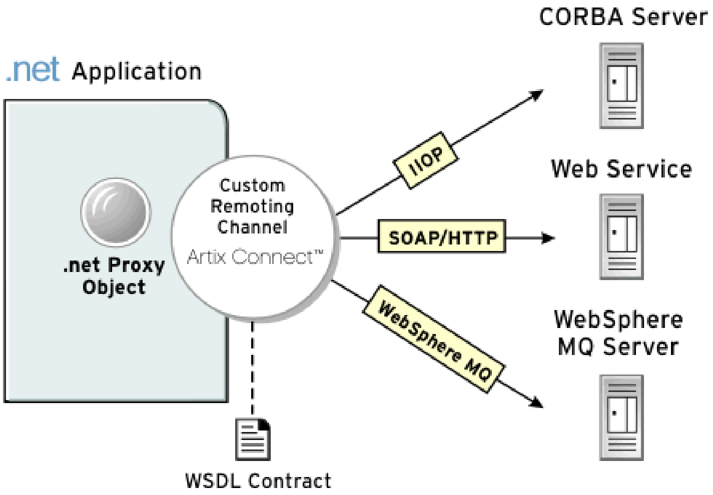
The default Microsoft .NET remoting channel only supports SOAP over HTTP and SOAP over TCP/IP. The Artix remoting channel, however, uses the Artix runtime to provide support for all of the transports and protocols that Artix supports, as well as quality of services such as security. This includes the ability to mix and match transport protocols and bindings (marshalling schemes) to enable .NET clients and servers to communicate with other technologies such as J2EE, WebSphere MQ (MQSeries), Tibco, and mainframes using native formats or SOAP over native transports.

In addition, the Artix remoting channel can be customized using Artix APIs. This is analogous to using custom sinks and formatters in .NET remoting.

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 file

To connect your .NET client or server to any of the middleware platforms supported by Artix, all Artix Connect requires is the Web Services Description Language (WSDL) file for that service.

Artix uses WSDL files 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 file 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 149.

For more information about Artix and WSDL, see the Artix 4.0 documentation, available online at:
http://www.iona.com/support/docs/artix/4.0/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
- CORBA IIOP
- 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 valid license.

Artix Connect System Components

Overview

This section describes the various components that comprise an Artix Connect system.

Bridge

Artix Connect provides a dynamic bridge for .NET in the form of a custom 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 file to formulate dynamic requests that can be invoked on the service defined in the WSDL file. 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 either the Add Artix Reference wizard or Add IDL Reference wizard, which generate .NET metadata from a WSDL file or an IDL file respectively, 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 file. Artix Connect uses the transport and protocol details contained in the WSDL file to communicate between the .NET client machine and the server. The .NET client can use any of the transports and protocols supported by Artix. The WSDL file is the only thing required by Artix Connect to enable the .NET client to successfully invoke on the server. No changes are required on the server side.

The client can be written in any language compatible with .NET, including Visual Basic .NET, Visual C++, C#, J#, and Jscript. The .NET client simply registers the `Artix.Remoting` custom remoting channel and creates a proxy for the remote service. It 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 server.

.NET server

A .NET server implemented using Artix Connect can communicate with clients using any of the transports and protocols supported by Artix. The server can be written in any language compatible with .NET, including Visual Basic .NET, Visual C++, C#, J#, and Jscript.

Artix Connect Usage Scenarios

Overview

This section gives an overview of two Artix Connect usage scenarios:

.NET Client Invoking on Web service using SOAP over HTTP	page 25
.NET Client Invoking on a CORBA Server using IIOP	page 27

.NET Client Invoking on Web service using SOAP over HTTP

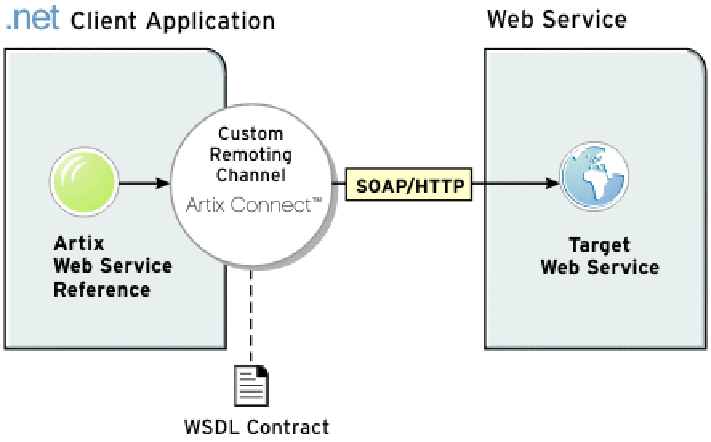
Overview

This subsection describes a scenario in which Artix Connect connects a .NET client to a Web service using SOAP over HTTP.

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 at:

http://www.iona.com/support/docs/artix/4.0/index.xml

WSDL file

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

For more details on using Artix to design WSDL files, see the Using Artix Designer guide.

.NET client and Artix Connect

Artix Connect provides an Add Artix Reference wizard that generates .NET metadata from the WSDL file 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 the 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 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:

```
InstallDir\artix\Version\demos\dotnet\hello_world
```

For details on how to run this demo, see "Getting Started" on page 29.

.NET Client Invoking on a CORBA Server using IIOP

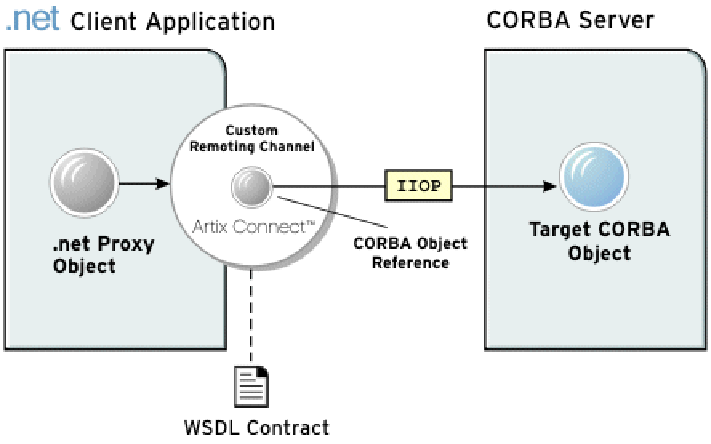
Overview

This subsection describes a scenario in which Artix Connect connects a .NET client to a CORBA server.

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 at:

http://www.iona.com/support/docs/orbix/6.3/index.xml

WSDL file

The CORBA server's interface is specified in a CORBA IDL file. The Add IDL Reference wizard was used to generate a WSDL file. The WSDL file specifies that clients should communicate with the server using IIOP. In addition, the WSDL file contains details of the CORBA server's location (IOR, `corbaname` or `corbaloc`).

.NET client and Artix Connect

The Add IDL Reference wizard also generates .NET metadata from the WSDL file 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:

```
InstallDir\artix\Version\demos\dotnet\corba_grid
```

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

For more detail on how to develop .NET clients that can communicate with CORBA servers, see "Developing .NET Clients for CORBA Services" on page 67.

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 an Artix Web service can be invoked from a standard C# .NET client using Artix Connect.

In this chapter

This chapter contains the following sections:

Introduction	page 30
Running the Hello World Demo	page 31
Background Information	page 35

Introduction

Overview

This chapter is based on running the 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. This section gives details of the prerequisites to running the demo and provides some basic details.

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:

```
InstallDir\artix\Version\demos\dotnet\hello_world
```

Running from the command line

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

Running the Hello World Demo

Overview

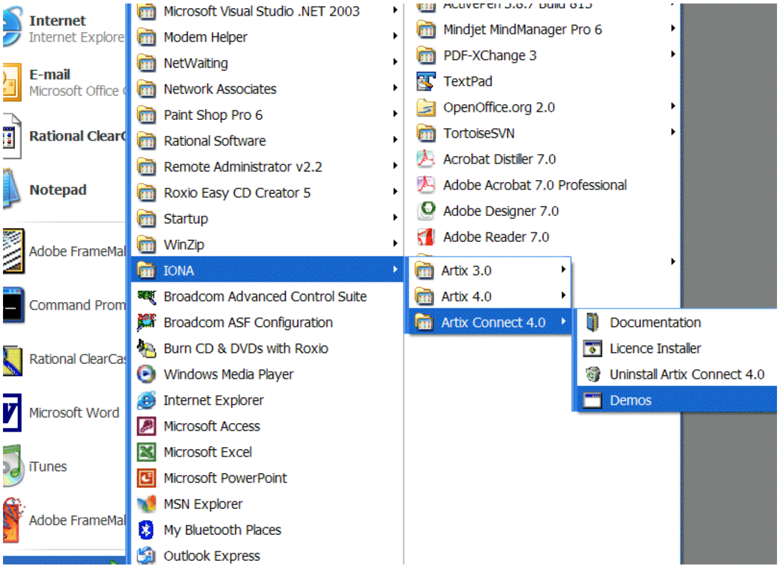
To run the Hello World demo from within the Microsoft Visual Studio .NET 2003 development environment, complete the following steps:

Step	Action
1	Open the Artix Connect Demos in Visual Studio .NET
2	Run the server
3	Run the client

Open the Artix Connect Demos in Visual Studio .NET

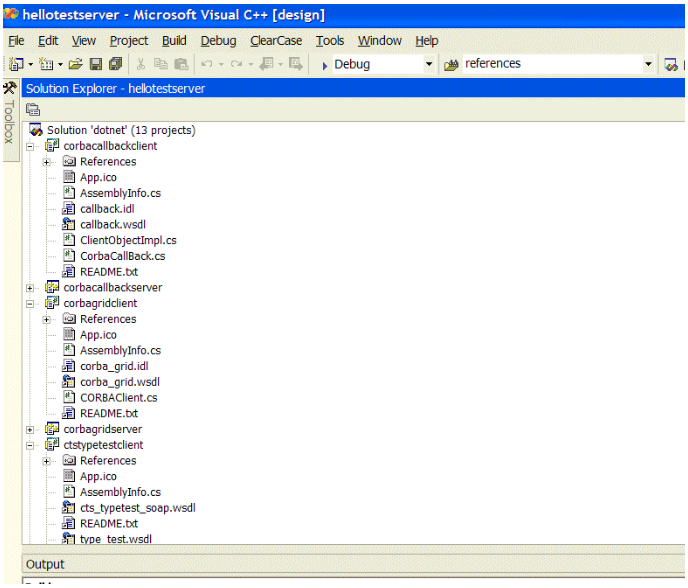
To open the Artix Connect demos in Visual Studio .NET, from the Windows **Start** menu, select the Artix Connect 4.0 **Demos**, as shown in Figure 4:

Figure 4: *Selecting Artix Connect Demos*



The demos load into the Visual Studio .NET 2003 development environment as shown in Figure 5.

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



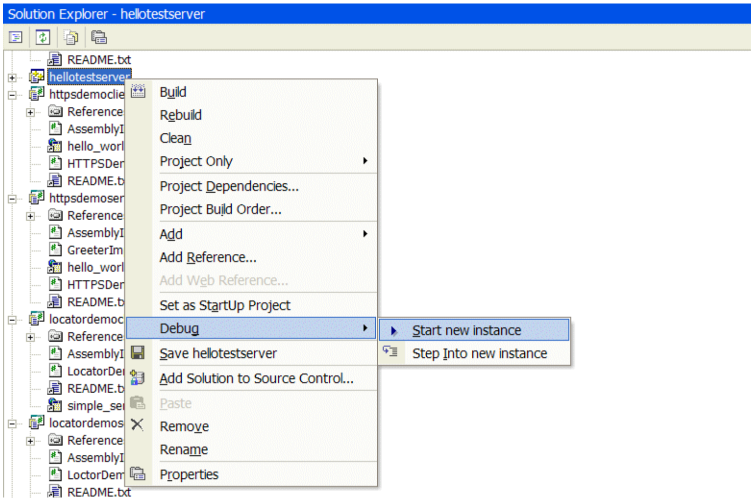
hellotestserver - Microsoft Visual C++ [design]
File Edit View Project Build Debug ClearCase Tools Window Help
Debug
references
Toolbox
Solution Explorer - hellotestserver
Output

```
Solution 'dotnet' (13 projects)
  corbacallbackclient
    References
    App.ico
    AssemblyInfo.cs
    callback.idl
    callback.wsdl
    ClientObjectImpl.cs
    CorbaCallBack.cs
    README.txt
  corbacallbackserver
  corbagridclient
    References
    App.ico
    AssemblyInfo.cs
    corba_grid.idl
    corba_grid.wsdl
    CORBAClient.cs
    README.txt
  corbagridserver
  ctstypetestclient
    References
    App.ico
    AssemblyInfo.cs
    cts_typetest_soap.wsdl
    README.txt
    type_test.wsdl
```

Run the server

To run the server, right-click on the `hellotestserver` icon and select **Debug|Start new instance**, as shown in Figure 6:

Figure 6: *Running the Hello World Server*



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

Run the client

To run the client, right-click on the `hellotestclient` icon and select **Debug|Start new instance**.

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.

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` file, which is located in:

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

The WSDL file 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 server is implemented in C++ and was developed using Artix.

For more information on Artix development, see the Artix 4.0 documentation library.

The demo also includes a C# server.

Client

The Artix Connect Add Artix Reference wizard was used to generate the type information required by the .NET client to invoke on the server. All it required was the WSDL file; 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 `HelloTest.cs` file. Application logic was added to the `HelloTest.cs` file.

For more information on developing .NET clients, see "Developing .NET Clients for Artix Services" on page 51.

WSDL file

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

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

It was designed using the Artix Designer, which is an Artix GUI. The WSDL file specifies that clients should communicate with the server using SOAP over 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 files, see the Using Artix Designer guide.

Using other transports and protocols

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

- HTTP
- CORBA IIOP
- 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 license.

The .NET client only requires the WSDL file. The transports and protocols used by deployed C# clients can be changed by simply changing the contents of the WSDL file. You might want to change transports and protocols if, for example, the SOAP over 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.

Part II

Development and Deployment

In this part

This part contains the following chapters:

Development Support Tools	page 41
Developing .NET Clients for Artix Services	page 51
Developing .NET Clients for CORBA Services	page 67
Exposing a .NET Server as an Artix Service Endpoint	page 89
Deploying an Artix Connect Application	page 101

CHAPTER 3

Development Support Tools

The first step in writing a .NET client or server that can communicate using any of the transports and protocols supported by Artix is to obtain .NET metadata, which describes the target service interfaces and types as .NET interfaces and types. Artix Connect includes an Add Artix Reference wizard and an Add IDL Reference wizard that generate the .NET metadata and the client and server starting point code for you, all from within the Visual Studio .NET 2003 development environment. In addition, Artix Connect includes command-line utilities that you can use instead of the wizards.

In this chapter

This chapter discusses the following topics:

Development Prerequisites	page 42
Add Artix Reference Wizard	page 43
wsdltodotnet Command-line Utility	page 45
Add IDL Reference Wizard	page 47
idltowsdl Command-line Utility	page 49

Development Prerequisites

Overview

This section describes the prerequisites to starting application development with Artix Connect.

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 file or IDL file that defines the service and details of where the service is running.

Adding Artix Connect to the Global Assembly Cache

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. See the Artix Connect Installation Guide for more detail.

Add Artix Reference Wizard

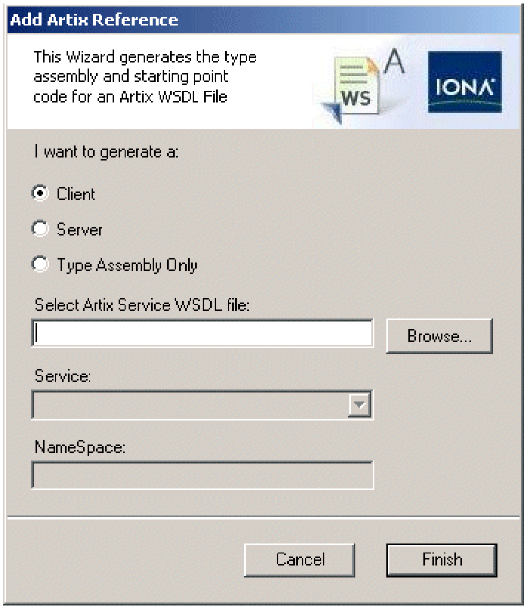
Overview

Artix Connect provides an **Add Artix Reference** wizard, which you can use to generate a .NET assembly that contains the metadata that 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 file for the service that you want to implement or to which you want your .NET client to connect. In addition to producing the .NET assembly from the WSDL file, the wizard can produce client or server starting point code that you can use to develop your application. The .NET assembly is stored in a DLL file that is generated, behind the scenes, by the `wsdltodotnet` command-line utility.

Main screen

Figure 7 shows the Add Artix Reference wizard's main screen:

Figure 7: *Add Artix Reference Wizard*



Add Artix Reference

This Wizard generates the type assembly and starting point code for an Artix WSDL File

I want to generate a:

● Client

○ Server

○ Type Assembly Only

Select Artix Service WSDL file:

Service:

NameSpace:

Fields

The Add Artix Reference wizard fields are described below:

`Client` Generates the type assembly and client starting point C# code.

`Server` Generates the type assembly and empty server implementation C# code.

`Type Assembly Only` Generates the type assembly only.

`Select Artix Service WSDL file` Specifies the WSDL filename and its location.

`Service` Specifies the name of the service that you want to your client to connect to, or that you want your server to implement. If the WSDL defines more than one service, the wizard selects the first service. If this is not the service that you want, you can use the drop-down list to select the right one.

`NameSpace` Specifies the namespace to use for the generated code. Defaults to *FirstPortTypeinWSDLfile*`NameSpace`.

Usage example

For examples of how to use the Add Artix Reference wizard, see "Developing .NET Clients for Artix Services" on page 51 and "Exposing a .NET Server as an Artix Service Endpoint" on page 89.

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 Add Artix Reference 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 WSDLFileName_types.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 WSDLFileName_types.dll by browsing to the location where you have it stored.

Generating .NET metadata assembly

You can generate the .NET metadata assembly at the command line using the following command:

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

You must specify the location of a valid WSDL, wsdlurl, for the wsdltodotnet assembly 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.

-impl Generates empty server implementation C# code. The WSDL file must include at least one service definition.

-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 file name of the generated output 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

The following are some common usage examples for the `wsdltodotnet` command-line utility. In each case the command is being run from the directory in which the WSDL file, `hello_world.wsdl`, exists; that is:

InstallDir\artix\Version\demos\dotnet\hello_world\etc

Example 1

The following command generates a .NET metadata assembly within a `.dll` file, based on the `Greeter` port type described in the `hello_world.wsdl` file:

```
wsdltodotnet hello_world.wsdl
```

Example 2

The following command generates a .NET metadata assembly within a `TestGreeter.dll` file, based on the `Greeter` port type described in the `hello_world.wsdl` file:

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

Example 3

The following command generates a C# source file, `GreeterImpl.cs`, which contains server starting-point code for the `Greeter` port type described in the `hello_world.wsdl` file:

```
wsdltodotnet -impl hello_world.wsdl
```

Add IDL Reference Wizard

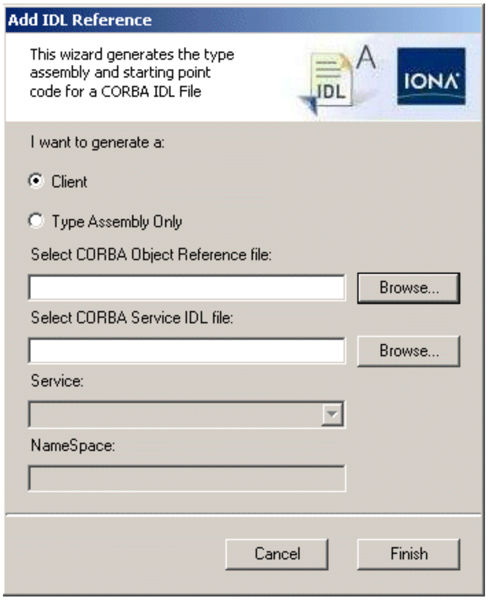
Overview

Artix Connect provides an **Add IDL Reference** wizard, which you can use to generate a .NET assembly that contains the metadata that describes the target CORBA 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 IDL file and IOR file for the CORBA service to which you want the client to connect and, as well as producing the .NET assembly, the wizard can produce client helper code that you can use to develop your client application. The .NET assembly is stored in a DLL file that is generated, behind the scenes, by the `idltowsdl` and the `wsdltodotnet` command-line utilities.

Main screen

Figure 8 shows the Add IDL Reference wizard's main screen:

Figure 8: *Add IDL Reference Wizard*



Add IDL Reference

This wizard generates the type assembly and starting point code for a CORBA IDL File

I want to generate a:

● Client

○ Type Assembly Only

Select CORBA Object Reference file:

Select CORBA Service IDL file:

Service:

NameSpace:

Fields

The Add IDL Reference wizard fields are described below:

`Client`	Generates the type assembly and client starting point C# code.
`Type Assembly Only`	Generates the type assembly only.
`Select CORBA Object Reference file`	Specifies the CORBA object reference (IOR) file.
`Select CORBA IDL file`	Specifies the CORBA IDL file.
`Service`	Specifies the name of the service that the client wants to use. If the WSDL defines more than one service, the wizard selects the first service. If this is not the service that you want, you can use the drop-down list to select the right one.
`NameSpace`	Specifies the namespace to use for the generated code. Defaults to *`FirstPortTypeinWSDLfile`*`NameSpace`.

Usage example

For an example of using the Add Artix Reference wizard, see "Developing .NET Clients for CORBA Services" on page 67

idltowsdl Command-line Utility

Overview

Artix Connect provides an `idltowsdl` command-line utility that you can use to create a WSDL file from an IDL file. You could use the `idltowsdl` command-line utility as an alternative to the Add IDL Reference wizard but, in order to generate the .NET type assembly for the CORBA service, you must subsequently run the `wsdltodotnet` command-line utility to map the WSDL file to a .NET type assembly.

Note: If you use the `wsdltodotnet` command-line utility to generate the .NET metadata, you must add the `Artix.Remoting.dll` and the *`WSDLFileName_types`*`.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 *`WSDLFileName_types`*`.dll` by browsing to the location where you have it stored.

Mapping IDL to WSDL

For details on how to use this command, see the *CORBA Utilities in Artix* chapter in the Artix for CORBA guide.

CHAPTER 4

Developing .NET Clients for Artix Services

This chapter explains how to develop a simple .NET client, written in C#, which can invoke on an existing Artix Web service. The client can use any of the transports and protocols supported by Artix and is developed from within the Visual Studio 2003 environment.

In this chapter

This chapter discusses the following topics:

Generating .NET Metadata	page 52
Writing a Client	page 61
Building and Running your Client	page 63
Client Factory Class	page 65

Generating .NET Metadata

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 .NET metadata. Artix Connect includes an Add Artix Reference wizard that generates the type assembly, which contains the .NET metadata, and client helper code from an Artix WSDL file. It does this from within the Microsoft Visual Studio .NET 2003 development environment. The .NET metadata describes the target service interface and 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.

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

For more information on using Artix to develop WSDL files, see the Using Artix Designer guide.

Demonstration code

The `Hello World` demo is used as an example application. It shows a C# .NET client invoking on an Artix Web service, using SOAP over HTTP. It is located in:

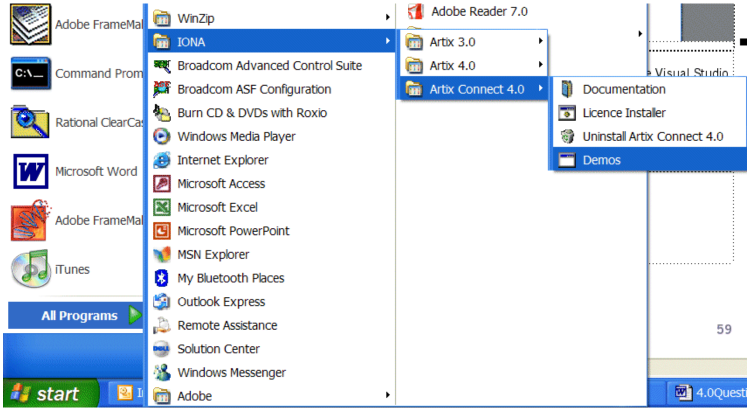
```
InstallDir\artix\Version\demos\dotnet\hello_world
```

Implementation steps

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

1. From the Windows **Start** menu, select the Artix Connect 4.0 **Demos**, as shown in Figure 9.

Figure 9: *Selecting Artix Connect Demos*

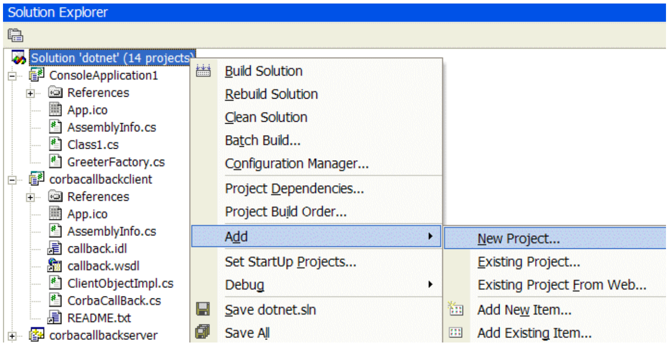


This opens the Artix Connect demos solution within the Visual Studio environment.

Note: You do not have to add your new project to the Artix Connect demo solution. Doing so, however, provides you with a server that you can use to test the client implementation that you are developing in this chapter.

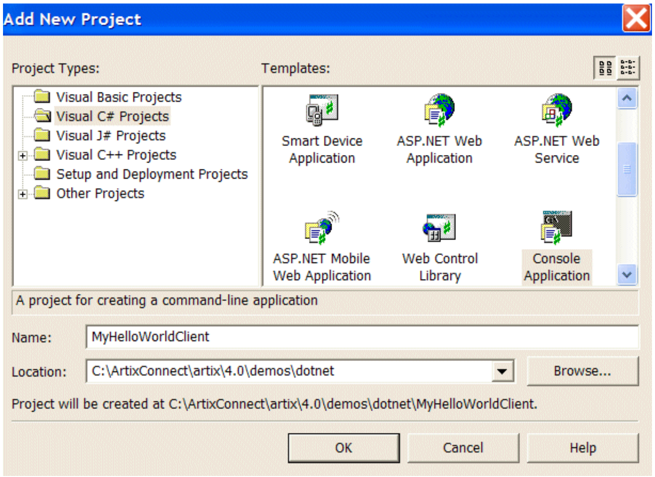
2. Right-click on the dotnet solution file and select **Add | New Project** to add a new project, as shown in Figure 10:

Figure 10: *Adding a New Project to the Demo Solution*



3. The **New Project** dialog box appears as shown in Figure 11. Select the project type that you want to create—in this example, a Visual C# project using the Console Application template:

Figure 11: *Saving New Project Details*



A project for creating a command-line application

Name: MyHelloWorldClient

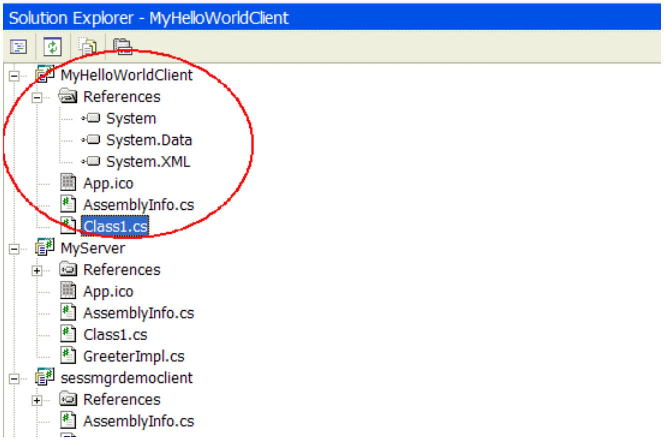
Location: C:\ArtixConnect\artix\4.0\demos\dotnet

Project will be created at C:\ArtixConnect\artix\4.0\demos\dotnet\MyHelloWorldClient.

Note: The Add Artix Reference wizard supports C# only. For projects that 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 45 for more detail.

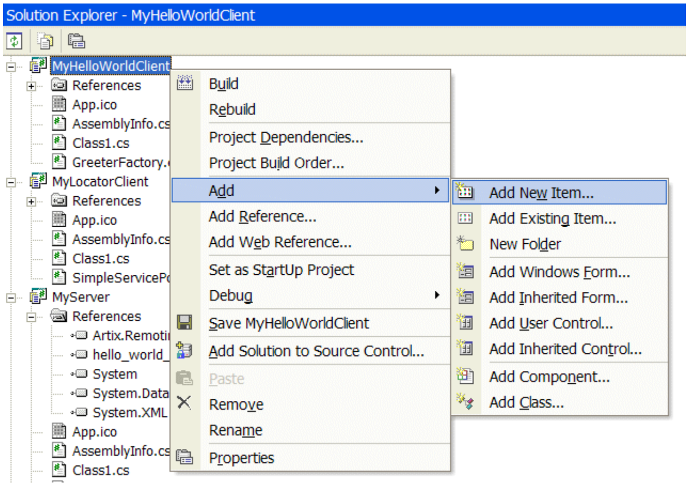
4. Enter a name for your project and select the Artix Connect demo directory as the location for storing your project (as shown in Figure 11).
5. Click **OK**. The Visual Studio .NET 2003 Development Environment creates a C# project, as shown in Figure 12:

Figure 12: *C# Project*



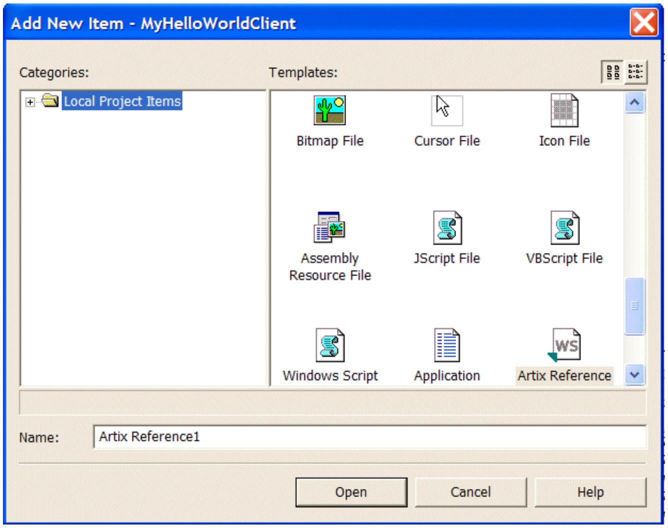
6. Next you need to add an Artix reference to the project. To do this, right-click on your project and select **Add | Add New Item**, as shown in Figure 13:

Figure 13: *Launching the Add New Item Dialog Box*



7. The **Add New Item** dialog box appears as shown in Figure 14. Select the **Artix Reference** option and click **Open**:

Figure 14: *Launching the Add Artix Reference Wizard*



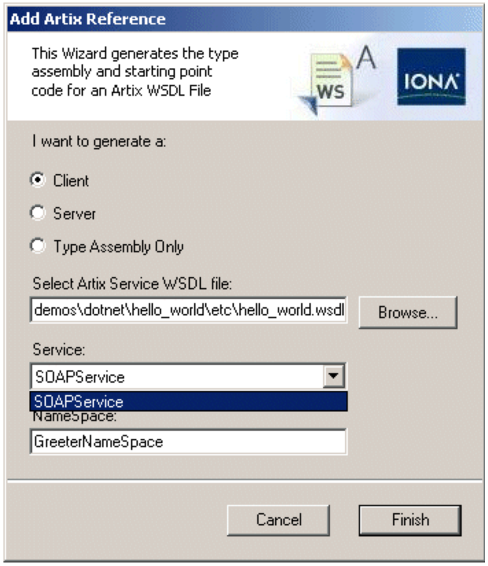
Name: Artix Reference1

8. The **Add Artix Reference** wizard appears as shown in Figure 15 on page 58. You need to:

   i. Select the **Client** radio button.

   ii. Browse for the WSDL file associated with the Artix service to which you want the client to connect. In this example, the `hello_world.wsdl` file, located in `InstallDir\Artix\Version\demos\dotnet\hello_world\etc` is selected.

Figure 15: *Generating .NET Metadata and Client Helper Code*



Add Artix Reference

This Wizard generates the type assembly and starting point code for an Artix WSDL File

I want to generate a:

● Client

○ Server

○ Type Assembly Only

Select Artix Service WSDL file: demos\dotnet\hello_world\etc\hello_world.wsdl

Service: SOAPService

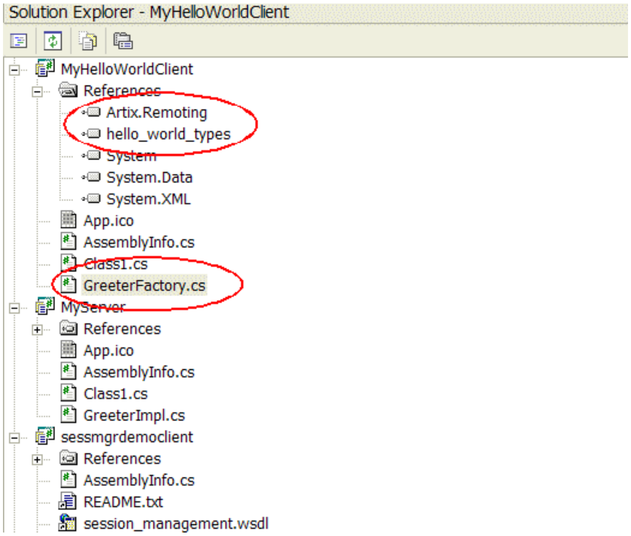
NameSpace: GreeterNameSpace

The **Add Artix Reference** wizard fills in the **Service** and **NameSpace** fields with values taken from the WSDL file. If the WSDL file contains more than one service, the wizard selects the first service. Alternatively, you can select the service that you want from the drop-down list.

9. Click **Finish** to import the WSDL file and generate the .NET type assembly and client helper code for the Artix service.

The **Add Artix Reference** wizard adds three required items to the client project, as shown in Figure 16:

Figure 16: *Required Files Added to Project*



It adds the following references:

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

And the following file:

- Client factory class—in this example, `GreeterFactory.cs`. The client factory class uses standard .NET remoting code to get a reference to the Artix service. You can either invoke on the factory class' `Create()` method to get a reference to the Artix service or copy the code from the factory class and paste it directly into your client mainline. For more information on the contents of the factory class, see "Client Factory Class" on page 65.

Note: If you use the `wsdltodotnet` command-line utility to generate the types assembly, you must add the `Artix.Remoting.dll` and the *`WSDLFileName_types`*`.dll` (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 *`WSDLFileName_types`*`.dll` by browsing to the location where you have it stored .

Writing a Client

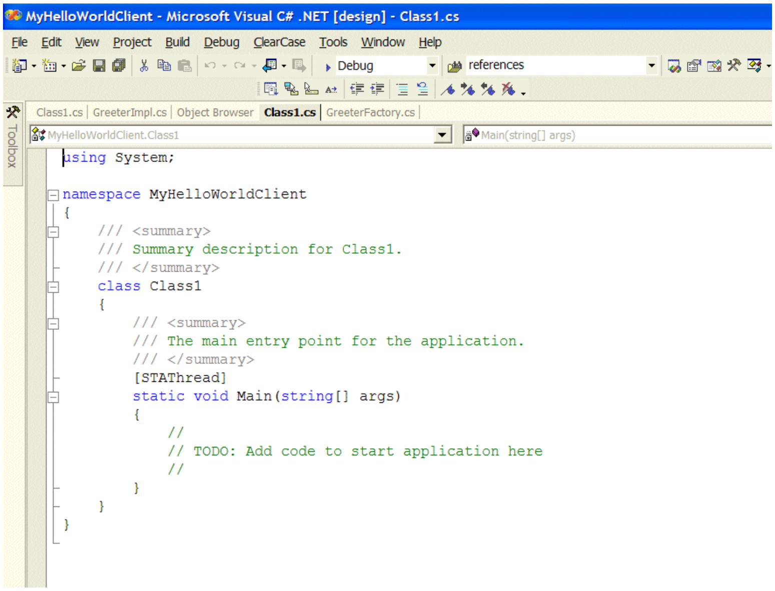
Overview

As shown in the previous section, "Generating .NET Metadata", the Add Artix Reference wizard generates a client factory class that contains a `Create()` method that returns a reference to the Artix service. To complete the client you can either add code that calls `Create()` on the factory and invokes on the Artix service, or you can copy the code from the factory class and paste it directly into your client mainline. This section documents how to call the factory class' `Create()` method. For more information on the factory class, see "Client Factory Class" on page 65.

Client mainline class

In this example, the client mainline file is called `Class1.cs` and is shown in Figure 17.

Figure 17: *Class1.cs*



```
using System;

namespace MyHelloWorldClient
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
        }
    }
}
```

To complete the client you must write the code that invokes on the service. For instance, Example 1 shows code that invokes on the hello world service’s `greetMe()` operation:

Example 1: *.NET Client Invoking on an Artix Service*

```
  // C#
  {
1 GreeterNameSpace.Greeter obj = GreeterFactory.Create();
2 Console.WriteLine(obj.greetMe(".NET Connector"));

  Console.WriteLine("Press Enter to exit.");
  Console.ReadLine();}
  }
```

The code shown in Example 1 can be explained as follows:

1. Calls `Create()` on the factory—in this example, `GreeterFactory`. The client factory class uses standard .NET remoting code to get a reference to the Artix service. For more information on the factory class, see “Client Factory Class” on page 65.
2. Invokes on the service—in this case, invokes on the `greetMe()` operation. You can invoke on any of the available operations.

Building and Running your Client

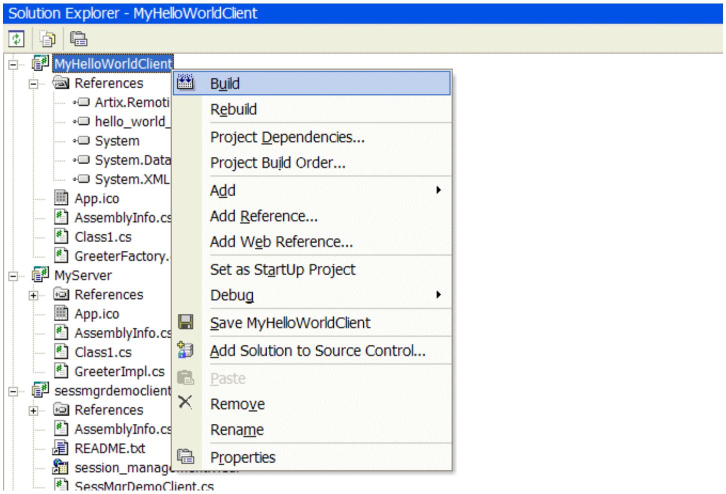
Overview

This section describes how to build and run the client that you wrote in the previous section, Writing a Client.

Building the client

To build the client, highlight your client project and select **Build**, as shown in Figure 18:

Figure 18: *Building your Client*

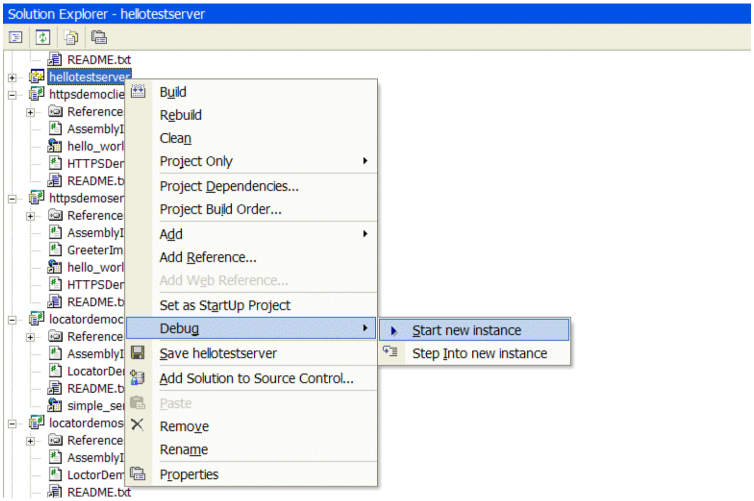


Running the client

To run your client:

1. Run the server by right-clicking on the `hellotestserver` icon and selecting **Debug|Start new instance**, as shown in Figure 19:

Figure 19: *Running the Hello World Server*



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

2. Run the client by right-clicking on your client project icon and selecting **Debug|Start new instance**.

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

Client Factory Class

Overview

This section describes the code in the client factory class. The client factory class uses standard .NET remoting code to get a reference to the service that the client wants to invoke.

Code example

Example 2 shows the contents of the `GreeterFactory.cs` class, produced by the Add Artix Reference wizard for the `hello_world.wsdl` file.

Example 2: *Client Factory Class—GreeterFactory.cs*

```
   using System;
   using System.Diagnostics;
   using System.Runtime.Remoting.Channels;
   using IONA.Remoting;
   using GreeterNameSpace;

   public class GreeterFactory
   {
     public static Greeter Create()
       {
         Greeter obj = null;
         try
         {
          if (null == ChannelServices.GetChannel("Artix.Remoting"))
           {
1            ChannelServices.RegisterChannel(new
             ArtixClientChannel());
           }
2          obj = (Greeter)
3        Activator.GetObject(typeof(Greeter),
4     "artixref:C:\\ArtixConnect\\artix\\4.0\\demos\\dotnet\\hello_
      world\\etc\\hello_world.wsdl
      http://www.iona.com/hello_world_soap_http SOAPService
      SoapPort");
         }
         catch (Exception ex)
         {
         Console.Error.WriteLine("Exception: " + ex.ToString());
         }
         return obj;
      }
```

Example 2: *Client Factory Class—GreeterFactory.cs*

```
}
```

The code shown in Example 2 on page 65 can be explained as follows:

1. Registers the remoting channel that the client wants to use. The custom remoting channel is registered in the same way as any other .NET remoting channel. The 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.
2. Creates a proxy instance of the remote target object in the client's address space.
3. 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`).
4. The call to `GetObject()` also specifies an Artix reference, which points the client to the WSDL file 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 file—in this example, the `hello_world.wsdl`.
    ii. The target namespace—in this example, `http://www.iona.com/hello_world_soap_http`. This is taken from the WSDL file.
    iii. The name of the service that the client wants to use—in this example, `SOAPService`. This is taken from the WSDL file.
    iv. The name of the port that the client wants to use—in this example, `SoapPort`. This is taken from the WSDL file.

CHAPTER 5

Developing .NET Clients for CORBA Services

This chapter explains how to develop a simple .NET client, written in C#, which can invoke on an existing CORBA service. The client is developed from within the Visual Studio 2003 environment.

In this chapter

This chapter discusses the following topics:

Generating .NET Metadata for a CORBA Service	page 68
Writing a Client for a CORBA Service	page 78
Building and Running your Client	page 80

Generating .NET Metadata for a CORBA Service

Overview

The first task in implementing a .NET client that can communicate with a CORBA server, is to generate the .NET metadata that describes the target service interface. Artix Connect includes a GUI, the Add IDL Reference wizard, which enables you to generate .NET metadata and client helper code from an IDL file from within the Microsoft Visual Studio .NET 2003 development environment.

Demonstration code

The `Grid` demo is used as an example application. It shows a C# .NET client invoking on an CORBA service using IIOP. It is located in:

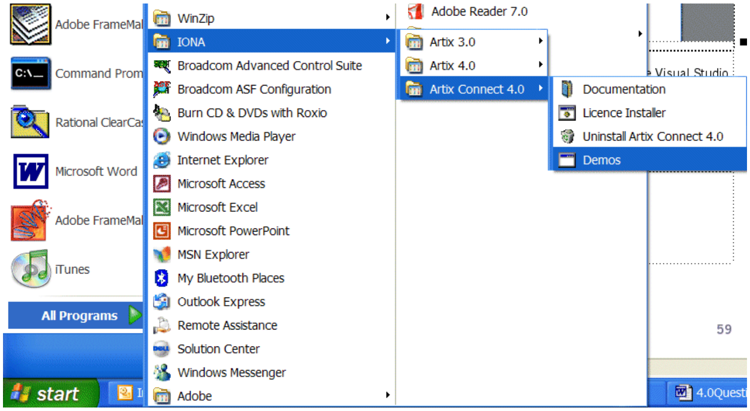
```
InstallDir\artix\Version\demos\dotnet\corba_grid
```

Implementation steps

To generate .NET metadata from within the Microsoft Visual Studio .NET 2003 development environment, using the Add IDL Reference wizard, do the following:

1. From the Windows **Start** menu, select the Artix Connect 4.0 **Demos**, as shown in Figure 20.

Figure 20: *Selecting Artix Connect Demos*



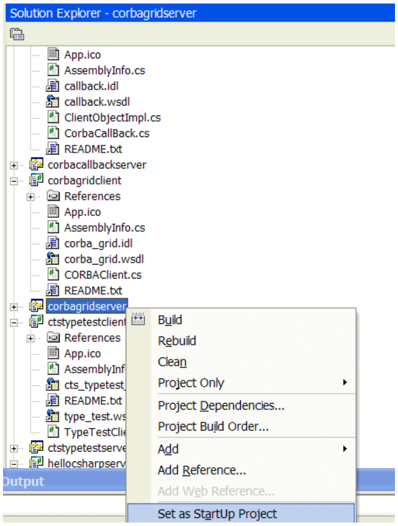
This opens the Artix Connect demos solution within the Visual Studio environment.

Note: You do not have to add your new project to the Artix Connect demo solution. Doing so, however, provides you with a CORBA server that you can use to test the client implementation that you are developing in this chapter.

2. Start the CORBA server by:

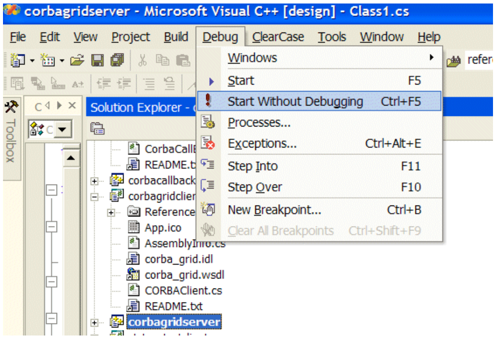
   i. Right-clicking on `corbagridserver` project and selecting **Set as StartUp Project**, as shown in Figure 21:

Figure 21: *Starting the CORBA Server—Set as StartUp Project*



ii. Selecting **Debug|Start Without Debugging**, as shown in Figure 22:

Figure 22: *Starting the CORBA Server—Start Without Debugging*

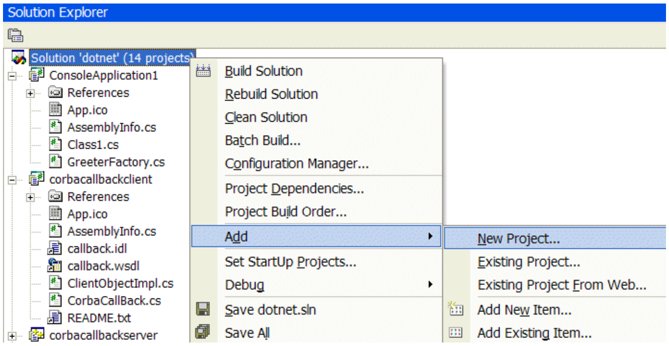


The CORBA server starts in a new command window and writes its stringified object reference (IOR) to the `grid_corba_service.ior` file in the following directory:

`InstallDir\artix\Version\demos\dotnet\corba_grid\etc`

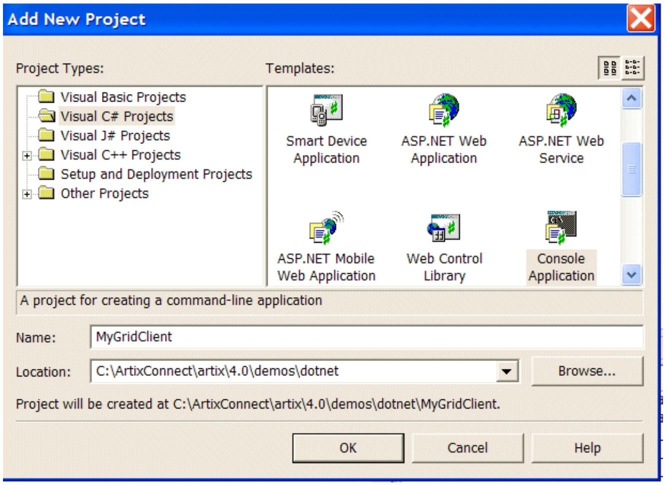
3. Create your client project by right-clicking on the dotnet solution file and select **Add | New Project** to add a new project, as shown in Figure 23:

Figure 23: *Adding a New Project to the dotnet Demo Solution*



4. The **Add New Project** dialog box appears as shown in Figure 24. Select the project type that you want to create—in this example, a Visual C# project using the Console Application template:

Figure 24: *Saving New Project Details*



A project for creating a command-line application

Name: MyGridClient

Location: C:\ArtixConnect\artix\4.0\demos\dotnet

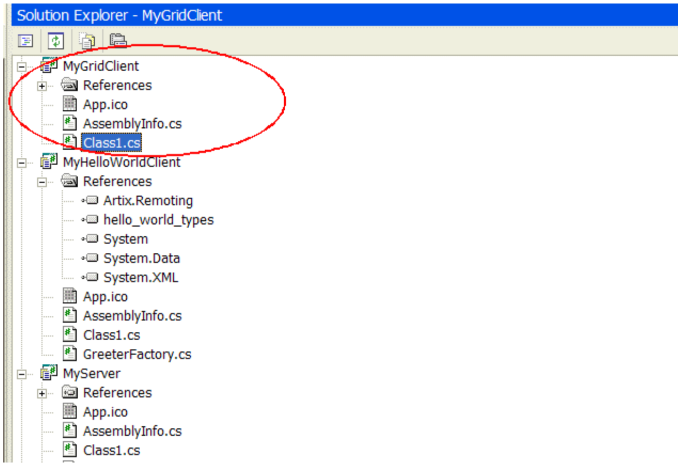
Project will be created at C:\ArtixConnect\artix\4.0\demos\dotnet\MyGridClient.

Note: The Add IDL Reference wizard supports C# only. For projects that use other languages, you should use the `idltowsdl` command-line utility, followed by the `wsdltodotnet` command-line utility to generate the .NET metadata. See “idltowsdl Command-line Utility” on page 49 and the “wsdltodotnet Command-line Utility” on page 45 for more detail.

5. Enter a name for your project and select the Artix Connect demo directory as the location for storing your project (as shown in Figure 24 on page 72).

6. Click **OK**. The Visual Studio .NET 2003 Development Environment creates a C# project

Figure 25: *C# Project*

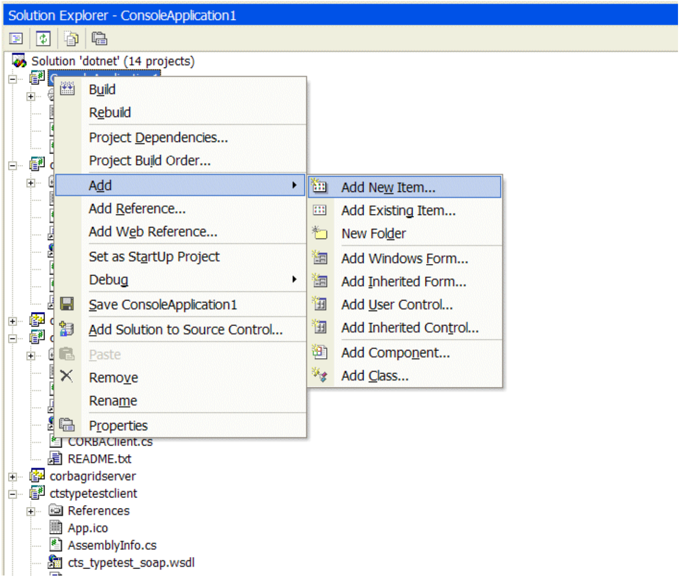


Solution Explorer - MyGridClient

- MyGridClient
  - References
  - App.ico
  - AssemblyInfo.cs
  - Class1.cs
- MyHelloWorldClient
  - References
    - Artix.Remoting
    - hello_world_types
    - System
    - System.Data
    - System.XML
  - App.ico
  - AssemblyInfo.cs
  - Class1.cs
  - GreeterFactory.cs
- MyServer
  - References
  - App.ico
  - AssemblyInfo.cs
  - Class1.cs

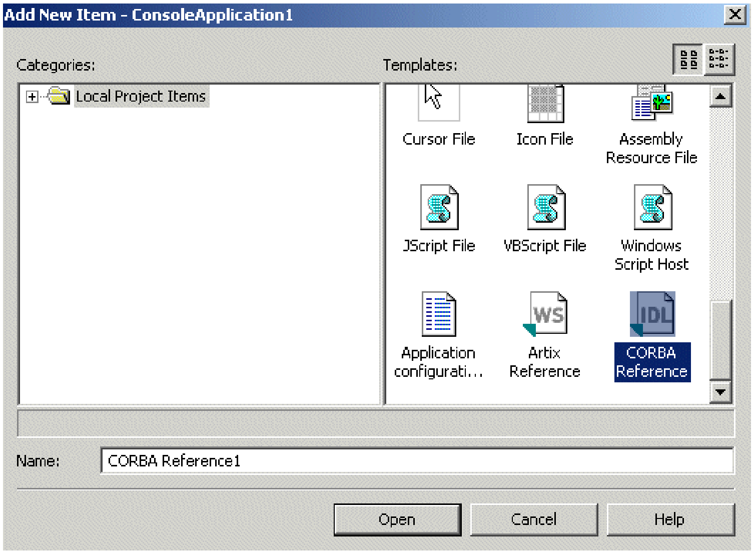
7. Next you need to add a CORBA reference to your project. To do this, right-click on your project and select **Add | Add New Item**, as shown in Figure 26, to launch the **Add New Item** dialog box:

Figure 26: *Launching the Add New Item Dialog Box*



8. The **Add New Item** dialog box appears as shown in Figure 27. Select the **CORBA Reference** option and click **Open**.

Figure 27: *Launching the Add IDL Reference Wizard*



Name: CORBA Reference1

9. The **Add IDL Reference** wizard appears as shown in Example 28 on page 76. You need to:

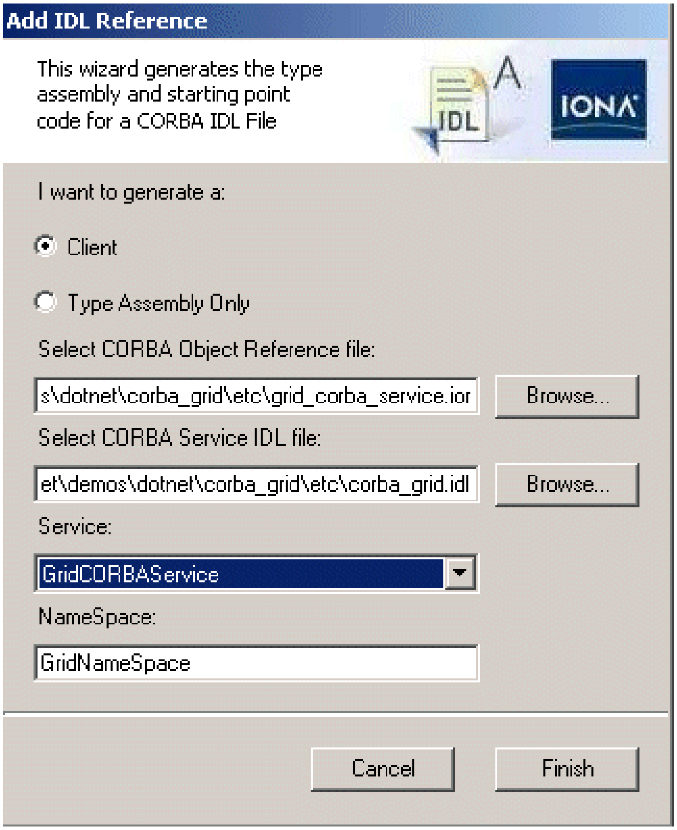
   i. Select the **Client** radio button to indicate that you want the wizard to generate the .NET type assembly and client helper code.

   ii. Browse for the CORBA object reference file for the CORBA service to which you want your client to connect. In this example, select the IOR file for the CORBA server that you started in step **2**; that is, the `grid_corba_service.ior` file located in

   `*InstallDir*\Artix\*Version*\demos\dotnet\corba_grid\etc`

iii. Browse for the IDL file for the CORBA service. In this example, the `corba_grid.idl` located in
`InstallDir\Artix\Version\demos\dotnet\corba_grid\etc`

Figure 28: *Generating .NET Metadata and Client Helper Code for CORBA Service*



Add IDL Reference

This wizard generates the type assembly and starting point code for a CORBA IDL File

I want to generate a:

● Client

○ Type Assembly Only

Select CORBA Object Reference file: s\dotnet\corba_grid\etc\grid_corba_service.ior

Select CORBA Service IDL file: et\demos\dotnet\corba_grid\etc\corba_grid.idl

Service: GridCORBAService

NameSpace: GridNameSpace

The **Add IDL Reference** wizard fills in the **Service** and **NameSpace** fields with values taken from the service WSDL file. If the WSDL file contains more than one service, the wizard selects the first service. Alternatively, you can select the service that you want from the drop-down list.

10. Click **Finish** to generate the metadata and client helper code for this service.
11. The **Add IDL Reference** wizard adds three required items to the client project, including the following references:
    - The `Artix.Remoting` assembly, which is required at runtime by all Artix Connect clients.
    - The *`WSDLFileName_types`*`.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 `corba_grid_types.dll`.

    And the following file:

    - Client factory class—in this example, `GridFactory.cs`. The client factory class uses standard .NET remoting code to get a reference to the CORBA service. You can either invoke on the factory class' `Create()` method to get a reference to the CORBA service or copy the code from the factory class and paste it directly into your client mainline. For more information on the contents of the factory class, see "Client Factory Class" on page 65.

Note: If you use the `wsdltodotnet` command-line utility to generate the types assembly, you must add the `Artix.Remoting.dll` and the *`WSDLFileName_types`*`.dll` (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 *`WSDLFileName_types`*`.dll` by browsing to the location where you have it stored.

Writing a Client for a CORBA Service

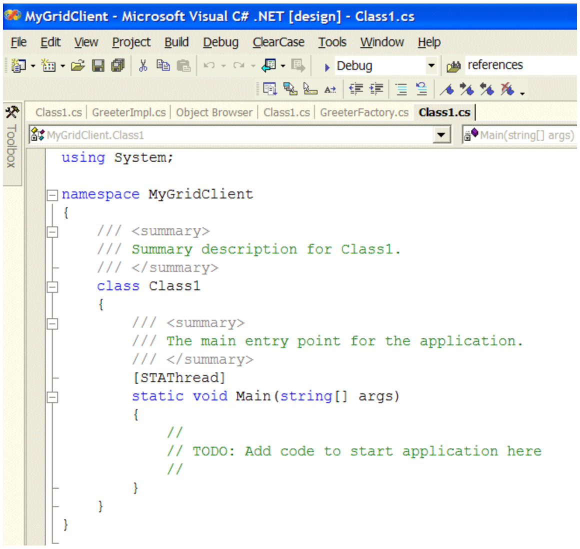
Overview

The next task in implementing a .NET client that can communicate with a CORBA service is to write the C# client. As shown in the previous section, "Generating .NET Metadata for a CORBA Service", the Add IDL Reference wizard generates a client factory class that contains a `Create()` method that returns a reference to the CORBA service. To complete the client you can either add code that calls `Create()` on the factory and invokes on the CORBA service, or you can copy the code from the factory class and paste it directly into your client mainline. This section documents how to call the factory class' `Create()` method. For more information on the factory class, see "Client Factory Class" on page 65.

Client mainline class

In this example, the client mainline file is called `Class1.cs` and is shown in Example 29.

Figure 29: *Grid Client—Class1.cs*


MyGridClient - Microsoft Visual C# .NET [design] - Class1.cs
File Edit View Project Build Debug ClearCase Tools Window Help
Debug
references
Toolbox
Class1.cs | GreeterImpl.cs | Object Browser | Class1.cs | GreeterFactory.cs | Class1.cs
MyGridClient.Class1
Main(string[] args)

```
using System;

namespace MyGridClient
{
    /// <summary>
    /// Summary description for Class1.
    /// </summary>
    class Class1
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main(string[] args)
        {
            //
            // TODO: Add code to start application here
            //
        }
    }
}
```

To complete the client, you must add code to invoke on the service. For instance, you can have the client invoke on the Grid service's `set()` and `get()` operations by adding the code shown Example 3:

Example 3: *.NET Client Invoking on a CORBA Service*

```
   //C#
   ...
   Console.WriteLine("Grid Client Starting");

1  GridNameSpace.Grid obj = GridFactory.Create();

   System.Int16 column = 5;
   System.Int16 row = 5;

2  obj.set(row, column, 99);

   System.Int32 val = obj.get(row, column);

   Console.WriteLine("Grid contents at location " + row + " " +
      column + " is " + val);

  Console.WriteLine("Press Enter to exit.");
  Console.ReadLine();
```

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

1. Calls `Create()` on the factory—in this example, `GridFactory`. The factory creates a proxy for the CORBA service. The client factory class uses standard .NET remoting code to get a reference to the CORBA service. For more information on the factory class, see "Client Factory Class" on page 65.
2. Invokes on the service—in this case, invokes on the Grid service's `get()` and set() operations. You can invoke on any of the available operations.

Building and Running your Client

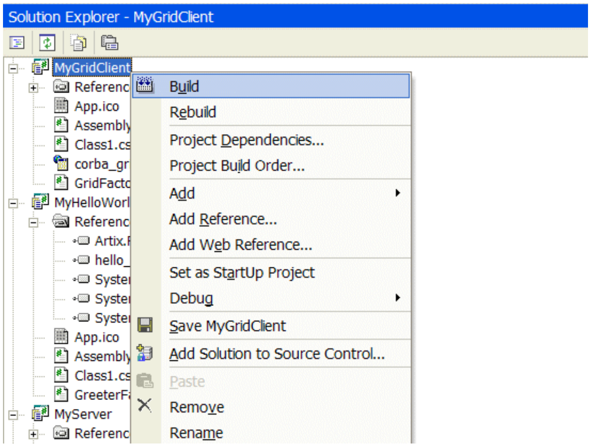
Overview

This section describes how to build the client that you wrote in the previous section, Writing a Client for a CORBA Service.

Building the client

To build the client, right-click on your client project and select **Build**, as shown in Figure 30:

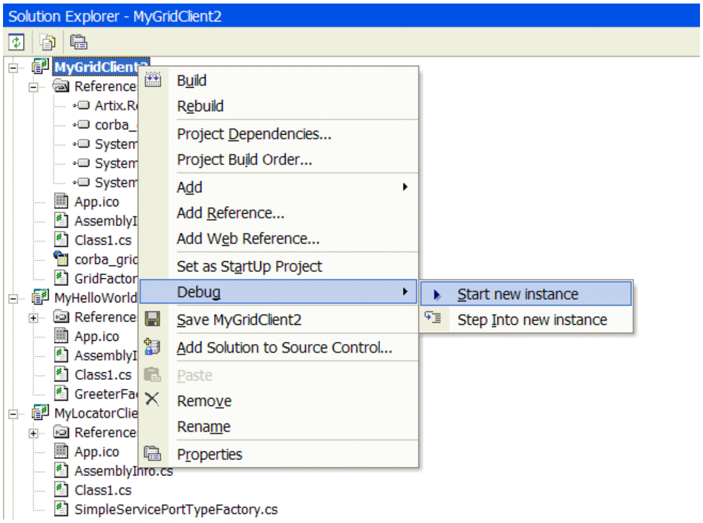
Figure 30: *Building your Grid Client*



Running the client

To run your client, right-click on your client project and select **Debug|Start new instance**, as shown in Figure 31:

Figure 31: *Running your Grid Client*



The client starts in a new DOS command window, invokes on the server and prints the following to the screen:

```
Grid Client Starting
Grid contents at location 5 5 is 99
Press Enter to exit.
```

CORBA System Defined by Multiple IDL Files

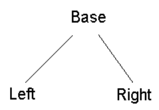
Overview

The simplest way to generate a .NET C# CORBA client is to use the **Add IDL Reference** wizard, as described in "Generating .NET Metadata for a CORBA Service" on page 68. The wizard generates a type assembly for each IDL file. If, however, a client makes invocations on multiple CORBA server objects that are defined in a hierarchy of IDL files that inherit from each other, this can result in the same types being defined multiple times in different client-side type assemblies. Although this does not cause any runtime problems for Artix Connect—the client uses the first instance of the type it encounters—it can waste memory.

Example Solution

One way to avoid the same types being defined multiple times in different client-side type assemblies is to use the Artix Connect command-line tools. You can write a makefile that uses the command-line tools to filter out the extraneous types before making the type DLL. This section outlines the contents of such a makefile, using the IDL inheritance example shown in Figure 32.

Figure 32: *IDL Interfaces that Inherit fron Each Other*



Each interface is defined in its own IDL file—`base.idl`, `left.idl`, `right.idl`. The IDL source files are shown in "IDL Source Files" on page 86.

Example makefile

Create a makefile that filters out extraneous types before making the type DLL. Example 4 shows the relevent lines of code.

Example 4: *Makefile that Generates Type Assembly*

```
   ...

1  idltowsdl left.idl
   idltowsdl right.idl

2  wsdltodotnet -source left.wsdl
   wsdltodotnet -source right.wsdl

3  csc /target:library *
```

The code shown in Example 4 can be explained as follows:

1. The `idltowsdl` command generates a WSDL file from a supplied IDL file. In this case it is run twice, once for the `left.idl` file and once for the `right.idl` file, and generates one WSDL files for each IDL file.
2. The `wsdltodotnet` command generates a `.cs` source file from a supplied WSDL file. The `-source` command-line option generates the source `.cs` C# file for the types and does not generate the type assembly. The second `wsdltodotnet` command generates some identical duplicate `Base.cs` source files, which overwrite those files generated by the first command. This is harmless as the source files are identical.
3. The standard .NET compile command. Compiles the type assembly (`.dll` file).

You can inspect the DLL to confirm that it contains all of the types, and only one instance of each type, from the multiple IDL files by running the following command:

```
ildasm Base.dll
```

CORBA Client-Side Runtime Memory Requirements

Overview

Table 1 shows the plug-ins that are used by the client (in addition to the standard Windows and .NET runtime libraries):

Table 1: *Client-side Plug-ins*

Library	Description	Size
`corbagridclient.exe`	Client mainline	About 5k. Varies depending on client code.
`Grid.dll`	Client-side type assembly containing server types.	About 5k for a small interface with two operations and two types.
`Artix.Remoting.dll`	Artix Connect runtime.	About 300k.
`it_afc5_vc71.dll`	Artix runtime library.	1,679 k
`it_art5_vc71.dll`	Artix runtime library.	4,272 k
`it_atli25_vc71.dll`	Artix runtime library.	151 k
`it_atli2_iop5_vc71.dll`	Artix runtime library.	147 k
`it_atli2_ip5_vc71.dll`	Artix runtime library.	294 k
`it_bus5_vc71.dll`	Artix runtime library.	1,372 k
`it_bus_xml5_vc71.dll`	Artix runtime library.	307 k
`it_codeset5_vc71.dll`	Optional	
`it_context_attribute5_vc71.dll`		1,499 k
`it_corba_common5_vc71.dll`		94 k
`it_csi5_vc71.dll`	Optional	
`it_dynany5_vc71.dll`	Optional	
`it_giop5_vc71.dll`	Artix runtime library.	671 k
`it_icudata2.dll`	Optional	

Table 1: *Client-side Plug-ins*

Library	Description	Size
`it_icudata2_vc71.dll`	Optional	
`it_icuuc2.dll`	Optional	
`it_icuuc2_vc71.dll`	Optional	
`it_ifc5_vc71.dll`	Artix runtime library.	368 k
`it_iiop5_vc71.dll`	Artix runtime library.	94 k
`it_iiop_profile5_vc71.dll`	Artix runtime library.	192 k
`it_location5_vc71.dll`	Optional	
`it_naming5_vc71.dll`	Optional	
`it_ots5_vc71.dll`	Optional	
`it_plain_text_key5_vc71.dll`	Optional	
`it_poa5_vc71.dll`	Optional	
`it_sm5_vc71.dll`	Optional	
`it_tls_atli25_vc71.dll`	Optional	
`it_ws_orb5_vc71.dll`	Optional	
`it_wsdl5_vc71.dll`	Artix runtime library.	659 k
`it_xerces2_vc71.dll`	Optional	

Note: To make configuration easier, the Artix Connect demos load all of the Artix plug-ins by default. As a result, profiling an Artix Connect demo, using the configuration that it uses, shows far greater memory usage than is required. Total memory used by Artix Connect (standard Windows DLLs not included) for a minimal CORBA only client runtime is about 12 MB.

IDL Source Files

Overview

The IDL source files used in the example in this section are shown below.

base.idl

The `base.idl` file is shown in Example 5:

Example 5: *base.idl*

```
#ifndef _BASE_IDL_
#define _BASE_IDL_

// *************************************************************
//
//
// Copyright (c) 2006 IONA Technologies PLC.
// All Rights Reserved.
//
//
// *************************************************************

interface Base
{
    typedef string a_base_type;

    a_base_type a_base_operation();
};

#endif  /*!_BASE_IDL_*/
```

left.idl

The `left.idl` file is shown in Example 6:

Example 6: *left.idl*

```
#ifndef _LEFT_IDL_
#define _LEFT_IDL_

// **************************************************************
//
//
//  Copyright (c) 2006 IONA Technologies PLC.
// All Rights Reserved.
//
//
// **************************************************************
#include "base.idl"

interface Left
{
    typedef string a_left_type;

    Base::a_base_type a_left_operation();
};

#endif  /*!_LEFT_IDL_*/
```

right.idl

The `right.idl` file is shown in Example 7:

Example 7: *right.idl*

```
#ifndef _RIGHT_IDL_
#define _RIGHT_IDL_

// *************************************************************
//
//
//  Copyright (c) 2006 IONA Technologies PLC.
// All Rights Reserved.
//
//
// *************************************************************

#include "base.idl"

interface Right
{
    typedef string a_right_type;

    Base::a_base_type a_right_operation();
};

#endif  /*!_RIGHT_IDL_*/
```

CHAPTER 6

Exposing a .NET Server as an Artix Service Endpoint

This chapter explains how to expose a .NET server as an Artix service endpoint. An Artix service endpoint is an endpoint that implements the business logic defined in a WSDL document and can be accessed using any of the transports and protocols supported by Artix.

Overview

This chapter contains the following sections:

Generating .NET Metadata	page 90
Writing a C# Artix Server	page 97
Building and Running your Server	page 100

Note: To expose a .NET server as an Artix service endpoint, you must have a valid license.

Generating .NET Metadata

Overview

The first task in implementing a .NET server that can be exposed as an Artix service endpoint that can be accessed by any of the transports and protocols supported by Artix is to generate the .NET metadata that describes the target service interface. Artix Connect includes a GUI, the Add Artix Reference wizard, which enables you to generate .NET metadata and server starting point code from a WSDL file from within the Microsoft Visual Studio .NET 2003 development environment.

> **Note:** This guide assumes that the WSDL file already exists and that you have been provided with it as a starting point.
>
> For more information on using Artix to develop WSDL files, see the Using Artix Designer guide.

Demonstration code

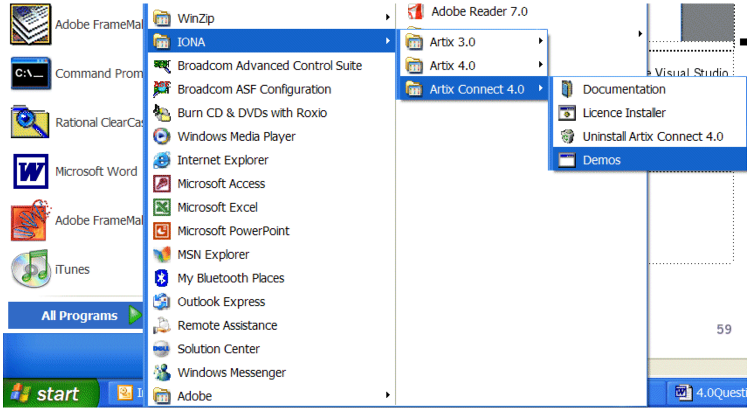
The `Hello World` demo is used as an example application. It is located in:

```
InstallDir\artix\Version\demos\dotnet\hello_world
```

Implementation steps

1. From the Windows **Start** menu, select the Artix Connect 4.0 **Demos**, as shown in Figure 33.

Figure 33: *Selecting Artix Connect Demos*

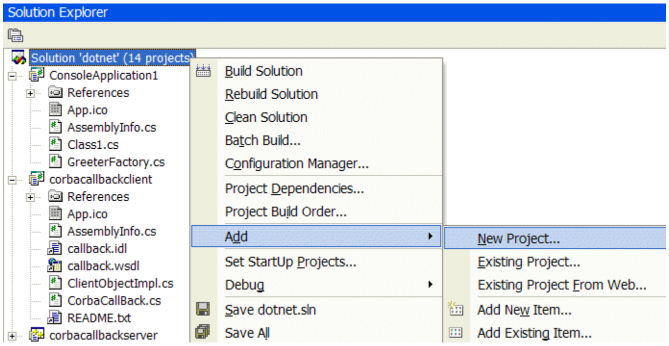


This opens the Artix Connect demos solution within the Visual Studio environment.

Note: You do not have to add your new project to the Artix Connect demo solution. Doing so, however, provides you with a client that you can use to test the server implementation that you are developing in this chapter.

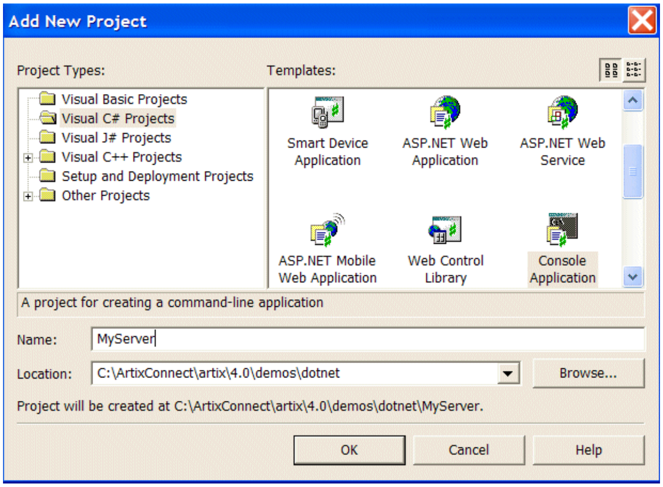
2. Right-click on the dotnet solution file and select **Add | New Project** to add a new project, as shown in Figure 34:

Figure 34: *Adding a New Project to the Demo Solution*



3. The **New Project** dialog box appears as shown in Figure 35. Select the project type that you want to create—in this example, a Visual C# project using the Console Application template:

Figure 35: *Selecting a Project Type*



A project for creating a command-line application

Name: MyServer

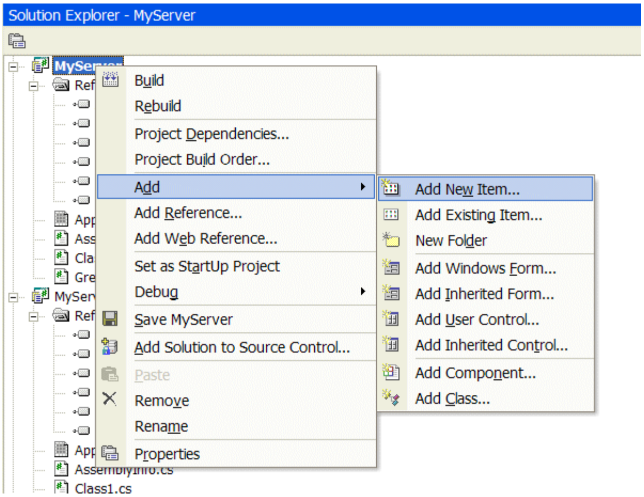
Location: C:\ArtixConnect\artix\4.0\demos\dotnet

Project will be created at C:\ArtixConnect\artix\4.0\demos\dotnet\MyServer.

Note: The Artix Connect GUI supports C# only. For projects that 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 45 for more detail.

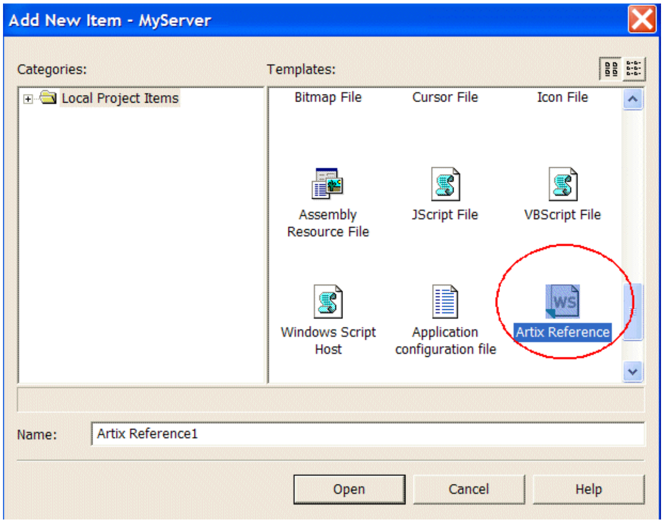
4. Enter a name for your project and select the Artix Connect demo directory as the location for storing your project (as shown in Figure 35).
5. Click **OK**. The Visual Studio .NET 2003 Development Environment creates a new C# project.
6. Next you need to add an Artix reference to your project. To do this, right-click on your project and select **Add | Add New Item**, as shown in Figure 36:

Figure 36: *Launching the Add New Item Dialog Box*



7. The **Add New Item** dialog box appears as shown in Figure 37. Select the **Artix Reference** options and click **Open**:

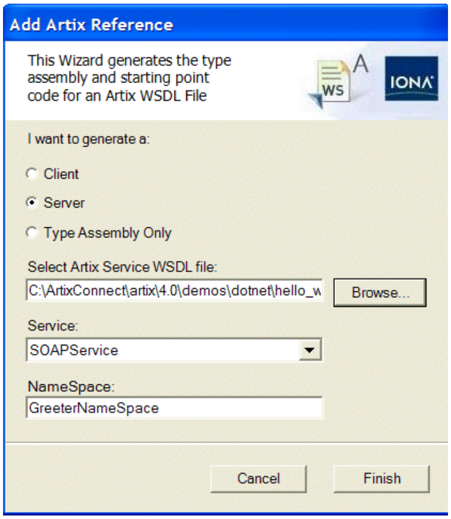
Figure 37: *Launching the Add Artix Reference Wizard*



Name: Artix Reference1

8. The **Add Artix Reference** wizard appears as shown in Example 38. You need to:
   i. Select the **Server** radio button.
   ii. Browse for the WSDL file that describes the service that you want to implement. In this example, select the `hello_world.wsdl` file, located in
   `*InstallDir*\Artix\*Version*\demos\dotnet\hello_world\etc`

Figure 38: *Generating .NET Metadata and Server Implementation Code*



Add Artix Reference

This Wizard generates the type assembly and starting point code for an Artix WSDL File

I want to generate a:

○ Client
● Server
○ Type Assembly Only

Select Artix Service WSDL file: C:\ArtixConnect\artix\4.0\demos\dotnet\hello_w

Service: SOAPService

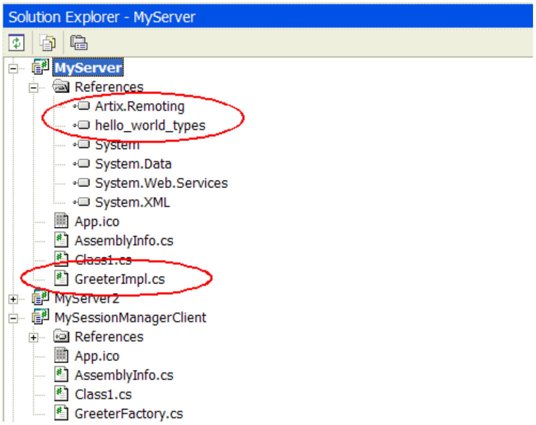
NameSpace: GreeterNameSpace

The **Add Artix Reference** wizard fills in the **Service** and **NameSpace** fields with values taken from the WSDL file. If the WSDL file contains more than one service, the wizard selects the first service. Alternatively, you can select the service that you want to implement from the drop-down list.

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

The **Add Artix Reference** wizard adds three required items to the server project, as shown in Figure 39):

Figure 39: *Required Files Added to Project*



It adds the following references:

- The `Artix.Remoting` assembly, which is required at runtime by all Artix Connect clients and servers.
- The *`WSDLFileName_types`*`.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 `hello_world_types.dll`.

And the following file:

- Server implementation class—in this case, `GreeterImpl.cs`. This is where you add your server implementation code.

Writing a C# Artix Server

Overview

The next task in implementing an Artix server in .NET is to write the C# server. As shown in the previous section, Generating .NET Metadata, the Add Artix Reference wizard generates an empty server implementation from the supplied WSDL file.

Server implementation code

In this example, the file is called `GreeterImpl.cs` and is shown in Example 8. `GreeterImpl.cs` is the implementation class of the `Greeter` porttype defined in `hello_world.wsdl` file, which you selected when running the Add Artix Reference wizard in the previous section. You simply add application logic code to the server.

Example 8: *Greeter Implementation Class—GreeterImpl.cs*

```
using System;

public class MyGreeterImpl : ArtixObject,
    GreeterNameSpace.Greeter
{
     public MyGreeterImpl()
     {
     }

1    public System.String sayHi()
     {
          // TODO: Add the implementation of sayHi
     }

2    public System.String greetMe(System.String me)
     {
          // TODO: Add the implementation of greetMe
     }
}
// Use this class to create an instance of the SOAPService
    service as defined in hello_world.wsdl
// The GreeterImpl class provides the implementation for the
    porttype corresponding to this service:
[System.Web.Services.WebService(Name="SOAPService",
    Namespace="http://www.iona.com/hello_world_soap_http",
    Description="hello_world.wsdl")]
```

Example 8: *Greeter Implementation Class—GreeterImpl.cs*

```
public class SOAPService : MyGreeterImpl
{
}
```

The code shown in Example 8 can be explained as follows:

1. The `sayHi()` method is generated and left empty. You must add the business logic. For example, add the following business logic code to complete the implementation of the `sayHi()` method:

   ```
   Console.WriteLine("sayHi invoked");
   return "Hello from Artix";
   ```

2. The `greetMe()` method is generated and left empty. You must add the business logic. For example, add the following business logic code to complete the implementation of the `greetMe()` method:

   ```
   Console.WriteLine("greetMe invoked from: " + me);
   return String.Concat("Hello " + me);
   ```

Note: The skeleton code generation is driven by services. Therefore, if you want to generate server implementation code, the WSDL file that you reference must include at least one service definition.

Note: The dotnet demos solution already contains a `GreeterImpl.cs`. For the purposes of this example, therefore, change any occurrence of `GreeterImpl` in the your `GreeterImpl.cs` file to `MyGreeterImpl`.

Mainline code

Add the highlighted code shown in Example 9 to the mainline, `Class1.cs`.

Example 9: *Registering the Artix Channel and your Server Implementation*

```
using System;
using IONA.Remoting;
using System.Runtime.Remoting.Channels;

namespace MyServer;

   {
```

Example 9: *Registering the Artix Channel and your Server Implementation*

```
    class Class1
    {
    [STAThread]
    static void Main(string[] args)
       {
1      ArtixServerChannel artixServerChannel = new
       ArtixServerChannel();
       ChannelServices.RegisterChannel(artixServerChannel);

2      MyGreeterImpl greeterObj = new MyGreeterImpl();
       artixServerChannel.RegisterArtixService(greeterObj);

       Console.WriteLine("Press Enter to exit.");
       Console.ReadLine();
       }
    }
 }
```

The code shown in Example 9 can be explained as follows:

1. Creates and registers the Artix channel with remoting.
2. Creates and registers the server implementation object with the channel.

Building and Running your Server

Overview

This section describes how to build the server that you wrote in the previous section, Writing a C# Artix Server.

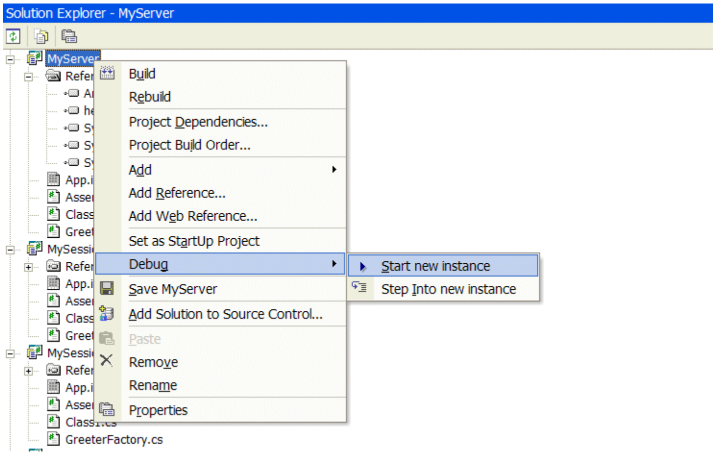
Building your server

To build your server, right-click on your project and select **Build**.

Running your server

To run the server, right-click on your project and select **Debug|Start new instance**, as shown in Example 40:

Figure 40: *Running your Server*



The server starts in a new DOS command window and prints `Server Ready` to the screen.

CHAPTER 7

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:

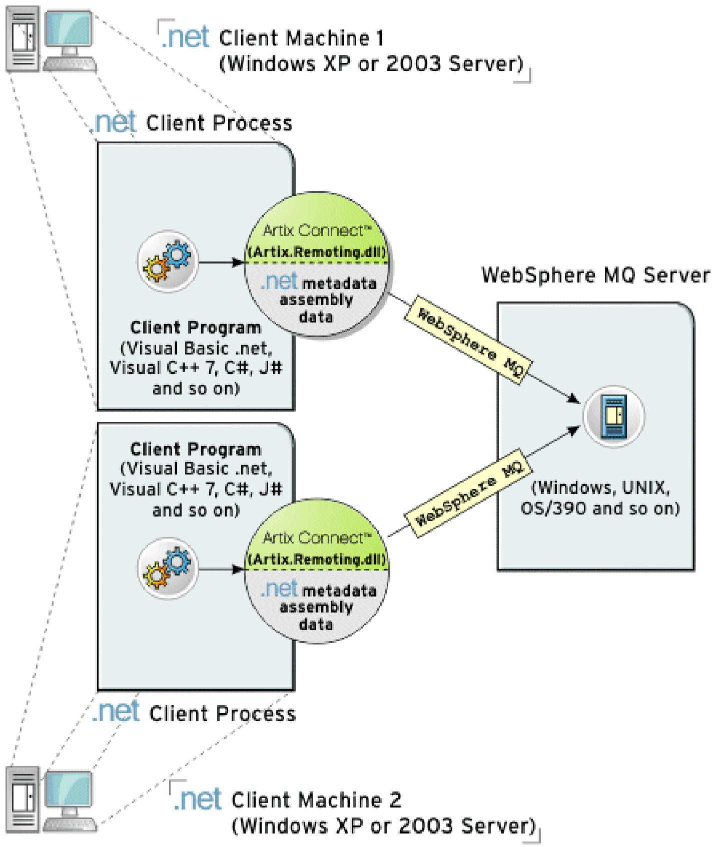
Deployment Model	page 102
Deployment Steps	page 104

Deployment Model

Overview

Figure 41 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 41: *Typical Deployment Scenario*


.net Client Machine 1
(Windows XP or 2003 Server)
.net Client Process
Artix Connect™
(Artix.Remoting.dll)
.net metadata assembly data
Client Program
(Visual Basic .net, Visual C++ 7, C#, J# and so on)
WebSphere MQ Server
WebSphere MQ
WebSphere MQ
(Windows, UNIX, OS/390 and so on)
Client Program
(Visual Basic .net, Visual C++ 7, C#, J# and so on)
Artix Connect™
(Artix.Remoting.dll)
.net metadata assembly data
.net Client Process
.net Client Machine 2
(Windows XP or 2003 Server)

Explanation

The deployment scenario overview in Figure 41 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 file.

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_connect_env.bat` script or set them manually later. See "Configuration" on page 197 for more details.
3. Copy the client executable and the .NET metadata DLL to the deployment host.
4. Copy the WSDL file for the service to which you want to connect.

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

Part III

Using Advanced Features

In this part

This part contains the following chapters:

Client Callbacks	page 107
Using the Artix Locator	page 121
Using the Artix Session Manager	page 135

CHAPTER 8

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	page 108
Implementing Callbacks	page 109

Introduction to Callbacks

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	page 110
Callback WSDL File	page 112
Implementing the Client in C#	page 116
Implementing the Server	page 119

Callback Demonstration

Overview

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

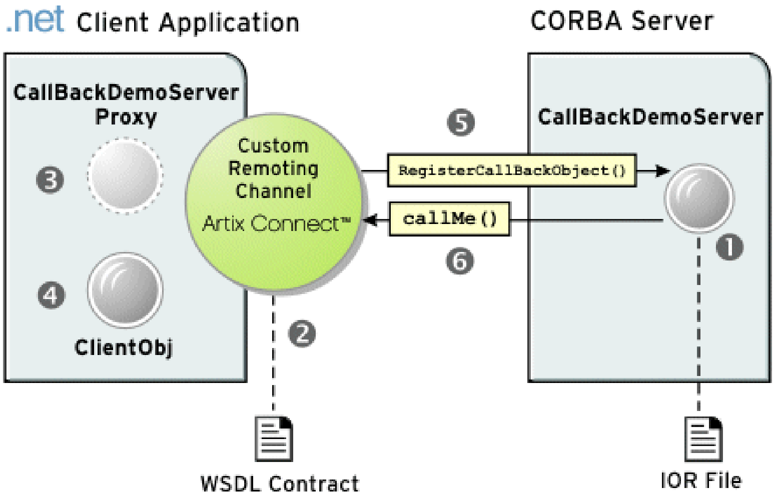
```
InstallDir\artix\Version\demos\dotnet\corba_callback
```

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

Graphical view

Example 42 illustrates how the callback proceeds:

Figure 42: *Callback in Progress*



Example 42 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 file. The WSDL file 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 File

Overview

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

In this subsection

This subsection describes the WSDL file 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 `idltowsdl` command-line utility.

WSDL file

Example 10 shows the WSDL file, `callback.wsdl`, used in the `CORBA Callback` demonstration. It is located in:

```
InstallDir\artix\Version\demos\dotnet\corba_callback\etc
```

Example 10: *Example Callback WSDL file*

```
<?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"
     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"/>
```

Example 10: *Example Callback WSDL file*

```
          </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>

  <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>
```

Example 10: *Example Callback WSDL file*

```
  </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 file shown “Callback WSDL File” on page 112. The client is an implementation of the `ClientObject` port type.

Main client code

Example 11 shows code contained in the `CorbaCallback.cs` file. It contains the C# mainline code that invokes on the server:

Example 11: *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 11 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 file.
   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 12 shows code contained in the `ClientObjectImpl.cs` file. It implements the .NET object that receives the server callback:

Example 12: *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 12: *ClientObjectImpl.cs*

```
    {
        called = false;
    }

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

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

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 file. This is normally the case for callbacks. If, however, the client port type is defined in a different WSDL file from the server port type, you must add a `Description` property that specifies the client WSDL file; 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 file.
3. It is possible to add operations and properties to the client that are not defined in the WSDL file. 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 file.

Implementing the Server

Overview

This subsection describes the CORBA server that is used in the CORBA Callback demonstration. The steps used to implement it are:

- 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 9

Using the Artix Locator

The Artix locator isolates clients from changes in a server's contact information. It is a central repository for storing references to Artix endpoints. If you set up your Artix servers to register their endpoints with the locator, you can code your clients to open server connections by retrieving endpoint references from the locator.

Overview

This chapter contains the following sections:

Overview of the Locator	page 122
Developing a .NET Client that Uses the Locator	page 124
Building and Running your Client	page 130

Overview of the Locator

Overview

The Artix locator isolates clients from changes in a server's contact information. Instead of the client having to know the server location in advance, the locator provides the client with a reference to the server.

The Artix locator is a central repository for storing references to artix endpoints. An Artix *endpoint* is a combination of the network address of a service with the protocol and data format necessary to use that service. It is defined using the WSDL `port` element. You can configure an Artix service to automatically register with the locator to publish its available endpoints. When the service is being shut down it will unregister its endpoints.

A client looks up a reference in the locator in order to find an endpoint associated with a particular service. After retrieving the reference from the locator, the client can then establish a remote connection to the relevant server by instantiating a client proxy object. This procedure is independent of the type of binding or transport protocol.

Locator features

The Artix locator provides the following features:

- **Location of known services**

  The Artix locator acts a central store of endpoint references and enables clients to establish connections to Artix services without pre-existing knowledge of the service location.

- **Load balancing**

  The Artix locator provides a lightweight mechanism for balancing workloads among a group of servers. When a number of servers with the same service name register with the Artix locator, it automatically creates a list of the references and hands out the references to clients using a round-robin algorithm. This process is invisible to both clients and servers.

- **High availability**

  The Artix locator can be run in a replicated, highly available mode. For example, you could have three Artix locator instances running, each one a replica of the others. In the event of a failed service connection the underlying logic automatically queries the locator for a secondary service instance. The process repeats until all endpoint options are exhausted or a successful invocation is made.

More information

For more information on the Artix locator, see the Artix Locator Guide.

Developing a .NET Client that Uses the Locator

Overview

This section describes what you need to do to enable your .NET client to work with the Artix locator.

Demonstration code

The locator demo is used as an example application. It shows a C# .NET client using the locator to find an Artix Web service. It is located in:

```
InstallDir\artix\Version\demos\dotnet\locator
```

Generating client metadata

To generate the metadata and client helper code needed for a .NET client to to contact an Artix Web service follow the steps in "Developing .NET Clients for Artix Services" on page 51, with the following changes when adding an Artix reference to your project:

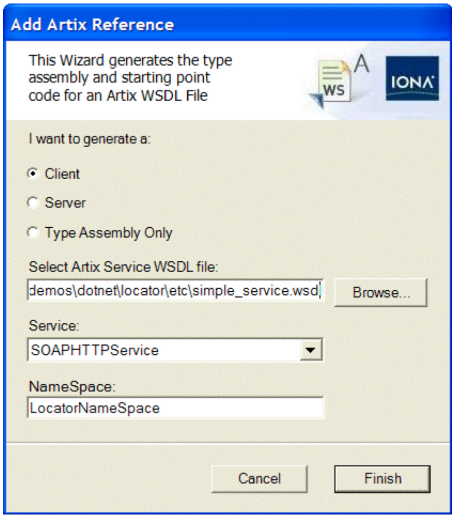
1. Instead of using the hello_world.wsdl file, use the simple_service.wsdl file located in *InstallDir*\artix\Version\demos\dotnet\locator\etc.

   The reason for implementing a client for the simple service defined in the simple_service.wsdl file is that the locator demo includes a server that implements the simple service defined in the simple_service.wsdl and registers itself with the locator. This provides you with a locator-enabled server that you can use when running the client that you develop in this section.

2. Use the LocatorNameSpace namespace, as shown in Figure 43 on page 125. If an application uses two services that are defined in separate WSDL files and these services interact with each other, you must ensure that you use the same namespace for the type assemblies. In this example, the locator and the simple service described in the simple_service.wsdl file must interact. The Locator.dll, the type assembly for the locator, uses the

`LocatorNameSpace` namespace. Therefore, when generating the type assembly for the `simple_service.wsdl`, you must use the same namespace.

Figure 43: *Generating Metadata for Simple Service*



This Wizard generates the type assembly and starting point code for an Artix WSDL File

I want to generate a:

● Client

○ Server

○ Type Assembly Only

Select Artix Service WSDL file: demos\dotnet\locator\etc\simple_service.wsd|

Service: SOAPHTTPService

NameSpace: LocatorNameSpace

Enabling your client to use the locator

To enable your .NET client to work with the Artix Locator you must:

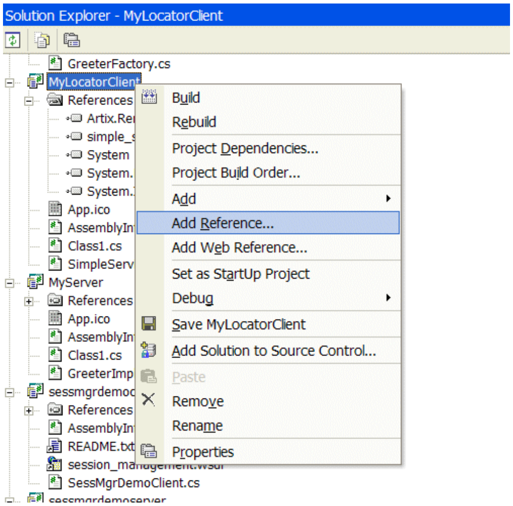
- Add the Locator.dll assembly to your project
- Add locator-specific code to your client

Add the Locator.dll assembly to your project

To add the `Locator.dll` to your project:

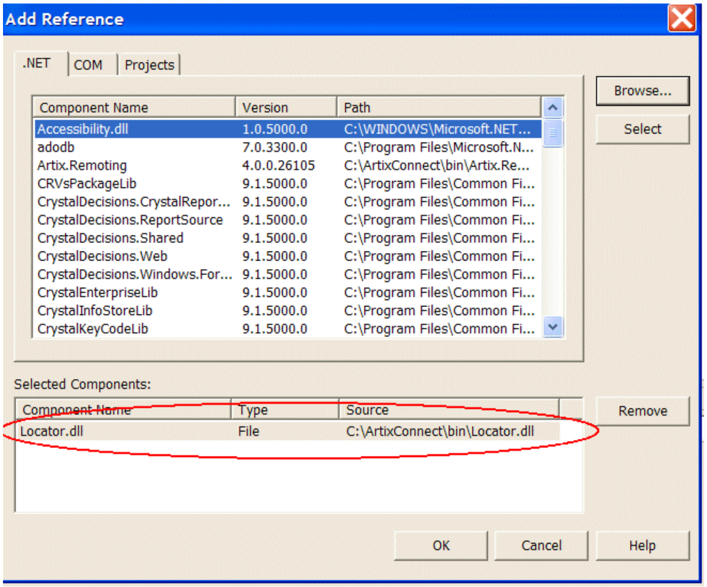
1. Right-click on your project and select **Add Reference**, as shown in Figure 44:

Figure 44: *Adding a Reference to your Project*



2. The Add Reference dialog box appears as shown in Figure 45 on page 127. Browse for the `Locator.dll`, which is located in the following *`InstallDir`*`\bin` directory, and click OK.

Figure 45: *Adding the Locator.dll to your Project*



Component Name	Version	Path
Accessibility.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET...
adodb	7.0.3300.0	C:\Program Files\Microsoft.N...
Artix.Remoting	4.0.0.26105	C:\ArtixConnect\bin\Artix.Re...
CRVsPackageLib	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.CrystalRepor...	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.ReportSource	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.Shared	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.Web	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.Windows.For...	9.1.5000.0	C:\Program Files\Common Fi...
CrystalEnterpriseLib	9.1.5000.0	C:\Program Files\Common Fi...
CrystalInfoStoreLib	9.1.5000.0	C:\Program Files\Common Fi...
CrystalKeyCodeLib	9.1.5000.0	C:\Program Files\Common Fi...

Selected Components:

Component Name	Type	Source
Locator.dll	File	C:\ArtixConnect\bin\Locator.dll

Add locator-specific code to your client

A client must able to look up a reference in the locator in order to find an endpoint associated with a particular service. After retrieving the reference from the locator, the client can then establish a remote connection to the relevant server by instantiating a client proxy object. This procedure is independent of the type of binding or transport protocol. Add the code shown in Example 13 to your client to enable it to use the locator to gain access to the simple service.

Example 13: *Client Using the Artix Locator*

```
using System;
using System.Runtime.Remoting.Channels;
using System.Diagnostics;
using System.Xml;
using IONA.Remoting;
using LocatorNameSpace;

namespace MyLocatorClient
{
/// <summary>
/// Summary description for Class1.
/// </summary>
class Class1

  {
    [STAThread]
     static void Main(string[] args)
    {
       try
       {
1         ChannelServices.RegisterChannel(new
          ArtixClientChannel("demo.locator.client"));
2         LocatorService artixLocator = (LocatorService)
          Activator.GetObject(typeof(LocatorService),
          "artixref:../../locator-activated.wsdl
          http://ws.iona.com/locator LocatorService
          LocatorServicePort");

          XmlQualifiedName serviceQName = new
          XmlQualifiedName("SOAPHTTPService",
          "http://www.iona.com/FixedBinding");
```

Example 13: *Client Using the Artix Locator*

```
3           SimpleServicePortType simpleService =
           (SimpleServicePortType)
            artixLocator.lookup_endpoint(serviceQName);
4           String myGreeting = "Greeting : " + serviceQName.Name;
            String result = simpleService.say_hello(myGreeting);
            Console.WriteLine();
            Console.WriteLine("say_hello method returned: " +
            result);
       }
       catch (Exception)
       {
          Console.Error.WriteLine("Exception: " + ex.ToString());
       }
         Console.WriteLine("Press Enter to exit.");
         Console.ReadLine();
     }
   }
}
```

The code shown in Example 13 can be explained as follows:

1. Registers the Artix Connect channel with the .NET 1.1 remoting framework.
2. Creates a locator proxy.
3. Invokes on `lookup_endpoint()` using the locator proxy. This returns a proxy to the service.
4. Invokes on a service operation using this proxy.

After the client initializes the proxy with the reference obtained from the locator, invocations are sent directly to the target server process—invocations are not redirected by the locator process.

Building and Running your Client

Overview

This section walks you through the steps to building and running your client against a service that has registered its endpoints with the locator.

Servers require specific configuration settings to enable them to register with the locator. Server-side requirements are beyond the scope of this book. For information on how to configure a server to register its endpoints with the locator, see the Artix Locator Guide.

Getting a reference to the locator

In order for your client to use the locator, it must first get a reference to the locator. By default, the locator is configured to deploy on a dynamic port. In the default locator WSDL file, the addressing information is as shown in Example 14:

Example 14: *Locator Dynamic Port in Default locator.wsdl File*

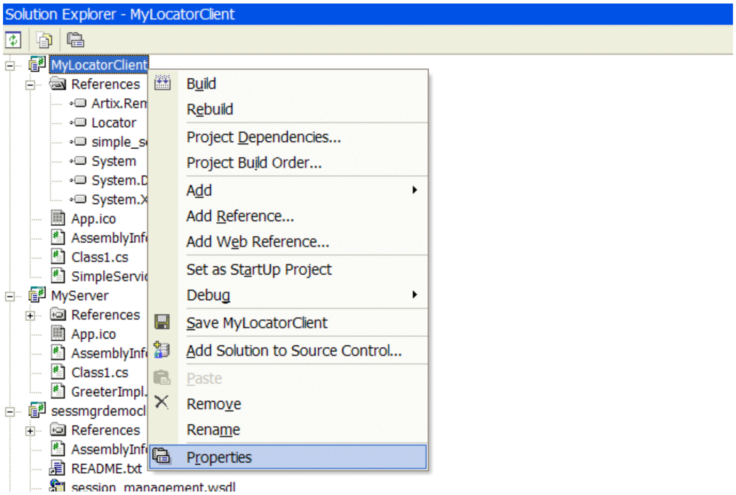
```
<service name="LocatorService">
       <port binding="ls:LocatorServiceBinding"
             name="LocatorServicePort">
           <soap:address
          location="http://localhost:0/services/LocatorService"/>
       </port>
</service>
```

The `localhost:0` port means that when the locator is activated, the operating system assigns a port dynamically on startup. There are, however, several ways to deploy the locator. In the locator demo, the locator is deployed using the Artix container. You can use the container admin console to publish a new `locator-activated.wsdl` file that contains the port details of the deployed locator. This is the WSDL file that the client must use.

To use the Artix container admin console to create a `locator-activated.wsdl` file:

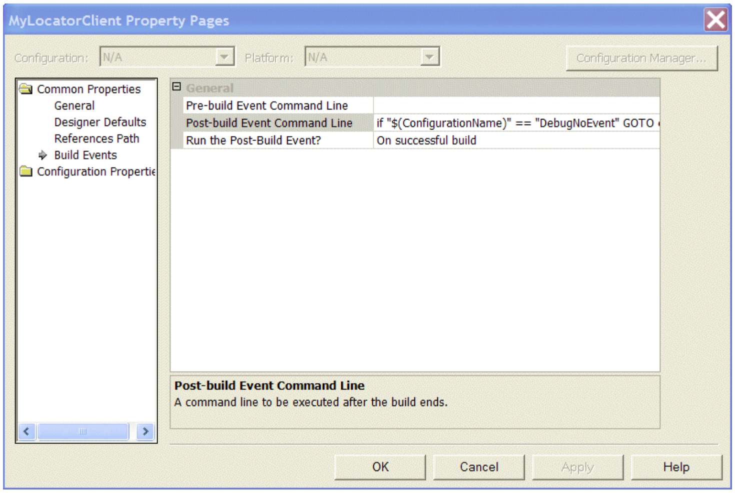
1. Create a post-build event by right-clicking on your client project and selecting **Properties**, as shown in Figure 46:

Figure 46: *Creating a Post-build Event*



The Properties dialog box appears as shown in Figure 47:

Figure 47: *Project Properties Dialog*



General	
Pre-build Event Command Line	
Post-build Event Command Line	if "$(ConfigurationName)" == "DebugNoEvent" GOTO
Run the Post-Build Event?	On successful build

Post-build Event Command Line
A command line to be executed after the build ends.

2. Expand the Common Properties node and select Build events.
3. Select the Post-build Event Command Line and enter the command shown in Example 15, substituting *InstallDir* with details of the directory into which you have Artix Connect installed:

Example 15: *Post-build Event to Get Locator Port Details*

```
if "$(ConfigurationName)" == "DebugNoEvent" GOTO end
it_container_admin -container
    InstallDir\artix\4.0\demos\dotnet\locator\etc\
     ContainerService.url -publishwsdl -service
    {http://ws.iona.com/locator}LocatorService -file
    InstallDir\artix\4.0\demos\dotnet\MyLocatorClient\
    locator-activated.wsdl
:end
echo Finished Post Build Event
```

4. Click **OK** and **Apply** to save your changes.

The post-build event creates the `locator-activated.wsdl` file and places it in your locator client directory.

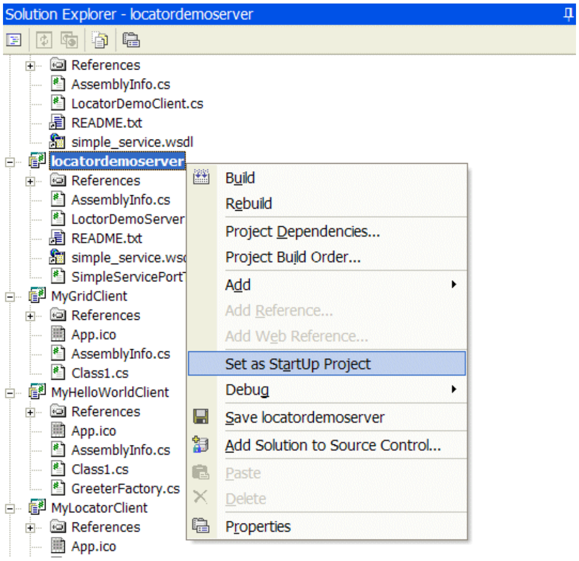
For more information on the Artix container, and the container admin console and its commands, see the *Deploying Services in an Artix Container* chapter in the Configuring and Deploying Artix Solutions guide.

Running the server

To run the server:

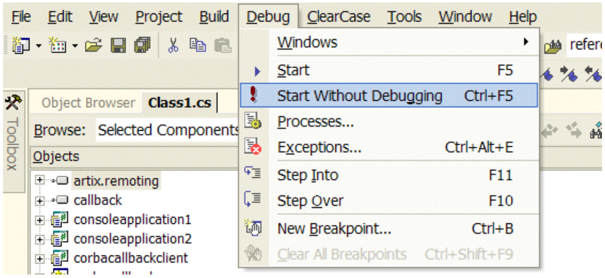
1. Right-click on the `locatordemoserver` project and select **Set as StartUp Project**, as shown in Figure 48:

Figure 48: *Starting the Simple Service Server—Set as StartUp Project*



2. From the **Debug** menu, select Start without debugging, as shown in Figure 49:

Figure 49: *Starting the Simple Service Server—Start Without Debugging*



The Artix container starts in a new console window and starts both the locator and the simple service server. When the server starts up it registers its endpoints with the running locator.

Building your client

To build your client, right-click on your client project and select **Build**.

Running your client

To run your client, right-click on your client project and select **Debug|Start new instance**. The client starts in a new Windows console, invokes on the simple service and prints to following to the screen:

```
say_hello method returned: Greeting : SOAPHTTPService
Press Enter to exit.
```

CHAPTER 10

Using the Artix Session Manager

The Artix session manager enables Web service clients to hold conversations with stateful servers. Client requests are identified as being part of a session and the server can hold state information relating to the client by identifying the requests as part of that client's session. In addition, the session manager controls the number of concurrent clients that can access a Web service and the amount of time allocated to each session.

Overview

This chapter contains the following sections:

Overview of the Session Manager	page 136
Developing a .NET Client that Uses the Session Manager	page 137
Building and Running your Client	page 143

Overview of the Session Manager

Overview

The Artix session manager is implemented as a group of plug-ins that work together to manage the number of concurrent clients allowed to connect to a group of services. An Artix plug-in is a code library that can be loaded into an Artix application at runtime. The session manager plug-ins work together to control how long a client has access to a service before it has to request an extension. In addition, the session manager notifies all registered services of session state changes, including when sessions begin and when they end. This section gives an overview of the session manager's use cases and describes the plug-ins and how they work together in a deployed system.

Use cases

The Artix session manager supports the following use cases:

Limiting the amount of time a client is connected to a service

You can use the Artix session manager to control the amount of time a client has access to a service. This is useful when you do not want clients to have unrestricted access to a service. For example, you might want to limit the amount of time available to complete a request form to five minutes. Clients can request session extensions.

Limiting the number of concurrent client connections to a service

You can specify how many concurrent connections are permitted to a service. For example, if your services are running on old hardware you could ensure higher performance by limiting the number of connections to a small number.

Stateful services

You can write services that store state data across multiple invocations. This is possible because clients of session managed services include identity details with each invocation. Using the session manager's callback mechanism, you can destroy any state information for a client once the client's session expires.

More information

For more information on the Artix session manager, see the Artix Session Manager Guide.

Developing a .NET Client that Uses the Session Manager

Overview

If you want your client to make requests on a session managed service, you must implement the client to interact with the Artix session manager and to pass session headers to the session managed services. This section describes the implementation steps.

Demonstration code

The `Session Management` demo is used as an example application. It shows a C# .NET client using a session managed server. It is located in:

```
InstallDir\artix\Version\demos\dotnet\session_management
```

Generating client metadata

To generate the metadata and client helper code needed for a .NET client to to contact an Artix Web service follow the steps in "Developing .NET Clients for Artix Services" on page 51, with the following changes when adding an Artix reference to your project:

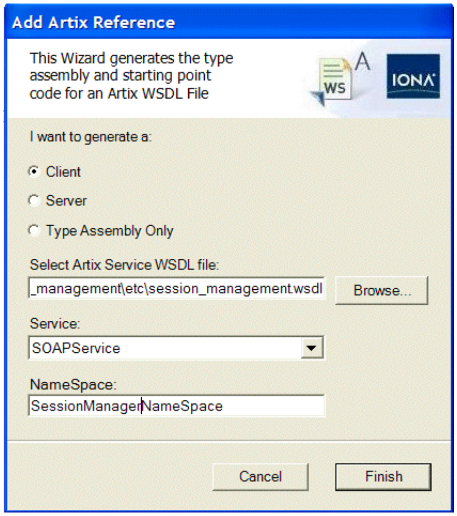
1. Instead of using the `hello_world.wsdl` file, use the `session_management.wsdl` file located in *InstallDir*\artix\Version\demos\dotnet\session_management\etc.

   The reason for implementing a client for the service defined in the `session_management.wsdl` file is that the `session management` demo includes a server that implements the SOAP service defined in the `session_management.wsdl` and registers itself with the session manager. This provides you with a session-managed server that you can use when running the client that you develop in this section.

2. Use the `SessionManagerNameSpace` namespace, as shown in Figure 50 on page 138. If an application uses two services that are defined in separate WSDL files and these services interact with each other, you must ensure that you use the same namespace for the type assemblies. In this example, the session manager and the service described in the `session_management.wsdl` file must interact. The `SessionManager.dll`, the type assembly for the session manager, uses

the `SessionManagerNameSpace` namespace. Therefore, when generating the type assembly for the `session_management.wsdl`, you must use the same namespace.

Figure 50: *Generating Metadata for Session Management Service*



Add Artix Reference

This Wizard generates the type assembly and starting point code for an Artix WSDL File

I want to generate a:

● Client

○ Server

○ Type Assembly Only

Select Artix Service WSDL file: _management\etc\session_management.wsdl

Service: SOAPService

NameSpace: SessionManagerNameSpace

Enabling your client to use the session manager

To enable your .NET client to work with the Artix session manager you must:

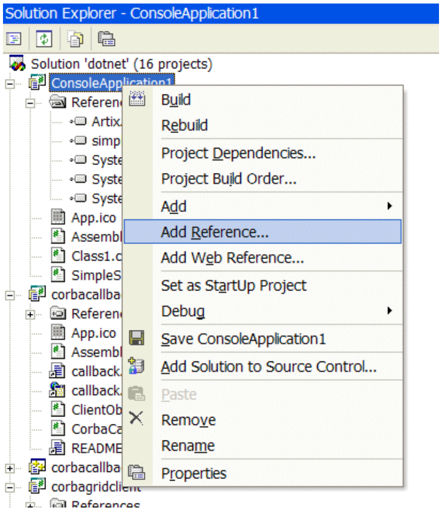
- Add the SessionManager.dll assembly to your project
- Add session manager-specific code to your client

Add the SessionManager.dll assembly to your project

To add the `SessionManager.dll` assembly to your project:

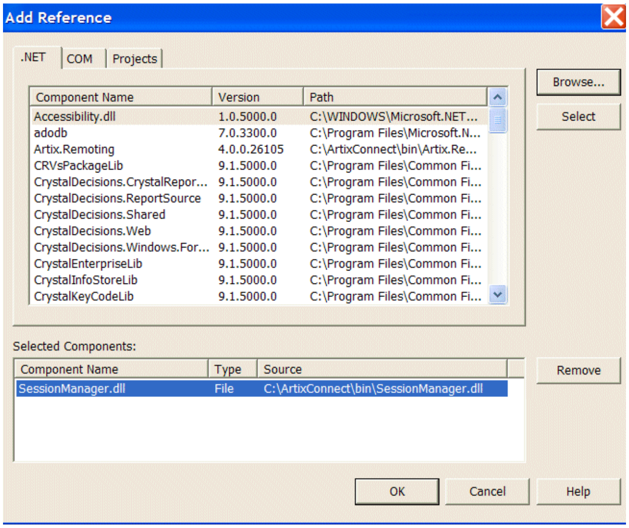
1. Right-click on your project and select **Add Reference**, as shown in Figure 51:

Figure 51: *Adding a Reference to your Project*



The Add Reference dialog box appears as shown in Figure 52. Browse for the `SessionManager.dll`, which is located in the *InstallDir*`\bin` directory, and click OK.

Figure 52: *Adding the SessionManager.dll to your Project*



Component Name	Version	Path
Accessibility.dll	1.0.5000.0	C:\WINDOWS\Microsoft.NET...
adodb	7.0.3300.0	C:\Program Files\Microsoft.N...
Artix.Remoting	4.0.0.26105	C:\ArtixConnect\bin\Artix.Re...
CRVsPackageLib	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.CrystalRepor...	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.ReportSource	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.Shared	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.Web	9.1.5000.0	C:\Program Files\Common Fi...
CrystalDecisions.Windows.For...	9.1.5000.0	C:\Program Files\Common Fi...
CrystalEnterpriseLib	9.1.5000.0	C:\Program Files\Common Fi...
CrystalInfoStoreLib	9.1.5000.0	C:\Program Files\Common Fi...
CrystalKeyCodeLib	9.1.5000.0	C:\Program Files\Common Fi...

Selected Components:

Component Name	Type	Source
SessionManager.dll	File	C:\ArtixConnect\bin\SessionManager.dll

Add session manager-specific code to your client

Add the code shown in Example 16 to your client:

Example 16:

```
//C#
using System;
using System.Runtime.Remoting.Channels;
using System.Diagnostics;
using System.Threading;
using IONA.Remoting;
using SessionManagerNameSpace;
...
```

1
```
ChannelServices.RegisterChannel(new
   ArtixClientChannel("demo.session_management.client"));
```

Example 16:

```
2   SessionManager artixSessionManager = (SessionManager)
       Activator.GetObject(typeof(SessionManager),
       "artixref:../../session-manager-activated.wsdl
       http://ws.iona.com/sessionmanager SessionManagerService
       SessionManagerPort");

3   SessionInfo sessionInfo =
       artixSessionManager.begin_session("SM_Demo",20);

4   SessionId groupSession = sessionInfo.session_id;

5   EndpointList endpointList = null;
    endpointList =
       artixSessionManager.get_all_endpoints(groupSession);

6   Greeter greeterObj = (Greeter)endpointList.endpoint[0];

7   greeterObj.__SetSessionId(groupSession.name,
       groupSession.endpoint_group);

8   String response;
    response = greeterObj.sayHi();
    Console.WriteLine("sayHi response: " + response);

    Console.WriteLine("Invoking greetMe method");
    response = greeterObj.greetMe(".NET Connector");
    Console.WriteLine("greetMe response: " + response);

9   artixSessionManager.end_session(groupSession);
    }
```

The code in Figure 16 can be explained as follows:

1. Registers the Artix Connect channel with the .NET remoting channel.
2. Creates a proxy for the Artix session manager.
3. Begins a session for the desired service's group using the session manager proxy.
4. Retrieves the session ID from the response.
5. Obtains the list of available endpoints.
6. Instantiates a proxy to the target Web service using one of the endpoints in the group.

7. Builds a session header that contains the session ID to be passed to the service.
8. Invokes on the target service using the proxy.
9. Ends the session using the session manager proxy when finished with the services.

More information

For more detailed information on the writing a session managed client, see the Artix Session Manager Guide.

Building and Running your Client

Overview

This section walks you through the steps to building and running your client against a session-managed service.

Servers require specific configuration settings to enable them to register with the session manager. Server-side requirements are beyond the scope of this book. For information on how to configure a server to register its endpoints with the session manager, see the Artix Session Manager Guide.

Getting a reference to the session manager

In order for your client to use the session manager, it must first get a reference to the session manager. By default, the session manager is configured to deploy on a dynamic port. In the default session manager WSDL file, the addressing information is as shown in Example 17:

Example 17: *Session Manager on Dynamic Port in Default session_manager.wsdl File*

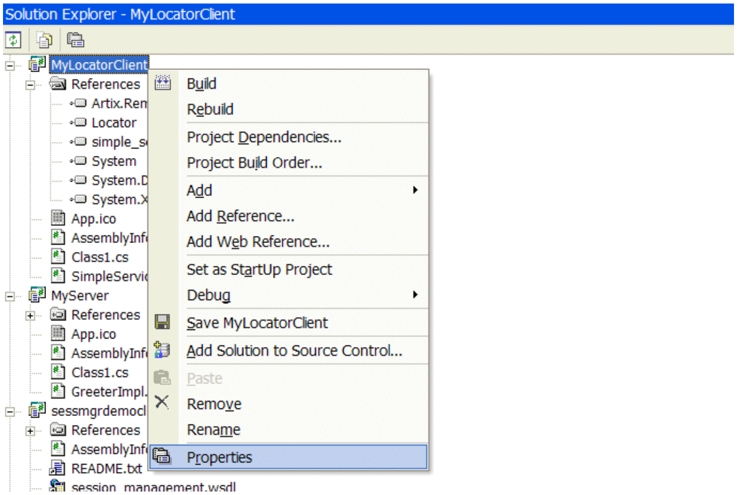
```
<service name="SessionManagerService">
   <port binding="sm:SessionManagerBinding"
     name="SessionManagerPort">
      <soap:address
       location="http://localhost:0/services/sessionManagement/
       sessionManagerService"/>
    </port>
</service>
```

The `localhost:0` port means that when the session manager is activated, the operating system assigns a port dynamically on startup. There are, however, several ways to deploy the session manager. In the session manager demo, the session manager is deployed using the Artix container. You can use the container admin console to publish a new `session-manager-activated.wsdl` file that contains the port details of the deployed session manager. This is the WSDL file that the client must use.

To use the Artix container admin console to create a `session-manager-activated.wsdl` file:

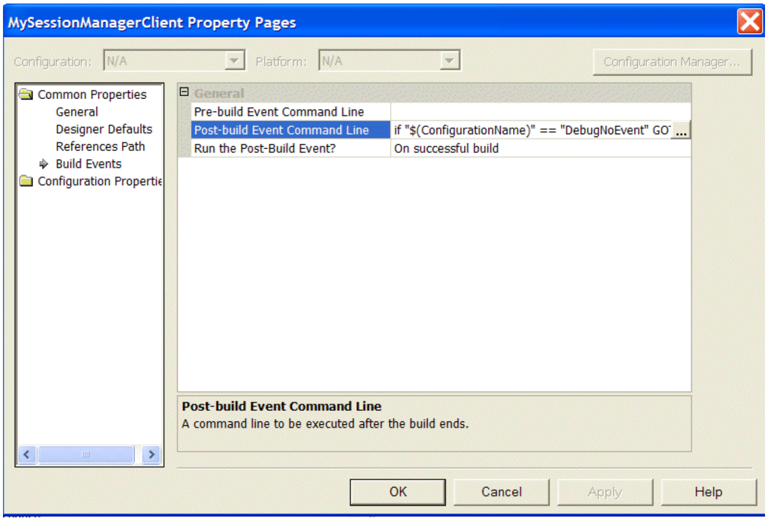
1. Create a post-build event by right-clicking on your client project and selecting **Properties**, as shown in Figure 53:

Figure 53: *Creating a Post-build Event*



The Properties dialog box appears as shown in Figure 54:

Figure 54: *Project Properties Dialog*



Pre-build Event Command Line

Post-build Event Command Line: if "$(ConfigurationName)" == "DebugNoEvent" GO

Run the Post-Build Event?: On successful build

Post-build Event Command Line
A command line to be executed after the build ends.

2. Expand the Common Properties node and select Build events.
3. Select the Post-build Event Command Line and enter the command shown in Example 18, substituting `InstallDir` with details of the directory into which you have Artix Connect installed and `MySessionManagerClient` with the name of your client project:

Example 18: *Post-build Event to Get Session Manager Port Details*

```
if "$(ConfigurationName)" == "DebugNoEvent" GOTO end
it_container_admin -container
   InstallDir\artix\4.0\demos\dotnet\session_management\etc\
   ContainerService.url -publishwsdl -service
   {http://ws.iona.com/sessionmanager}SessionManagerService
   -file
   InstallDir\artix\4.0\demos\dotnet\MySessionManagerClient\
   session-manager-activated.wsdl
:end
echo Finished Post Build Event
```

4. Click **OK** and **Apply** to save your changes.

The post-build event creates the `session-manager-activated.wsdl` file and places it in your client project directory.

For more information on the Artix container, and the container admin console and its commands, see the *Deploying Services in an Artix Container* chapter in the Configuring and Deploying Artix Solutions guide.

Running the server

To run the server:

1. Right-click on the `sessmgrdemoserver` project and select **Set as StartUp Project**
2. From the **Debug** menu, select **Start Without Debugging**

The Artix container starts in a new console window and starts both the session manager and the server. When the server starts up it registers its endpoints with the running session manager.

Building your client

To build your client, right-click on your client project and select **Build**.

Running your client

To run your client, right-click on your client project and select **Debug|Start new instance**. The client starts in a new Windows console, invokes on the server and prints to following to the screen:

```
sayHi response: Hello from Artix
Invoking greetMe method
greetMe response: Hello .NET Connector
Press Enter to exit.
```

Part IV

Reference Material

In this part

This part contains the following chapters:

Introduction to WSDL	page 149
WSDL to .NET Mapping	page 163
Configuration	page 197

Introduction to WSDL

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

In this chapter

This chapter discusses the following topics:

WSDL Basics	page 150
Abstract Data Type Definitions	page 153
Abstract Message Definitions	page 156
Abstract Interface Definitions	page 159
Mapping to the Concrete Details	page 162

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 files 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 19 shows a simple WSDL document. It defines a SOAP over HTTP service access point that returns the date.

Example 19: *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 19: *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 20, 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 20: *personalInfo*

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

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

Example 21 shows one mapping of personalInfo into XSD. This mapping is a direct representation of the data types defined in Example 20. 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 21: *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 22. WIth this mapping, however, you cannot reuse the description of hairColorType.

Example 22: *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 22: *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 20 on page 153 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 23.

Example 23: *personalInfo lookup method*

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

This method signature could be mapped to the WSDL fragment shown in Example 24.

Example 24: *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 2

Table 2: *Part Data Type Attributes*

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

Messages are allowed to reuse part names. For instance, if a method has a parameter, `foo`, 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 25.

Example 25: *Reused part*

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

Example 25: *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 3.

Table 3: *Operation Message Elements*

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

Table 3: *Operation Message Elements*

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

An operation is required to have at least one input or output element. The elements are defined by two attributes listed inTable 4.

Table 4: *Attributes of 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 20 on page 153, you might use an interface similar to the one shown in Example 26.

Example 26: *personalInfo lookup interface*

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

This interface could be mapped to the port type in Example 27.

Example 27: *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 12

WSDL to .NET Mapping

To enable interworking between .NET clients and services described in WSDL files, .NET clients must be presented with metadata that describes the interfaces exposed by the WSDL file. 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. The mappings are based on the WSDL to C# mapping as defined by Microsoft.

In this chapter

This chapter discusses the following topics:

Mapping a WSDL File to CTS	page 164
Simple Types	page 173
Complex Types	page 179
Occurrence Constraints	page 194
SOAP Arrays	page 196

Mapping a WSDL File to CTS

Overview

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

In this section

This section contains the following subsections:

Port Types	page 165
Operations	page 167
Messages	page 168
Document/Literal Wrapped Style	page 170

Port Types

Overview

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

WSDL file example

For example, the WSDL file shown in Example 28 generates a C# interface called `sportsCenterPortType`. which contains one operation, called `update`. (see Example 29)

Example 28: *Segment of Sports Center WSDL File*

```
<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 29 shows how the preceding WSDL file maps to a C# interface defined using the Common Type System:

Example 29: *C# Mapping for Sports Center WSDL File*

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

Operations

Overview

Every operation element contained in a WSDL file 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 file example

Example 30 shows a WSDL file that contains an operation called greetMe:

Example 30: *WSDL File 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 file shown in Example 30 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 file.

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 file example

For example, the WSDL file fragment shown in Example 31 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 31: *Segment of WSDL File*

```
<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 32 shows how the preceding WSDL file maps to a C# interface defined using the Common Type System:

Example 32: *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 file, 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 file, 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 return 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 file 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 file example

Example 33 shows how the WSDL file shown in Example 31 could be expressed in WSDL using the document/literal style:

Example 33: *Segment of Sports Final WSDL File 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 33: *Segment of Sports Final WSDL File 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 34 shows how the preceding WSDL file maps, for example, to a C# interface defined using the Common Type System:

Example 34: *C# Mapping for Sports Final WSDL File 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	page 174
Lists	page 176
Unsupported Simple Types	page 178

Atomic Types

Table of atomic types

Table 5 shows how the XSD schema atomic types map to .NET CTS types:

Table 5: *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 5: *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 file example

Example 35 shows a WSDL definition for a list of strings:

Example 35: *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 file shown in Example 35 maps to a .NET array as shown in Example 36:

Example 36: *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	page 180
Arrays	page 182
Choice Complex Type	page 184
Attributes	page 186
Attributes with In-line Data Descriptions	page 188
Enumerations	page 191
Deriving a Complex Type from a Simple Type	page 192

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 file. 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 file example

Example 37 shows an XSD sequence type with three simple elements:

Example 37: *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 38 shows the result of mapping the SequenceType type (from the preceding Example 37) to C# defined using CTS:

Example 38: *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 Occurrence Constraints.

WSDL file example

Example 39 shows an example of such a sequence:

Example 39: *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 39: *WSDL Definition for Sequence with one Element containing minOccurs and maxOccurs Attributes*

```
</portType>
```

CTS mapping

Artix Connect maps the WSDL file shown in Example 39 to C# as shown in Example 40:

Example 40: *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 file shown in Example 39 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 file example

Example 41 shows an XSD choice type with three elements:

Example 41: *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 42 shows the result of mapping the `ChoiceType` (from the preceding Example 41) to C#:

Example 42: *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 42: *C# Mapping of ChoiceType*

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

Attributes

Overview

Artix Connect maps an attribute to a public field in the generated C# class.

WSDL file example

Example 43 shows a segment of a WSDL file that includes an attribute, called "`varAttrString`":

Example 43: *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 43 maps to C# as shown in Example 44:

Example 44: *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 44: *C# Mapping for Attribute varAttrString*

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

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

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

Attributes with In-line Data Descriptions

Overview

Artix Connect maps an attribute with in-line data to a C# class as shown below. An attribute is mapped to a public field in the generated C# class.

WSDL file example

Example 45 shows two attributes—`Value` and `SDValue`—that have in-ine data descriptions.

Example 45: *WSDL Definition of Attributes with In-line Data Descriptions*

```
<complexType name="complex_attributes">
    <attribute default="int" name="DataType" type="xsd:string"/>
    <attribute name="Number" type="xsd:int"/>
    <attribute name="Value" use="required">
        <simpleType>
            <restriction base="xsd:string">
                <enumeration value="1"/>
                <enumeration value="2"/>
                <enumeration value="3"/>
                <enumeration value="4"/>
            </restriction>
        </simpleType>
    </attribute>
    <attribute name="SDValue">
        <simpleType>
            <restriction base="xsd:string">
                <enumeration value="MemberAcct"/>
                <enumeration value="ClearingProp"/>
                <enumeration value="MemberMember"/>
                <enumeration value="Other"/>
            </restriction>
        </simpleType>
    </attribute>
</complexType>
```

CTS mapping

The WSDL schema definition shown in Example 45 maps to C# classes shown in Example 46, Example 47, and Example 48.

In attributes mapping, if the attribute is a value type, and the `use` attribute is set to `"optional"`, and the `default` and `fixed` are not specified, two fields are created—one to hold the value; and another special field whose type is bool with XmlIgnoreAttribute and the name of which is the field name appended with Specified. Example 46, for instance, shows the attribute Number mapped to public System.Int32 Number and the special field mapped to NumberSpecified. The NumberSpecified special field specifies whether XmlSerializer should write the attribute Number. The NumberSpecified field appears with System.Xml.Serialization.XmlIgnoreAttribute to prevent it from being serialized by XmlSerializer.

Example 46: *complex_attributes.cs*

```
[System.Serializable()]
public class complex_attributes {

    public System.String DataType = "int";

    public System.Int32 Number;

    [System.Xml.Serialization.XmlIgnoreAttribute()]
    public bool NumberSpecified;

    public complex_attributesValue Value;

    public complex_attributesSDValue SDValue;

    [System.Xml.Serialization.XmlIgnoreAttribute()]
    public bool SDValueSpecified;

}
```

Note:

1. If the use attribute is not specified, the default use attribute is "optional". That is why the NumberSpecified special field is generated for the Number attribute.

2. The enum is special case value type. That is why a special field—SDValueSpecified—is generated for the SDValue attribute.

The `Value` attribute has in-line data and is mapped to a `complex_attributesValue` C# class, as shown in Example 47:

Example 47: *complex_attributesValue.cs*

```
[System.Serializable()]
public enum complex_attributesValue {

    [System.Xml.Serialization.XmlEnumAttribute(Name="1")]
    Item1,

    [System.Xml.Serialization.XmlEnumAttribute(Name="2")]
    Item2,

    [System.Xml.Serialization.XmlEnumAttribute(Name="3")]
    Item3,

    [System.Xml.Serialization.XmlEnumAttribute(Name="4")]
    Item4,

}
```

The `SDValue` attribute has in-line data and is mapped to a `complex_attributesSDValue` C# class, as shown in Example 48:

Example 48: *complex_attributesSDValue.cs*

```
[System.Serializable()]
public enum complex_attributesSDValue {

    MemberAcct,

    ClearingProp,

    MemberMember,

    Other,
}
```

Enumerations

Overview

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

WSDL file example

Example 49 shows a WSDL definition for an enumeration, `DecimalEnum`:

Example 49: *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 50

Example 50: *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,
}
```

Deriving a Complex Type from a Simple Type

Overview

Artix Connect supports the derivation of a complex type from a simple type. A simple type has, by definition, neither subelements nor attributes. Therefore, one of the main reasons for deriving a complex type from a simple type is to add attributes to the simple type.

WSDL file example

Example 51 shows a WSDL segment in which two complex types—`SimpleContent1` and `SimpleContent2`—are derived from simple types.

Example 51: *WSDL Definition of Complex Type Derived from Simple Type*

```
<xsd:complexType name="SimpleContent1">
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="attrib1a" type="xsd:int"
   use="optional"/>
            <xsd:attribute name="attrib1b" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
<xsd:complexType name="SimpleContent2">
    <xsd:simpleContent>
        <xsd:extension base="s:SimpleContent1">
            <xsd:attribute name="attrib2a" type="xsd:string"/>
            <xsd:attribute name="attrib2b" type="xsd:string"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

CTS mapping

The `SimpleContent1` and `SimpleContent2` types are mapped to C# classes, as shown in Example 52 and Example 53:

Example 52: *SimpleContent1.cs*

```
[System.Serializable()]
public class SimpleContent1 {

    public System.Int32 attrib1a;
```

Example 52: *SimpleContent1.cs*

```
    [System.Xml.Serialization.XmlIgnoreAttribute()]
    public bool attrib1aSpecified;

    public System.String attrib1b;

    private System.String _Value;

    public virtual System.String Value {
        get {
            return this._Value;
        }
        set {
            this._Value = value;
        }
    }
}
```

The complex type `SimpleContent1` is mapped to the C# class `SimpleContent1`. The `extension` element in `SimpleContent1` specifies that the derivation is based on `xsd:string`. The base type `xsd:string` is mapped to `System.String Value` in the `SimpleContent1` class.

Each attribute in the WSDL definition shown in Example 51 on page 192 is mapped to a field in the C# class. For example, `attrib1b` is mapped to `public System.String attrib1b` in `SimpleContent1` class.

The `SimpleContent2` type is derived from the `SimpleContent1` type. As a result, the `SimpleContent2` C# class (shown in Example 53) inherits from the `SimpleContent1` C# class.

Example 53: *SimpleContent2.cs*

```
[System.Serializable()]
public class SimpleContent2 : SimpleContent1 {

    public System.String attrib2a;

    public System.String attrib2b;
}
```

Occurrence 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 file example

Example 54 shows a WSDL sequence element with minOccurs and maxOccurs constraints:

Example 54: *WSDL Sequence with Occurrence Constraints*

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

CTS mapping

Example 54 maps to C# as follows:}

Example 55: *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;
     }
```

Example 55: *C# Mapping of WSDL Sequence with Occurrence Constraints*

```
  }
```

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 file example

Example 56 shows a WSDL definition of a SOAP array:

Example 56: *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 56 maps to C# as follows:

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

CHAPTER 13

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	page 198
Environment Variables	page 199

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:

```
InstallDir\artix\Version\etc\domains\artix.cfg
```

More information

See the Deploying and Managing 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_connect_env.bat` script or set the variables manually. See "Running the artix_connect_env.bat script" on page 201 and "Setting manually" on page 202 for more detail.

Environment variables

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

- `IT_PRODUCT_DIR`
- `IT_LICENSE_FILE`
- `IT_DOMAIN_NAME`
- `IT_CONFIG_DOMAINS_DIR`
- `IT_IDL_CONFIG_FILE`
- `PATH`
- `JETVMPROP`

The environment variables are explained in Table 6:

Table 6: *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 *`InstallDir`*`\etc\licenses.txt`

Table 6: *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:
	`*InstallDir*\artix\*Version*\etc\domains`
	For example:
	`C:\iona\ArtixConnect\artix\4.0\etc\domains`

`IT_IDL_CONFIG_FILE`	Specifies the configuration used by the Artix Connect IDL compiler. If this variable is not set, you will be unable to run the IDL to WSDL tools provided with Artix Connect. This variable is required for an Artix Connect development installation. The default location is:
	`%IT_PRODUCT_DIR%\artix\Version\etc\idl.cfg`
	For example:
	`C:\iona\ArtixConnect\artix\4.0\etc\idl.cfg`
	Note:
	Do not modify the default IDL configuration file.

Table 6: *Artix Connect Environment Variables*

Variable	Description
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= *InstallDir*\etc\licenses.txt For example: -Dcom.iona.artix.LicenseFile= C:\iona\ArtixConnect\etc\licenses.txt

Running the artix_connect_env.bat script

Artix Connect includes an artix_connect_env.bat script that you can use to set your Artix Connect environment. To set your Artix Connect environment, open a Windows command prompt and run the following command, from the *InstallDir*\artix*Version*\bin directory:

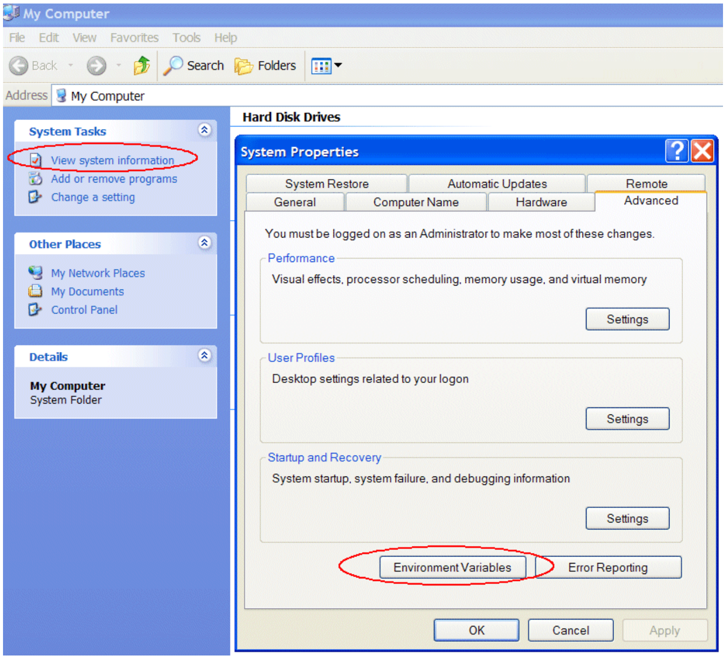
```
artix_connect_env.bat
```

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 55:

Figure 55: *Selecting My Computer*



You must be logged on as an Administrator to make most of these changes.

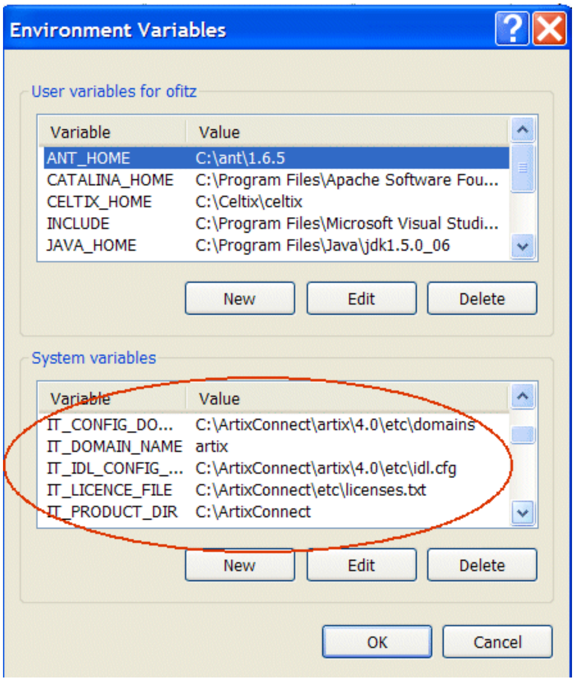
Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

2. Select the **Advanced** tab and click **Environment Variables**, as shown in Figure 55. The **Environment Variables** dialog box appears as shown in Figure 56:

Figure 56: *Setting Environment Variables Manually*



User variables for ofitz

Variable	Value
ANT_HOME	C:\ant\1.6.5
CATALINA_HOME	C:\Program Files\Apache Software Fou...
CELTIX_HOME	C:\Celtix\celtix
INCLUDE	C:\Program Files\Microsoft Visual Studi...
JAVA_HOME	C:\Program Files\Java\jdk1.5.0_06

System variables

Variable	Value
IT_CONFIG_DO...	C:\ArtixConnect\artix\4.0\etc\domains
IT_DOMAIN_NAME	artix
IT_IDL_CONFIG_...	C:\ArtixConnect\artix\4.0\etc\idl.cfg
IT_LICENCE_FILE	C:\ArtixConnect\etc\licenses.txt
IT_PRODUCT_DIR	C:\ArtixConnect

3. Add each of the environment variables, including the correct value for your installation, as described in “Environment variables”.

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

Index

Symbols

.NET clients
  building and running 63, 80, 100
  implementing in C# 61, 78, 97
  introduction to 23
.NET metadata
  generating from WSDL using GUI 52, 68, 90
-? 46

A

Add Artix Reference wizard 43
  fields 44
Add IDL Reference wizard 47
  fields 48
all complex types
  WSDL-to.NET mapping 180
arrays
  WSDL-to-.NET mapping 182
artix.cfg 198
artix_connect_env.bat 201
Artix Service WSDL file 44
atomic types
  WSDL-to.NET mapping 174
attributes
  WSDL-to.NET mapping 186
attributes with in-line data descriptions
  WSDL-to.NET mapping 188

B

bindings 150, 162
  supported 22
bridge
  introduction to 23

C

C#
  writing clients in 61, 78, 97
callbacks 107–119
  demonstration 110
  implementing 109
  implementing the client in C# 116
  implementing the server 119
  introduction to 108
  typical use case 108
  WSDL contract 112
choice complex types
  WSDL-to.NET mapping 184
Client 44, 48
clients. *See* .NET clients
complex type from simple type
  WSDL-to.NET mapping 192
complex types
  WSDL-to-.NET mapping 179
configuration domain 198
CORBA Object Reference file 48
CORBA Reference Wizard 47

D

deployment
  required components 104
  steps 104
  typical scenario 102
development prerequisites 42
document/literal wrapped style
  WSDL-to-.NET mapping 170

E

enumerations
  WSDL-to.NET mapping 191
environment variables 197–203
  IT_CONFIG_DOMAINS_DIR 200
  IT_DOMAIN_NAME 200
  IT_LICENSE_FILE 199
  IT_PRODUCT_DIR 199
  JETVMPROP 201
  PATH 201
  setting 199
  setting manually 202

G

graphical overview 21

H

Hello World demo

background information 35
building and running 31
client 35
location of 30
server 35
WSDL file 36

I
-impl 45
IT_CONFIG_DOMAINS_DIR 200
IT_DOMAIN_NAME 200
IT_LICENSE_FILE 199
IT_PRODUCT_DIR 199

J
JETVMPROP 201

L
lists
WSDL-to.NET mapping 176
locator 121

M
main screen 43
marshalling schemes
supported 22
messages
WSDL-to.NET mapping 168

N
-name 46
NameSpace 44, 48
-namespace 46

O
occurance contraints
WSDL-to.NET mapping 194
operations 159
WSDL-to-.NET mapping 167

P
PATH 201
ports 150
PortTypes
WSDL-to-.NET mapping 165
portTypes 150, 159
protocols
supported 22

Q
-quiet 45

S
sequence types
WSDL-to.NET mapping 180
Server 44
servers
implementing for client callbacks 119
Service 44, 48
services 162
session manager 135
simple types
WSDL-to-.NET mapping 173
SOAP arrays
WSDL-to.NET mapping 196
-source 45
system components 23

T
transports
supported 22

U
unsupported simple types
WSDL-to.NET mapping 178

V
-v 46
-verbose 45

W
W3C 150
Web Services Description Language, see WSDL
World Wide Web Consortium, see W3C
WSDL 149–162
WSDL file
introduction to 21
WSDL-to-.NET mapping 163–196
all complex types 180
arrays 182
atomic types 174
attributes 186
attributes with in-line descriptions 188

choice complex types 184
complex type from simple type 192
complex types 179
document/literal wrapped style 170
enumerations 191
lists 176
massages 168
occurance constraints 194
operations 167
PortTypes 165
sequence types 180
  simple types 173
  SOAP arrays 196
  unsupported simple types 178
wsdltodotnet
  arguments 45
  examples of using 46

X

XSD 151, 153