Microsoft

MCTS EXAM
70-503

Microsoft® .NET Framework 3.5–Windows® Communication Foundation

Bruce Johnson,
Peter Madziak,
Sara Morgan, with
GrandMasters

SELF-PACED
Training Kit

How to access your CD files

The print edition of this book includes a CD. To access the CD files, go to http://aka.ms/625655/files, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft Press

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008929790

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9 QWT 3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Ken Jones
Developmental Editor: Laura Sackerman
Project Editor: Valerie Woolley
Editorial Production: nSight, Inc.
Technical Reviewer: Kurt Meyer; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X14-15149

About the Authors

Bruce Johnson

Bruce Johnson is a partner at ObjectSharp Consulting in Toronto, Canada. For over 25 years, he has been involved in various parts of the computer industry, starting with UNIX and PL/1, through C++ and Microsoft Visual Basic (pre .NET), and, finally, all manner of Microsoft Windows applications and .NET technologies. His position as a consultant has allowed him to implement consumer-facing Web applications, Windows applications, and the whole gamut of service-based applications (Web Services, .NET remoting, and Windows Communication Foundation [WCF]). As well as having fun working just behind the bleeding edge of technology (you know, the place where stuff actually has to be delivered), he has given more than 200 presentations at conferences and user groups across North America. His writings include magazine columns and articles, a blog (found at *http://www.objectsharp.com/blogs/bruce*), and a number of Microsoft Press training kit books.

Peter Madziak

Peter Madziak is a senior consultant and instructor with ObjectSharp Consulting—a Microsoft Gold Partner based in Toronto, Canada. He is a technical leader with more than 10 years' experience helping development teams plan, design, and develop large software projects. Peter's primary focus over the past few years has been on helping customers understand service-oriented architecture (SOA), Workflow and Business Process Management (BPM), Web Services (both RESTful and WS-*), event-driven architecture (EDA), and, more important, how all these technologies and architectural styles can be reconciled into an architecture that aligns well with business needs.

As an SOA and BPM expert, Peter helps customers implement solutions, using technologies such as WCF, Windows Workflow, Microsoft BizTalk Server 2006, SQL Service Broker, and ASP.NET Web applications. You can visit his blog at *http://www.objectsharp.com/cs/blogs/pmadziak/default.aspx*.

Sara Morgan

Sara Morgan is a robotics software engineer with CoroWare, Inc., (*http://www.coroware.com*) and author of the newly released Programming Microsoft Robotics Studio (Microsoft Press, 2008). In addition to robotics, she has extensive experience with Microsoft SQL Server and Microsoft Visual Studio .NET and has been developing database-driven Web applications since the earliest days of Internet development.

Prior to joining CoroWare, she was an independent author and developer, and her main client was Microsoft. During that time, she co-wrote four training kits for Microsoft Press. Developers use these training kits to study for certification exam; the kits cover topics such as distributed development, Web application development, SQL Server query optimization, and SQL Server business intelligence. Sara has also written several articles for the online development journal, *DevX.com*, concerning Speech Server and the newly released Microsoft Robotics Studio. In early 2007, she was named a Microsoft Most Valuable Professional (MVP) for the Office Communications Server (OCS) group.

Contents at a Glance

Table of Contents

Introduction

This training kit is designed for developers who plan to take the Microsoft Certified Technology Specialist (MCTS) exam, Exam 70-503, as well as for developers who need to know how to develop Windows Communication Foundation (WCF)–based applications, using Microsoft .NET Framework 3.5. It's assumed that, before using this training kit, you already have a working knowledge of Microsoft Windows and Microsoft Visual Basic or C# (or both).

By using this training kit, you will learn how to do the following:

- Create and configure a WCF service.
- Implement a client for a WCF service, using different transport protocols.
- Provide security to a WCF application, using transport-level and message-level protection.
- Extend WCF by using custom behaviors, including message inspectors, parameter inspectors, and operation invokers.
- Perform end-to-end (E2E) tracing on a WCF application.

Hardware Requirements

The following hardware is required to complete the practice exercises:

- A computer with a 1.6-gigahertz (GHz) or faster processor.
- A minimum of 384 megabytes (MB) of random access memory (RAM).
- A minimum of 2.2 gigabytes (GB) of available hard disk space is required to install Microsoft Visual Studio 2008. Additionally, 75 MB of available hard disk space is required to install the labs.
- A DVD-ROM drive.
- A 1024 × 768 or higher resolution display with 256 colors or more.
- A keyboard and Microsoft mouse or compatible pointing device.

Software Requirements

The following software is required to complete the practice exercises:

- One of the following operating systems:
  - Windows Vista (any edition except Windows Vista Starter)
  - Windows XP with Service Pack 2 or later (any edition except Windows XP Starter)
  - Microsoft Windows Server 2003 with Service Pack 1 or later (any edition)

    - Windows Server 2003 R2 or later (any edition)
    - Windows Server 2008
- Microsoft Visual Studio 2008

NOTE Evaluation edition of Visual Studio included

A 90-day evaluation edition of Visual Studio 2008 Professional edition is included on a DVD that comes with this training kit.

Using the CD and DVD

A companion CD and an evaluation software DVD are included with this training kit. The companion CD contains the following:

- **Practice Tests** You can reinforce your understanding of how to create WCF applications in Visual Studio 2008 with .NET Framework 3.5 by using electronic practice tests that you can customize to meet your needs from the pool of Lesson Review questions in this book. Alternatively, you can practice for the 70-503 certification exam by using tests created from a pool of 200 realistic exam questions, which will give you enough different practice tests to ensure that you're prepared.
- **Practice Files** Most chapters in this training kit include code and practice files that are associated with the lab exercises at the end of every lesson. For some exercises, you are instructed to open a project prior to starting the exercise. For other exercises, you create a project on your own and then reference a completed project on the CD if you have a problem following the exercise procedures. Practice files can be installed to your hard drive by simply copying them to the desired directory. After copying the practice files from the CD to your hard drive, you must clear the *Read Only* attribute to work with the files on your hard drive.

  The instructions in an exercise will look like the following:

  1. Navigate to the <*InstallHome*>/Chapter8/Lesson1/Exercise1/<*language*>/Before directory and double-click the Exercise1.sln file to open the solution in Visual Studio.

     In the path described here, the <*InstallHome*> directory is the one into which you copy the sample files from the CD.

  Many samples require that you either run Visual Studio or the solution's executable files with Administrator privileges. Also be aware that antivirus software can interfere with some samples. As you work through an exercise, you are expected to add appropriate *Imports/using* statements as necessary.

- **eBook** An electronic version (eBook) of this training kit is included for use at times when you don't want to carry the printed book with you. The eBook is in Portable Document Format (PDF), and you can view it by using Adobe Acrobat or Adobe Reader. You can use the eBook to cut and paste code as you work through the exercises. Command-line commands should be typed directly into a command prompt, because pasting these commands sometimes causes the hyphen (-) to be misinterpreted in the command prompt.
- **Webcasts** Several webcasts are mentioned throughout this book. The companion CD has links that will take you directly to the webcasts.
- **Sample Chapters** Sample chapters from other Microsoft Press titles on Windows Communication Foundation. These chapters are in PDF format.
- **Evaluation software** The evaluation software DVD contains a 90-day evaluation edition of Visual Studio 2008 Professional edition in case you want to use it instead of a full version of Visual Studio 2008 to complete the exercises in this book.

Digital Content for Digital Book Readers: If you bought a digital-only edition of this book, you can enjoy select content from the print edition's companion CD.
Visit **http://www.microsoftpressstore.com/title/9780735625655** to get your downloadable content. This content is always up-to-date and available to all readers.

How to Install the Practice Tests

To install the practice test software from the companion CD to your hard disk, perform the following steps:

1. Insert the companion CD into your CD drive and accept the license agreement that appears onscreen. A CD menu appears.

NOTE Alternative installation instructions if AutoRun is disabled

If the CD menu or the license agreement doesn't appear, AutoRun might be disabled on your computer. Refer to the Readme.txt file on the CD-ROM for alternative installation instructions.

2. Click Practice Tests and follow the instructions on the screen.

How to Use the Practice Tests

To start the practice test software, follow these steps:

1. Click Start and select All Programs and Microsoft Press Training Kit Exam Prep.

   A window appears that shows all the Microsoft Press training kit exam prep suites that are installed on your computer.
2. Double-click the lesson review or practice test you want to use.

Lesson Review Options

When you start a lesson review, the Custom Mode dialog box appears, enabling you to configure your test. You can click OK to accept the defaults, or you can customize the number of questions you want, the way the practice test software works, which exam objectives you want the questions to relate to, and whether you want your lesson review to be timed. If you are retaking a test, you can select whether you want to see all the questions again or only those questions you previously skipped or answered incorrectly.

After you click OK, your lesson review starts. You can take the test as follows:

- To take the test, answer the questions and use the Next, Previous, and Go To buttons to move from question to question.
- After you answer an individual question, if you want to see which answers are correct, along with an explanation of each correct answer, click Explanation.
- If you would rather wait until the end of the test to see how you did, answer all the questions, and then click Score Test. You see a summary of the exam objectives that you chose and the percentage of questions you got right overall and per objective. You can print a copy of your test, review your answers, or retake the test.

Practice Test Options

When you start a practice test, you can choose whether to take the test in Certification Mode, Study Mode, or Custom Mode.

- **Certification Mode** Closely resembles the experience of taking a certification exam. The test has a set number of questions, it is timed, and you cannot pause and restart the timer.
- **Study Mode** Creates an untimed test in which you can review the correct answers and the explanations after you answer each question.
- **Custom Mode** Gives you full control over the test options so that you can customize them as you like.

In all modes, the user interface you see when taking the test is basically the same, but different options are enabled or disabled, depending on the mode. The main options are discussed in the previous section, "Lesson Review Options."

When you review your answer to an individual practice test question, a "References" section is provided. This section lists where in the training kit you can find the information that relates to that question, and it provides links to other sources of information. After you click Test Results to score your entire practice test, you can click the Learning Plan tab to see a list of references for every objective.

How to Uninstall the Practice Tests

To uninstall the practice test software for a training kit, use the Add Or Remove Programs option in Control Panel in Windows.

Microsoft Certified Professional Program

Microsoft certifications provide the best method to prove your command of current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop or implement and support solutions with Microsoft products and technologies. Computer professionals who become Microsoft-certified are recognized as experts and are sought after industrywide. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO List of Microsoft certifications

For a full list of Microsoft certifications, go to *http://www.microsoft.com/learning/mcp/default.mspx*.

Technical Support

Every effort has been made to ensure the accuracy of this book and the contents of the companion CD. If you have comments, questions, or ideas regarding this book or the companion CD, please send them to Microsoft Press by using either of the following methods:

E-mail: *tkinput@microsoft.com*

Postal Mail:

Microsoft Press
Attn: *MCTS Self-Paced Training Kit (Exam 70-503): Microsoft® .NET Framework 3.5–Windows® Communication Foundation* Editor
One Microsoft Way
Redmond, WA, 98052-6399

For additional support information regarding this book and the CD-ROM (including answers to commonly asked questions about installation and use), visit the Microsoft Press Technical Support Web site at *http://www.microsoft.com/learning/support/books*. To connect directly to the Microsoft Knowledge Base and enter a query, visit *http://support.microsoft.com/search*. For support information regarding Microsoft software, please connect to *http://support.microsoft.com*.

Evaluation Edition Software

The 90-day evaluation edition provided with this training kit is not the full retail product and is provided only for the purposes of training and evaluation. Microsoft and Microsoft Technical Support do not support this evaluation edition.

Information about any issues relating to the use of this evaluation edition with this training kit is posted in the Support section of the Microsoft Press Web site (*http://www.microsoft.com/learning/support/books/*). For information about ordering the full version of any Microsoft software, please call Microsoft Sales at (800) 426-9400 or visit *http://www.microsoft.com*.

Chapter 4

Consuming Services

Until this chapter, the primary focus of this book has been on building and configuring Windows Communication Foundation (WCF) services. However, the value of service orientation is to make certain capabilities (services) available for use (or consumption) in other programs, so in this chapter, the emphasis shifts to consuming services. The chapter begins with coverage of the mechanics of creating proxies to services and then discusses what you'll need to know to consume services effectively, using those proxies. Because one of the most powerful aspects of Extensible Markup Language (XML)-based Web services is that their consumption does not impose any platform, technology, or operating system constraints, the chapter will close with a look at how WCF can be used to consume services built on other platforms.

Exam objectives in this chapter:

- Create a service proxy.
- Call a service by using a service proxy.
- Consume non-WCF services.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have:

- A computer that meets or exceeds the minimum hardware requirements listed in the "About This Book" section at the beginning of the book.
- Any edition of Microsoft Visual Studio 2008 (including Microsoft Visual C# 2008 Express Edition or Microsoft Visual Basic 2008 Express Edition) installed on the computer.
- Access to the Visual Studio solutions for this chapter included on the companion CD.
- An active Internet connection for the lab following Lesson 2, "Consuming Non-WCF Services."

Real World

Peter Madziak

SOAP-based Web services have really come into their prime over the past few years when it comes to usage inside an enterprise or institution. Many companies are using Web services inside their organization for consumption by rich-client applications inside their enterprise, their own Web applications, and, in many cases, by partners in a business-to-business (B2B) setting. However, what are less common are Internet-facing SOAP services that are available for massive consumption by any number of consumers. They certainly exist; however, to use Gartner's term (see *http://en.wikipedia.org/wiki/Hype_cycle*), they are further behind on the hype cycle than inside-the-enterprise usage of SOAP-based Web services.

Even so, there are many who would argue that all this is about to change. Borrowing from Gartner again, Internet-facing Web services are rapidly approaching the "plateau of productivity". Whenever it happens, all will agree that it will be a fascinating era for software development, an era in which application development teams will be able to assemble applications more rapidly than ever before by consuming best-of-breed services on the Internet in much the same fashion as we consume utilities such as electricity and cable television services today. Central to working in this kind of an environment is being able to consume other services effectively (whether they are WCF services or not), which is the focus of this chapter. In the lab that follows the second lesson of this chapter, the exercise works through the steps of consuming Microsoft MapPoint, an Internet-facing mapping and geolocation Web service. This is an excellent example of a very powerful capability, namely, access to worldwide map images and data, which is available for use in applications more easily than it ever has been before. The true mark of its utility is that most application developers would never dream of trying to build this capability themselves and would be happy simply to consume it as a service offered by someone else.

Lesson 1: Consuming WCF Services

This lesson provides you with the tools you need to start consuming WCF services. Most Web services platforms, WCF included, provide developers with a mechanism for creating an object that can be used to communicate with the service. Such objects are called proxies, or proxy objects, because they are effectively acting as a proxy to the service. In this lesson, the focus is first on the four ways you can create a proxy to a WCF service and then on the details you need to consider to call services in the most effective manner using those proxies.

After this lesson, you will be able to:

- Generate proxy classes to WCF services, using the svcutil command-line utility and Visual Studio.
- Use *ChannelFactory* objects to create proxies dynamically to WCF services.
- Manually define classes whose instances can act as proxies to WCF services.
- Use proxies to call service operations, both synchronously and asynchronously.
- Create and use proxies to communicate with a service over a duplex channel.
- Create and use proxies to communicate with a non-WCF service.

Estimated lesson time: 40 minutes

Creating Proxies and Proxy Classes

WCF provides developers with several mechanisms for creating proxy objects that can be used to communicate with a Web service. This section covers these different approaches.

Generating Proxy Classes from Service Metadata

The most common approach to creating proxies is first to create a class based on the metadata of the service and then instantiate that class to create an actual proxy object. The metadata is typically accessed by engaging in a metadata exchange with the remote service. As Lesson 2 explores further, when the remote service is not a WCF service, the only option for representing that service metadata is through the standard Web Services Description Language (WSDL) approach. If the service is a WCF service, a richer WCF metadata exchange is possible. Either way, there are two ways of exchanging metadata, both of which are covered in this section. In the lab following this lesson, you use these mechanisms.

Using svcutil to Generate a Proxy Class WCF provides a command-line utility called svcutil that you can use to generate a proxy class to a service. To use it, you must open a Visual Studio command prompt or simply refer to its full path in a regular command prompt. Table 1-1 covers the most commonly used options you will need when using svcutil to generate proxy classes.

Table 4-1 Commonly Used Options for the svcutil Tool

Option	Description
/out:<file>	Specifies the filename for the generated code. Default: derived from the WSDL definition name, WSDL service name, or target namespace of one of the schemas (short form: */o*).
/config:<configFile>	Specifies the filename for the generated config file. Default: output.config.
/mergeConfig	Merges the generated configuration file into an existing file instead of overwriting the existing file.
/language:<language>	Indicates the programming language in which to generate code. Possible language names are *c#, cs, csharp, vb, visualbasic, c++*, and *cpp*. Default: *csharp* (short form: */l*).
/namespace:<targetNamespace, .NETNamespace>	Maps a WSDL or XML schema target namespace to a .NET namespace. Using an asterisk (*) for the target namespace maps all target namespaces without an explicit mapping to the matching .NET namespace. Default: derived from the target namespace of the schema document for Data contracts. The default namespace is used for all other generated types (short form: */n*).
/messageContract	Generates Message contract types (short form: */mc*).
/async	Generates both synchronous and asynchronous method signatures. Default: generates only synchronous method signatures (short form: */a*).
/serializer:XmlSerializer	Generates data types that use the *XmlSerializer* for serialization and deserialization.

The following is an example of how you might use the svcutil command to generate a proxy class to an *OrderEntryService* hosted at *http://localhost:8080/orders/*. (The commands are formatted on multiple lines to fit on the printed page.)

```
' VB
svcutil /l:VB /async /config:app.config /mergeConfig
    /namespace:*,SvcUtilProxy /out:ServiceUtilProxy.vb
    http://localhost:8080/orders/

// C#
svcutil /async /config:app.config /mergeConfig
    /namespace:*,SvcUtilProxy /out:ServiceUtilProxy.cs
    http://localhost:8080/orders/
```

Note a couple of things about this example:

- The default language is C#, so for the Visual Basic version, the language must be specified.

- Asynchronous operations are generated on the proxy because of the */async* option.
- It is common to use a wildcard such as * simply to map all XML namespaces encountered in the service metadata and Data contracts to a single .NET namespace. This is done in the preceding example using the **,SvcUtilProxy* pair.

MORE INFO More on svcutil

For more information on the svcutil command-line tool, consult the documentation from your local Visual Studio installation or online at *http://msdn.microsoft.com/en-us/library/aa347733.aspx*.

Using Visual Studio to Generate a Proxy Class In Visual Studio, you can add a service reference by right-clicking a project node in Solution Explorer and choosing Add Service Reference, as shown in Figure 4-1.

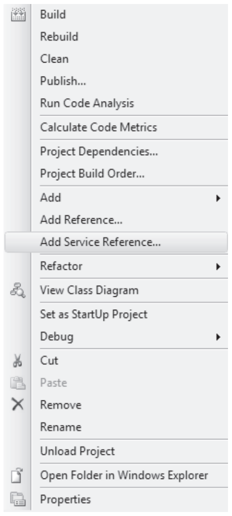
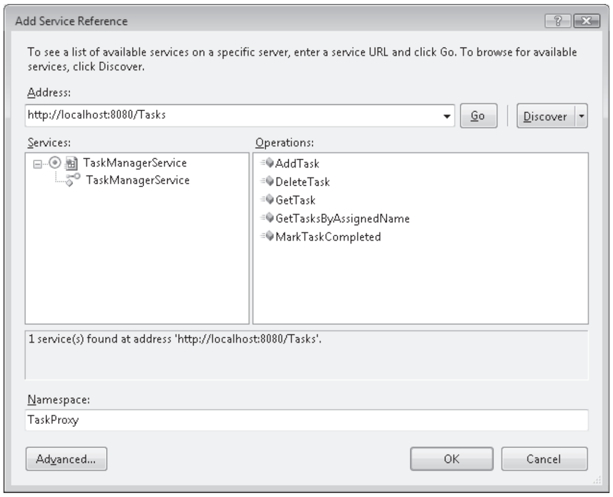


Figure 4-1 Adding a service reference using a context menu

In the resulting dialog box, enter the address of a service endpoint and click Go or click Discover to browse for available services. Click OK to have a proxy class generated for you. Figure 4-2 shows what the Add Service Reference dialog box might look like in the context of using Visual Studio to add a service reference to a task service.

To add asynchronous methods to your generated proxy class and control some of the collection types, first click the Advanced button in the Add Service Reference dialog box. Figure 4-3 shows the resulting Service Reference Settings dialog box.


Add Service Reference

To see a list of available services on a specific server, enter a service URL and click Go. To browse for available services, click Discover.

Address: http://localhost:8080/Tasks

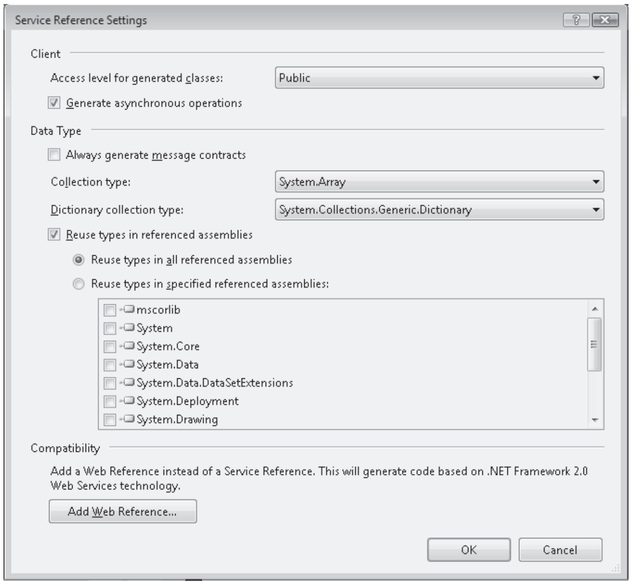
Services:
- TaskManagerService
  - TaskManagerService

Operations:
- AddTask
- DeleteTask
- GetTask
- GetTasksByAssignedName
- MarkTaskCompleted

1 service(s) found at address 'http://localhost:8080/Tasks'.

Namespace: TaskProxy

Figure 4-2 The Add Service Reference dialog box


Service Reference Settings

Client
- Access level for generated classes: Public
- [✓] Generate asynchronous operations

Data Type
- [ ] Always generate message contracts
- Collection type: System.Array
- Dictionary collection type: System.Collections.Generic.Dictionary
- [✓] Reuse types in referenced assemblies
  - (●) Reuse types in all referenced assemblies
  - ( ) Reuse types in specified referenced assemblies:
    - [ ] mscorlib
    - [ ] System
    - [ ] System.Core
    - [ ] System.Data
    - [ ] System.Data.DataSetExtensions
    - [ ] System.Deployment
    - [ ] System.Drawing

Compatibility

Add a Web Reference instead of a Service Reference. This will generate code based on .NET Framework 2.0 Web Services technology.

Figure 4-3 The Service Reference Settings dialog box for adding a service reference

Visual Studio uses the same svcutil tool, but it provides you with a friendly interface and takes care of some of the manual steps you need to perform when using svcutil yourself, such as adding the reference to *System.ServiceModel*, adding the resulting proxy code to your project, and so on.

Visual Studio Service Reference vs. svcutil

Which one is the better approach: using Visual Studio or svcutil to create your proxy classes? In many cases, it's a matter of developer preference, but here are a few points to consider:

- Using the svcutil utility gives you the most control.
- Using the svcutil utility requires additional manual steps such as adding the reference to *System.ServiceModel*, adding the resulting proxy code to your project, and so on, that Visual Studio does automatically.
- The svcutil utility can be useful in an automated build scenario in which you want to generate proxy classes as part of the automated build process.

Manually Defining a Proxy Class

As opposed to having a tool generate a proxy class for you, you can manually define a proxy class on your own using the same base classes that the tools use. Suppose you have the following Service contract:

```
' VB
<ServiceContract()> _
Public Interface IOrderEntryService
    <OperationContract()> _
    Function SubmitOrder(ByVal order As Order) _
            As OrderAcknowledgement

    ' Etc...
End Interface

// C#
[ServiceContract()]
public interface IOrderEntryService
{
    [OperationContract()]
    OrderAcknowledgement SumbitOrder(Order order);

    // Etc...
}
```

You could manually define a proxy class based on that contract as follows:

```
' VB
Public Class OrderEntryServiceProxy
    Inherits ClientBase(Of IOrderEntryService)
    Implements IOrderEntryService

    Public Sub New(ByVal binding As Binding, _
                   ByVal epAddr As EndpointAddress)
```

```
        MyBase.New(binding, epAddr)
    End Sub

    Public Sub New(ByVal endpointConfigurationName As String)
        MyBase.New(endpointConfigurationName)
    End Sub

    Public Function SubmitOrder(ByVal order As Order) _
        As OrderAcknowledgement _
            Implements IOrderEntryService.SubmitOrder
        Return Me.Channel.SubmitOrder(order)
    End Function
End Class

// C#
public class OrderEntryServiceProxy :
        ClientBase<IOrderEntryService>,IOrderEntryService
{
    public OrderEntryServiceProxy(
            Binding binding, EndpointAddress epAddr)
        : base(binding,epAddr)
    {
    }

    public OrderEntryServiceProxy(
            string endpointConfigurationName)
        : base(endpointConfigurationName)
    {
    }

    public OrderAcknowledgement SumbitOrder(Order order)
    {
        return this.Channel.SumbitOrder(order);
    }
}
```

If a callback channel is involved (perhaps the Service contract specifies a Callback contract because at least one of the service operations uses the Duplex message exchange pattern [MEP]), the base class should be *DuplexClientBase* instead of *ClientBase*. There are more steps to setting up a callback channel, as is explained later in this lesson and in the lab that follows.

Dynamically Creating a Proxy

In some cases, you don't need to have a proxy class explicitly defined anywhere because WCF provides the *ChannelFactory* class as a means of dynamically creating a proxy object based on the Service contract alone. Without ever explicitly generating a proxy class or manually defining one, you can create a proxy object, using only the Service contract and the *ChannelFactory* class. The following code shows how this would be done.

```
' VB
Dim binding As Binding
binding = New NetTcpBinding

Dim factory As ChannelFactory(Of IOrderEntryService)
factory = New ChannelFactory(Of IOrderEntryService)( _
    binding, "net.tcp://localhost:6789/orders/")

Dim proxy As IOrderEntryService
proxy = factory.CreateChannel()

Try
    Dim order As New Order
    order.Product = "Widget-ABC"
    order.Quantity = 10
    ' Etc...

    Dim ack As OrderAcknowledgement
    ack = proxy.SubmitOrder(order)

    Console.WriteLine( _
        "Order submitted; tracking number: {0}", _
        ack.TrackingNumber)
Catch ex As Exception
    Console.WriteLine("Error: {0}", ex.Message)
End Try

// C#
Binding binding = new NetTcpBinding();
ChannelFactory<IOrderEntryService> factory;
factory = new ChannelFactory<IOrderEntryService>(
    binding, "net.tcp://localhost:6789/orders/");

try
{
    IOrderEntryService proxy = factory.CreateChannel();

    Order order = new Order();
    order.Product = "Widget-ABC";
    order.Quantity = 10;
    // Etc...

    OrderAcknowledgement ack = proxy.SumbitOrder(order);

    Console.WriteLine(
        "Order submitted; tracking number: {0}",
        ack.TrackingNumber);
}

catch (Exception ex)
{
    Console.WriteLine("Error: {0}",ex.Message);
}
```

BEST PRACTICES **Generate proxy classes from service metadata**

One of the tenets of service orientation is that consumers should depend only on a service's schema and not on any of the service's classes. The only way consumers of a service should be coupled to the service is through the schema of the messages they must exchange with the service. That is certainly better than some of the middleware technologies of the past such as DCOM, CORBA, and .NET Remoting, which required consumers to have a code or binary-level dependency on the service and the request and response objects.

If you manually create proxy classes or use the *ChannelFactory* class to create proxy objects dynamically, you need to have access to the WCF Service contract. If you are not getting access to the service contract by generating code from service metadata, that means you must have a code or binary-level reference to the WCF Service contract. Avoid that form of coupling when building services. Always strive to minimize the coupling between a service and its consumers. In short, always generate proxy classes from service metadata, either by using svcutil or by adding a service reference in Visual Studio.

Using Proxies to Call Services

In one sense, using a proxy to call a service is as simple as calling a method on the proxy object, which the WCF plumbing will then translate into a message and send the message to the wire-level transport layer. Suppose you have the following service:

```
' VB
<DataContract()> _
Public Class Order
    <DataMember()> _
    Public Product As String

    <DataMember()> _
    Public Quantity As Integer

    ' Etc...
End Class

<DataContract()> _
Public Class OrderAcknowledgement
    <DataMember()> _
    Public TrackingNumber As String

    ' Etc...
End Class

<ServiceContract()> _
Public Interface IOrderEntryService
    <OperationContract()> _
    Function SubmitOrder(ByVal order As Order) _
            As OrderAcknowledgement
    ' Etc...
End Interface

Public Class OrderEntryService
    Implements IOrderEntryService

    Public Function SubmitOrder(ByVal order As Order) _
        As OrderAcknowledgement _
            Implements IOrderEntryService.SubmitOrder
        Dim ack As New OrderAcknowledgement
        ack.TrackingNumber = "alpha-bravo-123"
        Return ack
    End Function

    ' Etc...
End Class

// C#
[DataContract()]
public class Order
{
    [DataMemberAttribute()]
    public string Product;

    [DataMemberAttribute()]
    public int Quantity;

    // Etc...
}

[DataContract()]
public class OrderAcknowledgement
{
    [DataMemberAttribute()]
    public string TrackingNumber;

    // Etc...
}

[ServiceContract()]
public interface IOrderEntryService
{
    [OperationContract()]
    OrderAcknowledgement SumbitOrder(Order order);

    // Etc...
}

public class OrderEntryService : IOrderEntryService
{
    public OrderAcknowledgement SumbitOrder(Order order)
    {
        OrderAcknowledgement ack = new OrderAcknowledgement();
        ack.TrackingNumber = "alpha-bravo-123";
        return ack;
    }

    // Etc...
}
```

You could use the proxy to call such a service in a number of ways. First, you could use the *ChannelFactory* class to create a proxy dynamically, as shown here. (In this section, the code that differs among the three methods of using a proxy is shown in bold.)

```
' VB
Dim binding As Binding
binding = New NetTcpBinding
Dim factory As ChannelFactory(Of IOrderEntryService)
factory = New ChannelFactory(Of IOrderEntryService)( _
    binding, "net.tcp://localhost:6789/orders/")
Dim proxy As IOrderEntryService
proxy = factory.CreateChannel()
Dim order As New Order
order.Product = "Widget-ABC"
order.Quantity = 10
' Etc...

Dim ack As OrderAcknowledgement
ack = proxy.SubmitOrder(order)

Console.WriteLine( _
    "Order submitted; tracking number: {0}", _
    ack.TrackingNumber)

// C#
Binding binding = new NetTcpBinding();
ChannelFactory<IOrderEntryService> factory;
factory = new ChannelFactory<IOrderEntryService>(
    binding, "net.tcp://localhost:6789/orders/");
IOrderEntryService proxy = factory.CreateChannel();
Order order = new Order();
order.Product = "Widget-ABC";
order.Quantity = 10;
// Etc...

OrderAcknowledgement ack = proxy.SumbitOrder(order);

Console.WriteLine(
    "Order submitted; tracking number: {0}",
        ack.TrackingNumber);
```

Second, you could manually define a proxy class as follows:

```
' VB
Public Class OrderEntryServiceProxy
    Inherits ClientBase(Of IOrderEntryService)
    Implements IOrderEntryService

    Public Sub New(ByVal binding As Binding, _
                   ByVal epAddr As EndpointAddress)
        MyBase.New(binding, epAddr)
    End Sub

    Public Sub New(ByVal endpointConfigurationName As String)
        MyBase.New(endpointConfigurationName)
    End Sub

    Public Function SubmitOrder(ByVal order As Order) _
        As OrderAcknowledgement _
            Implements IOrderEntryService.SubmitOrder
        Return Me.Channel.SubmitOrder(order)
    End Function
End Class

// C#
public class OrderEntryServiceProxy :
        ClientBase<IOrderEntryService>,IOrderEntryService
{
    public OrderEntryServiceProxy(
            Binding binding, EndpointAddress epAddr)
        :base(binding,epAddr)
    {
    }

    public OrderEntryServiceProxy(
            string endpointConfigurationName)
        : base(endpointConfigurationName)
    {
    }

    public OrderAcknowledgement SumbitOrder(Order order)
    {
        return this.Channel.SumbitOrder(order);
    }
}
```

You would use the manually defined proxy class like this:

```
' VB
Dim binding As Binding
binding = New NetTcpBinding
Dim epAddr As EndpointAddress
epAddr = New EndpointAddress( _
    "net.tcp://localhost:6789/orders/")
Dim proxy As IOrderEntryService
proxy = New OrderEntryServiceProxy(binding, epAddr)
Dim order As New Order
order.Product = "Widget-ABC"
order.Quantity = 10
```

```
' Etc...

Dim ack As OrderAcknowledgement
ack = proxy.SubmitOrder(order)

Console.WriteLine( _
    "Order submitted; tracking number: {0}", _
    ack.TrackingNumber)

// C#
Binding binding = new NetTcpBinding();
EndpointAddress epAddr = new EndpointAddress(
    "net.tcp://localhost:6789/orders/");
IOrderEntryService proxy =
    new OrderEntryServiceProxy(binding,epAddr);  Order order = new Order();
order.Product = "Widget-ABC";
order.Quantity = 10;
// Etc...

OrderAcknowledgement ack = proxy.SumbitOrder(order);

Console.WriteLine("Order submitted; tracking number: {0}",
    ack.TrackingNumber);
```

Finally, if you used either svcutil or Visual Studio to generate a proxy class, you would use the generated proxy class like this:

```
' VB
Dim proxy As OrderEntryServiceClient
proxy = New OrderEntryServiceClient()
Dim order As New Order
order.Product = "Widget-ABC"
order.Quantity = 10
' Etc...

Dim ack As OrderAcknowledgement
ack = proxy.SubmitOrder(order)

Console.WriteLine( _
    "Order submitted; tracking number: {0}", _
    ack.TrackingNumber)

// C#
OrderEntryServiceClient proxy = new OrderEntryServiceClient();
Order order = new Order();
order.Product = "Widget-ABC";
order.Quantity = 10;
// Etc...

OrderAcknowledgement ack = proxy.SumbitOrder(order);

Console.WriteLine("Order submitted; tracking number: {0}",
    ack.TrackingNumber);
```

Recall that in Chapter 1, "Contracts," it was suggested that to design services well you must let go of many of the concepts of object-oriented programming. This is not only true for the design of the document-centric messages going in and out of your service; you must also always keep in mind what is being done for you by the plumbing. When you make a method call on a proxy, the WCF plumbing translates your method call into a message that it sends over the wire-level transport layer, so you always need to be aware of issues such as:

- How slow that might be in comparison to a normal method call on an in-process object.
- How network communication troubles could result in an inability to communicate with the remote service.

The following sections cover some of the other considerations you need to keep in mind to use proxies effectively to communicate with a service.

Invoking Service Operations Asynchronously

In many cases, a method call on a proxy, which is translated into a message that is sent to a remote service, might take longer than the consuming application can reasonably wait. The reason for the slowness can be poor network bandwidth, large message size, or a combination thereof. In GUI applications, you don't want to keep the UI unresponsive for any length of time while the application waits for a response from the service. In non-GUI applications as well, the call to a service typically must be done as quickly as possible.

A good solution is to invoke the service operation asynchronously. You can use the */async* option on svcutil or in Visual Studio. For the latter, click Advanced in the Add Service Reference dialog box, and then select the Generate Asynchronous Methods check box in the Service Reference Settings dialog box to generate a *Begin-* and *End-* pair of methods for each service operation. The method pairs together support the asynchronous invocation of a service operation.

To show how these method pairs are used, suppose you have generated a proxy to *OrderEntryService* with asynchronous methods. In addition to the *SubmitOrder* operation on the proxy class, there would be a corresponding *BeginSubmitOrder* and *EndSubmitOrder* method pair. This *Begin-* and *End-* method pair could be used as follows to invoke the *SubmitOrder* operation asynchronously:

```
' VB
Dim proxy As OrderEntryServiceClient
proxy = New OrderEntryServiceClient()

Dim order As New Order
order.Product = "Widget-ABC"
order.Quantity = 10
' Etc...

Dim cb As AsyncCallback
```

```
cb = New AsyncCallback(AddressOf HandleCallback)

proxy.BeginSubmitOrder(order, cb, proxy)

Console.WriteLine( _
    "Order submitted asynchronously; waiting for callback")

// C#
OrderEntryServiceClient proxy = new OrderEntryServiceClient();

Order order = new Order();
order.Product = "Widget-ABC";
order.Quantity = 10;
// Etc...

AsyncCallback cb = new AsyncCallback(HandleCallback);
proxy.BeginSumbitOrder(order, cb, proxy);

Console.WriteLine(
    "Order submitted asynchronously; waiting for callback");
```

The *HandleCallback* method, which might be defined as shown here, is called back when the asynchronously invoked operation completes:

```
' VB
Public Shared Sub HandleCallback( _
        ByVal result As IAsyncResult)
    Dim proxy As OrderEntryServiceClient
    proxy = result.AsyncState

    Dim ack As OrderAcknowledgement
    ack = proxy.EndSubmitOrder(result)
    Console.WriteLine( _
        "Order submitted; tracking number: {0}", _
        ack.TrackingNumber)

End Sub

// C#
static void HandleCallback(IAsyncResult result)
{
    OrderEntryServiceClient proxy =
        result.AsyncState as OrderEntryServiceClient;
    OrderAcknowledgement ack = proxy.EndSumbitOrder(result);
    Console.WriteLine(
        "Order submitted; tracking number: {0}",
        ack.TrackingNumber);
}
```

In the lab that follows this lesson, you use this technique.

Closing Proxies

It is a good practice to close service proxies whenever the client is finished using them. One of the main reasons for doing so is that when a session has been established between the client and the service, closing the proxy not only closes the connection to the service but also terminates the session with the service. The importance of this will become clearer when sessions are covered in Chapter 10, “Sessions and Instancing.”

Instead of calling the *Close* method explicitly, you can also use the *Dispose* method, which will close the proxy as well. As usual, the advantage of the *Dispose* method is that you can use it in the context of a *using* statement so that it is called implicitly even if an exception occurs:

```
' VB
Using proxy As MyServiceClient = New MyServiceClient()
    proxy.SomeOp1()
End Using

// C#
using(MyServiceClient proxy = new MyServiceClient())
{
    proxy.SomeOp1();
}
```

Alternatively, if the object reference’s type is the Service contract interface, as opposed to a concrete proxy class, you can use the following variation:

```
' VB
Dim proxy As IMyService
proxy = New MyServiceClient()
Using proxy As IDisposable
   proxy.SomeOp1()
End Using

// C#
IMyService proxy = new MyServiceClient();
using (proxy as IDisposable)
{
    proxy.SomeOp1();
}
```

Duplex Channels with Proxies

To enable a proxy to communicate using a Duplex, or callback, channel, you must perform the following steps:

- Define a class that implements the Callback contract.
- Construct an instance of the class implementing the Callback contract and pass the instance to an *InstanceContext* constructor.

- Pass the *InstanceContext* object to the constructor of the proxy class.
- The proxy class must inherit from *DuplexClientBase* instead of from *ClientBase*.

Typically, the easiest way to handle this is to wrap the autogenerated proxy with another class that deals with the *InstanceContext* details and acts as the callback object (implementing the Callback contract). In Exercise 4, "Consume a Service Using a Callback Channel," of this lesson, you step through this process in greater detail.

Exam Tip Pay close attention to the base type of your proxy classes. If you are defining a proxy that uses OneWay or Request/Response MEPs, the right base class is *ClientBase<IMyContract>*, where *IMyContract* is your Service contract. If your proxy uses the Duplex MEP, the proxy class must inherit from *DuplexClientBase<IMyContract>*.

Service Agents

It is very common to want to wrap usage of a proxy in another class that has some additional capabilities in terms of interacting with the remote service. The generic term for these more capable objects that facilitate access to a remote service is *service agent*. The following are some reasons you might want to wrap access to a proxy in a service agent:

- The client might have limited or unpredictable connectivity, or the consumer might simply need to operate in an offline mode.
- Performance problems associated with service calls might necessitate actions such as client-side caching, request batching, aggregate result disassembly, and so on.
- A proxy class might be very awkward or inefficient to work with, for example, taking several calls that could be wrapped up into one by an agent.

In Exercise 4 in this lesson, you will see a very simple example of when you might use an agent. In the lab that follows Lesson 2, you will see an even better example of a service agent being used.

Quick Check

1. If you want to generate a proxy class that uses the *XmlSerializer*, which of the four methods for generating a proxy would be best to use?
2. Does it ever make sense to invoke a OneWay operation asynchronously?

Quick Check Answers

1. The only way to generate a proxy if you want to use the *XmlSerializer* is to use the svcutil command-line tool with the */serializer:XmlSerializer* option. Adding a service reference in Visual Studio does not support this option. As long as the contract specifies usage of the *XmlSerializer*, dynamically creating a proxy object by using the *ChannelFactory* class would also ensure that the right serializer is used, but in that case, you are generating an object dynamically rather than a class. Similarly, if you hand-code a proxy class, you are not technically *generating* a class and, again, the right serializer depends on whether the contract is declared to use the *XmlSerializer*.
2. Yes. If there is a situation in which a message being sent to a service in a OneWay operation was very large and therefore time-consuming, it can still make sense to invoke the operation asynchronously. Keep in mind that, as was discussed in Chapter 1, OneWay calls on the proxy complete only when the dispatcher on the service end has successfully dispatched the incoming message to a call on a service type instance. If the message is large, this can take longer to happen than you would like to wait in a synchronous invocation setting, thereby making asynchronous invocation an attractive alternative.

Lab: Creating and Using WCF Service Proxies

In this lab, you will use the techniques that have been covered to create WCF proxies to WCF services. The lab focuses on the different ways in which the proxy classes or objects can be created and on some of the key points for each way that you should consider in using the proxies effectively. In the first three exercises, you create proxies to communicate with the Task Manager service that was first built in Chapter 1. In the fourth exercise, you create and use a proxy that interacts with the service by using a Duplex, or callback, channel. This lab requires that you start a WCF service, so you must have the appropriate permissions to do so. You might want to run Visual Studio as an Administrator.

▶ Exercise 1 Create a Proxy Dynamically

In this exercise, you will use the *ChannelFactory* class to create a proxy object dynamically to communicate with the Task Manager service.

1. Navigate to the <*InstallHome*>/Chapter4/Lesson1/Exercise1/<*language*>/Before directory and double-click the Exercise1.sln file to open the solution in Visual Studio.

   The solution consists of four projects:

2. Because the solution still contains the Windows Forms client that you used in Chapter 1 to consume the Task Manger service, first explore the code there to see how that proxy code is used. After you have explored the code in this sample application, you can start creating your own proxies.
3. Add a new Console project called **DynGenProxy** to the solution.
4. To this new Console project, add references to both *System.ServiceModel* and *System .Runtime.Serialization*.
5. Add project references to both *Tasks.Services* and *Tasks.Entities*.
6. In the main code file (Program.cs or Module1.vb as appropriate), add the following imports:

```
' VB
Imports System.ServiceModel
Imports System.ServiceModel.Channels
Imports Tasks.Entities
Imports Tasks.Services

// C#
using System.ServiceModel;
using System.ServiceModel.Channels;
using Tasks.Entities;
using Tasks.Services;
```

7. Define the *Main* method (*Program.Main* or *Module1.Main* as appropriate) so that it matches the following code, which uses the *ChannelFactory* class to create a proxy dynamically to the task service:

```
' VB
Sub Main()
    Dim binding As Binding
    binding = New BasicHttpBinding

    Dim factory As ChannelFactory(Of ITaskManagerService)
    factory = New ChannelFactory(Of ITaskManagerService)( _
        binding, "http://localhost:8080/Tasks/TaskManager")

    Dim proxy As ITaskManagerService
    proxy = factory.CreateChannel()

    Try
        Dim task As New Task()
        task.CreatedBy = "Vicki"
        task.AssignedTo = "Ian"
        task.DateCreated = DateTime.Now
        task.DateLastModified = task.DateCreated
        task.Description = "Clean your room"
        task.DueDate = DateTime.Now.AddDays(3)

        Dim ack As TaskAcknowledgement
        ack = proxy.AddTask(task)

        Console.WriteLine( _
            "Task number {0} added to service", _
            ack.TaskNumber)
    Catch ex As Exception
        Console.WriteLine("Error: {0}", ex.Message)
    End Try
End Sub

// C#
static void Main(string[] args)
{
    Binding binding = new BasicHttpBinding();

    ChannelFactory<ITaskManagerService> factory;
    factory = new ChannelFactory<ITaskManagerService>(
            binding, "http://localhost:8080/Tasks/TaskManager");

    try
    {
        ITaskManagerService proxy = factory.CreateChannel();

        Task task = new Task();
        task.CreatedBy = "Vicki";
        task.AssignedTo = "Ian";
        task.DateCreated = DateTime.Now;
        task.DateLastModified = task.DateCreated;
        task.Description = "Clean your room";
        task.DueDate = DateTime.Now.AddDays(3);

        TaskAcknowledgement ack = proxy.AddTask(task);

        Console.WriteLine(
            "Task number {0} added to service",
            ack.TaskNumber);
    }

    catch (Exception ex)
    {
        Console.WriteLine("Error: {0}",ex.Message);
    }
}
```

8. Build the solution.
9. Making sure the ServiceConsoleHost project is the startup project, start the service.
10. Make DynGenProxy the startup project and run this Console project.

    You should see that the Console application successfully submits a task to the service.
11. Leave the task service running; you will need it to be running for the next two exercises.

▶ Exercise 2 Generate a Proxy Class, Using svcutil

In this exercise, you will use the svcutil command-line utility to generate a proxy class that you then use to communicate with the Task Manager service. You also asynchronously invoke one of the operations on this service, using an instance of the autogenerated proxy class.

1. Navigate to the <*InstallHome*>/Chapter4/Lesson1/Exercise2/<*language*>/Before directory and double-click the Exercise2.sln file to open the solution in Visual Studio.

   The solution consists of the four projects you started with in Exercise 1, "Create a Proxy Dynamically."
2. Add a new Console project called **SvcUtilProxy** to the solution.
3. To this new Console project, add a new application configuration file item called **app.config**.
4. Open a Visual Studio command prompt to the directory in which this project resides, in this case, <*InstallHome*>/Chapter4/Lesson1/Exercise2/<*language*>/Before/SvcUtilProxy.
5. With the service still running from step 9 in Exercise 1, execute the following command to generate a proxy class. (Enter it as a single command; it is formatted here on multiple lines to fit on the printed page.)

   During this step, you might be informed that the configuration file must be reloaded; if so, just click Yes.

```
' VB
svcutil /l:VB /async /config:app.config
    /namespace:*,SvcUtilProxy /out:TaskServiceProxy.vb
    http://localhost:8080/Tasks

// C#
svcutil /async /config:app.config
    /namespace:*,SvcUtilProxy /out:TaskServiceProxy.cs
    http://localhost:8080/Tasks
```

6. Add the TaskServiceProxy (.cs or .vb as appropriate) file, which was just generated by the svcutil command, to the SvcUtilProxy project.
7. To the SvcUtilProxy project, add references to both *System.ServiceModel* and *System.Runtime.Serialization*. If you are working in C#, also add a project reference to *Tasks.Entities*.
8. In the main code file (Program.cs or Module1.vb as appropriate), add the following imports, noting that there is an intentional difference between the Visual Basic and C# versions, stemming from the different ways project references are handled in Visual Basic.NET compared to C#:

```
' VB
Imports System.ServiceModel
Imports System.ServiceModel.Channels
Imports SvcUtilProxy.SvcUtilProxy
```

```
// C#
using System.ServiceModel;
using System.ServiceModel.Channels;
using Tasks.Entities;
```

To experiment with the mechanics of asynchronously invoking a service operation by using a proxy, you first need to define a function that will be called back when the asynchronously invoked operation completes.

9. Define the following function directly below the *Main* method in the main code file (Program.cs or Module1.vb as appropriate):

```
' VB
Sub HandleTaskAdded(ByVal ar As IAsyncResult)
    Dim proxy As TaskManagerServiceClient
    proxy = CType(ar.AsyncState, TaskManagerServiceClient)

    Dim ack As TaskAcknowledgement
    ack = proxy.EndAddTask(ar)

    Console.WriteLine( _
        "Task number {0} was added to service", _
        ack.TaskNumber)

End Sub

// C#
static void HandleTaskAdded(IAsyncResult ar)
{
    TaskManagerServiceClient proxy = ar.AsyncState
        as TaskManagerServiceClient;

    TaskAcknowledgement ack = proxy.EndAddTask(ar);

    Console.WriteLine(
        "Task number {0} was added to service",
        ack.TaskNumber);
}
```

10. Define the *Main* method (*Program.Main* or *Module1.Main* as appropriate) so that it matches the following code, which instantiates the proxy class generated by the svcutil command and invokes the *AddTask* operation asynchronously:

```
' VB
Sub Main()
    Dim proxy As TaskManagerServiceClient
    proxy = New TaskManagerServiceClient()

    Try
        Dim task As New Task()
        task.CreatedBy = "Eric"
```

```
            task.AssignedTo = "Ian"
            task.DateCreated = DateTime.Now
            task.DateLastModified = task.DateCreated
            task.Description = "Practice your saxophone"
            task.DueDate = DateTime.Now.AddDays(3)

            Dim cb As AsyncCallback
            cb = New AsyncCallback(AddressOf HandleTaskAdded)
            proxy.BeginAddTask(task, cb, proxy)

            Console.WriteLine( _
                "Asynchronously adding a task to the service; " + _
                "Press Enter to exit")
            Console.ReadLine()
        Catch ex As Exception
            Console.WriteLine("Error: {0}", ex.Message)
        End Try
    End Sub

    // C#
    static void Main(string[] args)
    {
        try
        {
            TaskManagerServiceClient proxy =
                new TaskManagerServiceClient();

            Task task = new Task();
            task.CreatedBy = "Eric";
            task.AssignedTo = "Ian";
            task.DateCreated = DateTime.Now;
            task.DateLastModified = task.DateCreated;
            task.Description = "Practice your saxophone";
            task.DueDate = DateTime.Now.AddDays(3);

            AsyncCallback cb = new AsyncCallback(HandleTaskAdded);
            proxy.BeginAddTask(task,cb,proxy);

            Console.WriteLine(
                "Asynchronously adding a task to the service; " +
                "Press Enter to exit");

            Console.ReadLine();
        }

        catch (Exception ex)
        {
            Console.WriteLine("Error: {0}", ex.Message);
        }
    }
```

NOTE Usage of */async*

This exercise uses the */async* option on the svcutil command to generate operations on the proxy that support asynchronous invocation. Specifically, the *BeginAddTask* and *EndAddTask* operations are generated. They are used in tandem by this exercise's version of the consumer to handle the asynchronous invocation of the *AddTask* operation.

11. Build the solution.
12. Making sure the service is still running and that SvcUtilProxy is the startup project, run this Console project.

    You should see that the Console application successfully submits a task to the service asynchronously and that the *HandleTaskAdded* callback method is called when the operation completes.
13. Again, leave the Task Manager service running for the next exercise.

▶ Exercise 3 Generating a Proxy Class by Adding a Service Reference in Visual Studio

In this exercise, you will use Visual Studio to generate a proxy class by adding a service reference. You instantiate the resulting proxy class and use the instance to communicate with the Task Manager service, this time adding a *FaultException* handler that shows how you access the *FaultInfo* class, which the service contract has declared it might issue. (Chapter 9, "When Simple Is Not Sufficient," discusses in detail handling faults and exceptions on clients.)

1. Navigate to the <*InstallHome*>/Chapter4/Lesson1/Exercise3/<*language*>/Before directory and double-click the Exercise3.sln file to open the solution in Visual Studio.

   The solution consists of the four projects you started with in Exercise 1.
2. Add a new Console project called **VSProxy** to the solution.
3. To this new Console project, add a service reference, making sure the Task Manager service is running, by right-clicking the project node in Solution Explorer and choosing Add Service Reference.
4. In the Add Service Reference dialog box, enter the **http://localhost:8080/Tasks** URL at which the service is hosted and click the Go button to list the services at that URL. After you enter **Tasks** for the namespace, click OK.

   Note that this step takes care of adding the app.config file and any needed references to the project.
5. In the main code file (Program.cs or Module1.vb as appropriate), add the following import statements:

```
' VB
Imports VSProxy.Tasks
Imports System.ServiceModel
```

```
// C#
using VSProxy.Tasks;
using System.ServiceModel;
```

6. Define the *Main* method (*Program.Main* or *Module1.Main* as appropriate) so that it matches the following code, which instantiates the proxy class generated by Visual Studio when you added the service reference:

```
' VB
Sub Main()
    Dim proxy As TaskManagerServiceClient
    proxy = New TaskManagerServiceClient()

    Try
        Dim task As New Task()
        task.CreatedBy = "Eric"
        task.AssignedTo = "Vicki"
        task.DateCreated = DateTime.Now
        task.DateLastModified = task.DateCreated
        task.Description = "Do the laundry"
        task.DueDate = DateTime.Now.AddDays(3)

        Dim ack As TaskAcknowledgement
        ack = proxy.AddTask(task)

        Dim taskNum As Integer
        taskNum = ack.TaskNumber
        Console.WriteLine( _
            "Task number {0} added to service", _
            taskNum)

        ' Now try to mark that same task
        ' as completed:
        proxy.MarkTaskCompleted(taskNum)

    Catch fault As FaultException(Of FaultInfo)
        Console.WriteLine("Error: {0}", fault.Detail.Reason)
    End Try
End Sub

// C#
static void Main(string[] args)
{
    try
    {
        TaskManagerServiceClient proxy = new TaskManagerServiceClient();

        Task task = new Task();
        task.CreatedBy = "Eric";
        task.AssignedTo = "Vicki";
        task.DateCreated = DateTime.Now;
```

```
            task.DateLastModified = task.DateCreated;
            task.Description = "Do the laundry";
            task.DueDate = DateTime.Now.AddDays(3);

            TaskAcknowledgement ack = proxy.AddTask(task);

            int taskNum = ack.TaskNumber;
            Console.WriteLine(
                "Task number {0} added to service",
                taskNum);

            // Now try to mark that same task
            // as completed:
            proxy.MarkTaskCompleted(taskNum);
        }

        catch (FaultException<FaultInfo> fault)
        {
            Console.WriteLine("Fault: {0}",fault.Detail.Reason);
        }
    }
```

NOTE Usage of faults as part of the Service contract

This exercise's version of a consumer uses a *FaultException<FaultInfo>* exception, based on the Service contract's specification that the *MarkTaskCompleted* operation could issue a fault of type *FaultInfo*. This code shows how to access the *FaultInfo* object and its *Reason* property.

7. Build the solution.
8. Making sure the service is still running and that VSProxy is the startup project, run this Console project.

   You should see that it successfully submits a task to the service. You might also try modifying the code in step 6 to force an exception. This can be done easily by changing the call to *MarkTaskCompleted* to take a number for which you know there isn't a valid task, for instance, 111. Finally, you can shut down the Task Manager service.

▶ Exercise 4 Consume a Service Using a Duplex, or Callback, Channel

1. Chapter 1, in the section titled "Duplex," discussed a simple Hello World Service contract and service type that used a Duplex MEP by setting up a callback contract. In this exercise, you will take that as a starting point and build on it to practice going through the steps required to create a proxy that can consume the Greeting service by communicating with it using a two-way duplex (or callback) channel. Following is the code that defines the Service contract, the Callback contract, and the service type in which the Callback contract is accessed and called. You can find this code in the Services (.cs or .vb as appropriate) file when you open the solution for this lab.

```
' VB
Imports System.ServiceModel
Imports System.ServiceModel.Channels

<ServiceContract()> _
Public Interface IGreetingHandler
    <OperationContract(IsOneWay:=True)> _
    Sub GreetingProduced(ByVal greeting As String)
End Interface

<ServiceContract(CallbackContract:= _
                 GetType(IGreetingHandler))> _
Public Interface IGreetingService
    <OperationContract(IsOneWay:=True)> _
    Sub RequestGreeting(ByVal name As String)
End Interface

<ServiceBehavior(InstanceContextMode := _
        InstanceContextMode.PerSession)> _
Public Class GreetingService
    Implements IGreetingService

    Public Sub RequestGreeting(ByVal name As String) _
            Implements IGreetingService.RequestGreeting
        Console.WriteLine("In GreetingService.RequestGreeting")
        Dim callbackHandler As IGreetingHandler
        callbackHandler = _
            OperationContext.Current.GetCallbackChannel( _
                Of IGreetingHandler)()
        callbackHandler.GreetingProduced("Hello " + name)
    End Sub
End Class

// C#
using System.ServiceModel;
using System.ServiceModel.Channels;

[ServiceContract]
interface IGreetingHandler
{
    [OperationContract(IsOneWay = true)]
    void GreetingProduced(string greeting);
}

[ServiceContract(CallbackContract =
    typeof(IGreetingHandler))]
interface IGreetingService
{
    [OperationContract(IsOneWay = true)]
    void RequestGreeting(string name);
}
```

```
[ServiceBehavior(InstanceContextMode =
        InstanceContextMode.PerSession)]
class GreetingService : IGreetingService
{
    public void RequestGreeting(string name)
    {
        Console.WriteLine("In GreetingService.RequestGreeting");
        IGreetingHandler callbackHandler =
            OperationContext.Current.GetCallbackChannel<IGreetingHandler>();
        callbackHandler.GreetingProduced("Hello " + name);
    }
}
```

In this exercise, you not only create a proxy, but you also use the concept of an agent discussed in this lesson to create a simple agent that wraps the proxy and takes care of the details required both to set up the callback channel and implement the Callback contract. Finally, this lab also uses the technique of manually defining a proxy class to a service.

2. Navigate to the <*InstallHome*>/Chapter4/Lesson1/Exercise4/<*language*>/Before directory and double-click the Exercise4.sln file to open the solution in Visual Studio.

   The solution consists of only one project, a Console project in which you define both the client and the service, and you configure the endpoints in code rather than in a configuration file. You wouldn't do this in a production setting, but it is good to see how it's done to simplify code you might use to experiment with WCF. In this case, the focus is on the mechanics of creating a callback proxy, and everything else is simplified.

3. In the Program (.cs or .vb as appropriate) file, manually define a *GreetingServiceProxy* class (above the *Program* class) that you'll use to act as the proxy to the service. The class should be as follows:

```
' VB
Public Class GreetingServiceProxy
    Inherits DuplexClientBase(Of IGreetingService)
    Implements IGreetingService

    Public Sub New(ByVal inputInstance As InstanceContext)
        MyBase.New(inputInstance, New NetTcpBinding(), _
            New EndpointAddress("net.tcp://localhost:6789/service"))
    End Sub

    Public Sub RequestGreeting(ByVal name As String) _
            Implements IGreetingService.RequestGreeting
        Me.Channel.RequestGreeting(name)
    End Sub
End Class

// C#
class GreetingServiceProxy : DuplexClientBase<IGreetingService>,
                             IGreetingService
{
    public GreetingServiceProxy(InstanceContext inputInstance)
        : base(inputInstance, new NetTcpBinding(),
            new EndpointAddress("net.tcp://localhost:6789/service"))
    {
    }

    public void RequestGreeting(string name)
    {
        this.Channel.RequestGreeting(name);
    }
}
```

4. In the same file, below the proxy class, define a *GreetingServiceAgent* class that wraps an instance of the proxy class you just defined. This agent class also sets up the instancing context for the proxy in its constructor, and it implements the Callback contract.

   The class should be as follows:

```
' VB
Public Class GreetingServiceAgent
    Implements IGreetingService, IGreetingHandler, IDisposable

    Public Sub New()
        Try
            ' Set up instance context and pass it to proxy:
            Dim context As New InstanceContext(Me)
            _proxy = New GreetingServiceProxy(context)
            _proxy.Open()
        Catch ex As Exception
            _proxy = Nothing
        End Try
    End Sub

    Public Sub RequestGreeting(ByVal name As String) _
            Implements IGreetingService.RequestGreeting
        If Not _proxy Is Nothing Then
            _proxy.RequestGreeting(name)
        End If
    End Sub

    Public Sub GreetingProduced(ByVal greeting As String) _
            Implements IGreetingHandler.GreetingProduced
        Console.WriteLine( _
            "Called back with greeting: {0}", greeting)
    End Sub

    Public Sub Dispose() Implements IDisposable.Dispose
        If Not _proxy Is Nothing Then
            _proxy.Close()
        End If
    End Sub

    Private _proxy As GreetingServiceProxy
```

```
End Class

// C#
class GreetingServiceAgent : IGreetingService,
                             IGreetingHandler, IDisposable
{
    public GreetingServiceAgent()
    {
        try
        {
            // Set up instance context and pass it to proxy:
            InstanceContext context = new InstanceContext(this);
            _proxy = new GreetingServiceProxy(context);
            _proxy.Open();
        }

        catch
        {
            _proxy = null;
        }
    }

    public void Dispose()
    {
        if (_proxy != null)
            _proxy.Close();
    }

    public void RequestGreeting(string name)
    {
        if (_proxy != null)
            _proxy.RequestGreeting(name);
    }

    public void GreetingProduced(string greeting)
    {
        Console.WriteLine(
               "Called back with greeting: {0}", greeting);
    }

    private GreetingServiceProxy _proxy;
}
```

5. In the same file, in the *try* block of the *Main* method of the *Program* class, write the following code to call the agent:

```
' VB
Dim agent As New GreetingServiceAgent
agent.RequestGreeting("Sally")

// C#
GreetingServiceAgent agent = new GreetingServiceAgent();
agent.RequestGreeting("Sally");
```

For the sake of completeness, here is the Program class listing in its entirety:

```
' VB
Public Class Program

    Public Shared Sub Main()
        Dim t As Thread
        t = New Thread(New ThreadStart( _
                AddressOf Program.RunService))
        t.Start()
        autoEvent.WaitOne()

        Try
            Dim agent As New GreetingServiceAgent
            agent.RequestGreeting("Sally")
        Catch ex As Exception
            Console.WriteLine("Error: {0}", ex.Message)
        End Try

    End Sub

    Public Shared Sub RunService()
        Dim host As ServiceHost
        host = New ServiceHost(GetType(GreetingService))

        host.AddServiceEndpoint(GetType(IGreetingService), _
                New NetTcpBinding(), "net.tcp://localhost:6789/service")

        host.Open()
        autoEvent.Set()

        Console.WriteLine("Press Enter to exit")
        Console.ReadLine()
    End Sub

    Public Shared autoEvent As AutoResetEvent = New AutoResetEvent(False)
End Class

// C#
class Program
{
    static void Main(string[] args)
    {
        new Thread(new ThreadStart(RunService)).Start();
        autoEvent.WaitOne();

        try
        {
            GreetingServiceAgent agent = new GreetingServiceAgent();
            agent.RequestGreeting("Sally");
        }

        catch (Exception ex)
```

```
            {
                Console.WriteLine("Error: {0}", ex.Message);
            }
        }

        static void RunService()
        {
            ServiceHost host = new ServiceHost(typeof(GreetingService));
            host.AddServiceEndpoint(typeof(IGreetingService),
                new NetTcpBinding(), "net.tcp://localhost:6789/service");
            host.Open();
            autoEvent.Set();
            Console.WriteLine("Press Enter to exit");
            Console.ReadLine();
        }

        static AutoResetEvent autoEvent = new AutoResetEvent(false);
    }
```

6. Build and run the application.

   You should see that both the service and the callback object, which in this case is your agent instance, are successfully called.

Lesson Summary

- Both the command-line utility svcutil and Visual Studio can be used to generate proxy classes from a service's metadata, whose instances can be used as proxies to a service.
- Proxy classes can be manually defined by inheriting from the *ClientBase* class or from the *DuplexClientBase* class for proxies that need a callback channel.
- Proxy objects can be generated dynamically using the *ChannelFactory* class based only on the Service contract.
- Method calls on proxy objects are translated into messages sent to a remote service by the WCF client-side plumbing either synchronously or asynchronously. Asynchronous method calls use the *Begin-* and *End-* method pairs in tandem.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Consuming WCF Services." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. Suppose you have the following Service contract and associated Callback contract:

```
' VB
<ServiceContract()> _
Public Interface IRetrieveHandler
    <OperationContract(IsOneWay:=True)> _
    Sub HandleFileRetrieved( _
            ByVal fileName As String, ByVal data As Stream)
End Interface

<ServiceContract( _
    CallbackContract:=GetType(IRetrieveHandler))> _
Public Interface IStorageArchive
    <OperationContract(IsOneWay:=True)> _
    Sub RequestFileRetrieve(ByVal fileName As String)
End Interface

// C#
[ServiceContract()]
public interface IRetrieveHandler
{
    [OperationContract(IsOneWay=true)]
    void HandleFileRetrieved(
            string fileName, Stream data);
}

[ServiceContract(
    CallbackContract=typeof(IRetrieveHandler))]
public interface IStorageArchive
{
    [OperationContract(IsOneWay = true)]
    void RequestFileRetrieve(string fileName);
}
```

Suppose further that you want to define a proxy class manually that can be instantiated and used to communicate with this service. Which of the following is the correct definition for the proxy class?

A.
```
' VB
Public Class StorageArchiveProxy
    Inherits ClientBase(Of IStorageArchive)
    Implements IStorageArchive

    Public Sub New(ByVal instanceContext As InstanceContext, _
                   ByVal binding As Binding, _
                   ByVal epAddr As EndpointAddress)
        MyBase.New(instanceContext, binding, epAddr)
    End Sub

    Public Sub New(ByVal instanceContext As InstanceContext, _
                   ByVal endpointConfigurationName As String)
        MyBase.New(instanceContext, endpointConfigurationName)
    End Sub
    Public Sub RequestFileRetrieve( _
            ByVal fileName As String) _
            Implements IStorageArchive.RequestFileRetrieve
        Me.Channel.RequestFileRetrieve(fileName)
    End Sub
End Class

// C#
public class StorageArchiveProxy :
        ClientBase<IStorageArchive>, IStorageArchive
{
    public StorageArchiveProxy(
            InstanceContext instanceContext,
            Binding binding, EndpointAddress epAddr)
        : base(instanceContext, binding, epAddr)
    {
    }

    public StorageArchiveProxy(
            InstanceContext instanceContext,
            string endpointConfigurationName)
        : base(instanceContext, endpointConfigurationName)
    {
    }

    public void RequestFileRetrieve(string fileName)
    {
        this.Channel.RequestFileRetrieve(fileName);
    }
}
```

B.
```
' VB
Public Class StorageArchiveProxy
    Inherits DuplexClientBase(Of IStorageArchive)
    Implements IStorageArchive

    Public Sub New(ByVal instanceContext As InstanceContext, _
                   ByVal binding As Binding, _
                   ByVal epAddr As EndpointAddress)
        MyBase.New(instanceContext, binding, epAddr)
    End Sub

    Public Sub New(ByVal instanceContext As InstanceContext, _
                   ByVal endpointConfigurationName As String)
        MyBase.New(instanceContext, endpointConfigurationName)
    End Sub

    Public Sub RequestFileRetrieve( _
            ByVal fileName As String) _
            Implements IStorageArchive.RequestFileRetrieve
        Me.Channel.RequestFileRetrieve(fileName)
    End Sub
End Class
```

```
// C#
public class StorageArchiveProxy :
        DuplexClientBase<IStorageArchive>, IStorageArchive
{
    public StorageArchiveProxy(
            InstanceContext instanceContext,
                Binding binding, EndpointAddress epAddr)
        : base(instanceContext,binding, epAddr)
    {
    }

    public StorageArchiveProxy(
            InstanceContext instanceContext,
                string endpointConfigurationName)
        : base(instanceContext,endpointConfigurationName)
    {
    }

    public void RequestFileRetrieve(string fileName)
    {
        this.Channel.RequestFileRetrieve(fileName);
    }
}
```

C.
```
' VB
Public Class StorageArchiveProxy
    Inherits DuplexClientBase(Of IStorageArchive)
    Implements IStorageArchive

    Public Sub New(ByVal binding As Binding, _
                   ByVal epAddr As EndpointAddress)
        MyBase.New(binding, epAddr)
    End Sub

    Public Sub New(ByVal endpointConfigurationName As String)
        MyBase.New(endpointConfigurationName)
    End Sub

    Public Sub RequestFileRetrieve( _
            ByVal fileName As String) _
            Implements IStorageArchive.RequestFileRetrieve
        Me.Channel.RequestFileRetrieve(fileName)
    End Sub
End Class

// C#
public class StorageArchiveProxy :
        DuplexClientBase<IStorageArchive>, IStorageArchive
{
    public StorageArchiveProxy(
            Binding binding, EndpointAddress epAddr)
        : base(binding, epAddr)
    {
    }

    public StorageArchiveProxy(
            string endpointConfigurationName)
        : base(endpointConfigurationName)
    {
    }

    public void RequestFileRetrieve(string fileName)
    {
        this.Channel.RequestFileRetrieve(fileName);
    }
}
```

D.
```
' VB
Public Class StorageArchiveProxy
    Inherits ClientBase(Of IStorageArchive)
    Implements IStorageArchive

    Public Sub New(ByVal binding As Binding, _
                   ByVal epAddr As EndpointAddress)
        MyBase.New(binding, epAddr)
    End Sub

    Public Sub New(ByVal endpointConfigurationName As String)
        MyBase.New(endpointConfigurationName)
    End Sub

    Public Sub RequestFileRetrieve( _
            ByVal fileName As String) _
            Implements IStorageArchive.RequestFileRetrieve
        Me.Channel.RequestFileRetrieve(fileName)
    End Sub
End Class

// C#
public class StorageArchiveProxy :
        ClientBase<IStorageArchive>, IStorageArchive
{
    public StorageArchiveProxy(
            Binding binding, EndpointAddress epAddr)
        : base(binding, epAddr)
    {
    }

    public StorageArchiveProxy(
            string endpointConfigurationName)
        : base(endpointConfigurationName)
    {
    }

    public void RequestFileRetrieve(string fileName)
    {
        this.Channel.RequestFileRetrieve(fileName);
    }
}
```

2. Suppose you have the following Service contract:

```
' VB
<ServiceContract()> _
Public Interface IOrderEntryService
    <OperationContract()> _
    Function SubmitOrder(ByVal order As Order) _
            As OrderAcknowledgement

    ' Etc...
End Interface

// C#
[ServiceContract()]
public interface IOrderEntryService
{
    [OperationContract()]
    OrderAcknowledgement SumbitOrder(Order order);

    // Etc...
}
```

Suppose further that you have generated a proxy class for this service that is equipped with the asynchronous *Begin-* and *End-* method pair needed to invoke the *SubmitOrder* operation asynchronously. Several steps need to be taken to successfully invoke this operation asynchronously and be called back when the operation completes. Which of the following steps to achieve this goal is incorrectly implemented?

A. Define a handler method that will be called back when the asynchronously invoked operation completes, such as:

```
' VB
Public Shared Sub HandleOrderSubmitted( _
        ByVal cb As AsyncCallback)
    ' Etc...
End Sub

// C#
static void HandleOrderSubmitted(AsyncCallback cb)
{
    // Etc...
}
```

B. Asynchronously invoke the operation by calling the *Begin-* method as shown here:

```
' VB
Dim proxy As OrderEntryServiceClient
proxy = New OrderEntryServiceClient()

Dim cb As AsyncCallback
cb = New AsyncCallback(AddressOf HandleOrderSubmitted)
```

```
proxy.BeginSubmitOrder(order, cb, proxy)

// C#
OrderEntryServiceClient proxy = new OrderEntryServiceClient();
AsyncCallback cb = new AsyncCallback(HandleOrderSubmitted);
proxy.BeginSumbitOrder(order, cb, proxy);
```

C. Access the proxy object from the *AsyncState* property when the callback handler has been called:

```
' VB
Dim proxy As OrderEntryServiceClient
proxy = CType(result.AsyncState, OrderEntryServiceClient)

// C#
OrderEntryServiceClient proxy =
    result.AsyncState as OrderEntryServiceClient;
```

D. Having already accessed the proxy object in the callback handler, use it to end the call:

```
' VB
Dim ack As OrderAcknowledgement
ack = proxy.EndSubmitOrder(result)

// C#
OrderAcknowledgement ack = proxy.EndSumbitOrder(result);
```

Lesson 2: Consuming Non-WCF Services

In this lesson, the focus is still on the consumption of services but shifts away from the consumption of WCF services to look at what you need to know to use WCF effectively to consume non-WCF services, those built on other technology platforms. You'll begin by looking at what the Web services industry has defined as the minimal base of interoperability that all platforms should support and how WCF supports that. From there, you'll look into the three most common WS-* specifications that come into play when you are trying to achieve interoperability in a scenario in which the services involved go beyond that minimal base of interoperability.

After this lesson, you will be able to:

- Use the svcutil command-line tool or Visual Studio to generate proxies from the WSDL of a non-WCF service.
- Use the proxy to a non-WCF service to call operations on a non-WCF service.
- Use *BasicHttpBinding* to ensure that your WCF service is WS-I Basic Profile–compliant when exchanging messages with a non-WCF service.

Estimated lesson time: 40 minutes

Creating Proxies for Non-WCF Services

The only means available for accessing the metadata for a non-WCF service is through the standard WSDL. The two mechanisms to create WCF proxies you explored in Lesson 1 of this chapter that depend on access to an existing WCF Service contract (dynamically creating proxies using the *ChannelFactory* class and manually coding proxy classes) do not apply here. Instead, you must either use the svcutil command-line tool or add a service reference in Visual Studio to point at the WSDL of the non-WCF service you want to consume using WCF. In the lab following this lesson, you use the svcutil approach once again, this time to consume a non-WCF service.

Interoperability Through WS-I Basic Profile Support

What the software industry calls Web services is organic in the sense that the standards that comprise all the parts of XML-based Web services (XML, Hypertext Transfer Protocol [HTTP], SOAP, WSDL, XML schema definitions [XSD], WS-Addressing, to name a few) are constantly evolving. Not only are they evolving, but there is enough room for interpretation among the various standards that two Web services platform vendors could technically support all the standards but have trouble interoperating because the two vendors interpret the standards differently in important ways.

The classic example of this sort of differing interpretations comes from SOAP and the choices around *Rpc/Encoded* vs. *Document/Literal* mechanisms for structuring SOAP envelopes. (See Chapter 1 for more on these differences.) In the early days of SOAP usage, some vendors chose to use the *Rpc/Encoded* rules for formulating their SOAP messages whereas others used the SOAP *Document/Literal* rules to formulate their SOAP messages. Both were SOAP-compliant, but they could not interoperate because they were expecting their SOAP messages to be constructed differently. Enter the Web Services Interoperability Organization (WS-I).

WS-I

On its Web site at *http://www.ws-i.org/Default.aspx*, WS-I is defined as "an open industry organization chartered to establish Best Practices for Web services interoperability, for selected groups of Web services standards, across platforms, operating systems and programming languages." The approach WS-I takes is to help the Web services community by providing guidance, recommended practices, and supportive resources around the usage of existing Web services standards to promote interoperability. WS-I does not itself define any new standards; rather, its mandate is to guide the Web services community in using the existing standards to achieve interoperability.

WS-I Basic Profile

WS-I defines its Basic Profile as "...a set of non-proprietary Web services specifications, along with clarifications, refinements, interpretations and amplifications of those specifications which promote interoperability." (See *http://www.ws-i.org/deliverables/workinggroup.aspx?wg=basicprofile.*) The Basic Profile acts as a guide to the consistent usage of the foundational specifications that, taken together, form the core of Web services, namely:

- SOAP 1.1
- WSDL 1.1
- Universal Description, Discovery, and Integration (UDDI) 2.0
- XML 1.0 (Second Edition)
- XML Schema Part 1: Structures
- XML Schema Part 2: Data Types
- RFC 2246: The TLS (Transport Layer Security) Protocol Version 1.0
- RFC 2459: Internet X.509 Public Key Infrastructure Certificate and CRL Profile
- RFC 2616: Hypertext Transfer Protocol—HTTP/1.1
- RFC 2818: HTTP over TLS
- RFC 2965: HTTP State Management Mechanism
- The Secure Sockets Layer (SSL) Protocol Version 3.0

MORE INFO The WS-I Basic Profile

You can find the latest version of the full profile at *http://www.ws-i.org/Profiles/BasicProfile-1_2(WGAD).html*. Another version that is targeted at version 1.2 of SOAP is in the works; you can find that at *http://www.ws-i.org/Profiles/BasicProfile-2_0(WGD).html*.

WCF Basic Profile Support

WCF provides support for WS-I Basic Profile 1.1 through the *BasicHttpBinding* class. Writing interoperable WCF services that are WS-I Basic Profile–compliant is as easy as using this binding, either programmatically or through a configuration file setting. Use the *BasicHttpBinding* class programmatically as shown here:

```
' VB
Dim host As ServiceHost
host = New ServiceHost(GetType(OrderEntryService))
host.AddServiceEndpoint(GetType(IOrderEntryService), _
    New BasicHttpBinding(), "http://localhost:8080/orders/")
host.Open()

// C#
ServiceHost host = new ServiceHost(typeof(OrderEntryService));
host.AddServiceEndpoint(typeof(IOrderEntryService),
    new BasicHttpBinding(), "http://localhost:8080/orders/");
host.Open();
```

Use the *BasicHttpBinding* class through a configuration file setting as shown here:

```
<service name="Orders.OrderEntryService">
    <host>
        <baseAddresses>
            <add baseAddress="http://localhost:8080/orders/"/>
        </baseAddresses>
    </host>

    <endpoint address="OrderEntryService"
              binding="basicHttpBinding"
              contract="Orders.IOrderEntryService"
              name="OrderEntryServiceHttpEndpoint" />
</service>
```

One scenario in particular that comes up frequently is when WCF services need to consume existing Web services built on ASP.NET, the Microsoft Web services platform prior to WCF. In the lab following this lesson, you step through an example that uses WCF to consume the MapPoint Web service, an ASP.NET Web service Microsoft hosts that provides mapping services.

The Importance of Documentation

In theory, the WSDL that a service emits should contain enough service metadata for a Web services platform to generate proxies that are fully equipped to communicate with the service. In practice, however, there are gaps that, typically, some level of documentation is required to fill. Someday, these gaps might close, but today, they usually exist because there is a gap in the standards supported by one of the technology platforms, either on the service-provider side or on the consumer side.

As an example, look at authentication in the context of the MapPoint service you consume in the lab following this lesson. This service uses HTTP digest authentication, but the WSDL itself does not contain any policy stating this. Therefore, when you generate a WCF proxy class and configuration file to consume the service, it will fail unless you modify the configuration file to use this form of authentication.

The lab for this lesson details how to update your application to match the documentation but, for now, it's sufficient just to understand that this situation arises simply because the MapPoint service is built on the ASP.NET ASMX Web services technology (referred to as ASMX because of the .asmx extension to distinguish it from ASP.NET Web applications) and does not have any support for WS-Policy, something that is fully supported in WCF. Thus, although a WCF service would be able to emit policy information in its metadata exchange to specify its usage of this authentication mechanism, the ASMX-based MapPoint service could not. WCF consumers can still interoperate with this service, using WCF *BasicHttpBinding* (to specify Basic Profile support), and the MapPoint service documentation tells consumers how the service handles authentication.

Interoperability by Extended WS-* Support

The WS-I Basic Profile covers only the bare minimum of interoperability support, but what happens when you go beyond that and enter the realm of WS-* (WS-Security, WS-ReliableMessaging, and so on)? WCF has strong support for many of these specifications, but what are your chances for interoperability after you start using these extended Web services standards?

Unfortunately, the answer is that, at this point in time, each case has to be looked at individually. WS-I has defined additional profiles that will help, such as the Basic Security Profile (*http://www.ws-i.org/Profiles/BasicSecurityProfile-1.0.html*) and the Reliable Secure Profile (*http://www.ws-i.org/profiles/rsp-scenarios-1.0.pdf*), but in reality, the support for many of the WS-* specifications is mixed across the various Web services platforms.

Later chapters discuss WCF support for the broader WS-* specifications (the specifications that extend beyond what the Basic Profile offers and move into the realm of the qualities of service under which a service is capable of executing). For now, understand that WCF has very strong standards-based support for interoperability in the face of the following quality-of-service challenges:

- **The efficient transfer of large binary data** WCF supports the Message Transmission Optimization Mechanism (MTOM). MTOM allows messages that might contain large binary data to be sent across the wire as multi-part MIME (Multipurpose Internet Mail Extensions) messages in which the first part is the XML SOAP envelope that contains references to the binary parts that follow it and the binary data that is conceptually part of the message. This approach avoids the space and processing overhead inherent in having to encode the binary data in Base64, which is necessary if the binary chunk is simply placed inside the XML InfoSet.
- **Secure transmission of messages** WCF has very strong support for WS-Security and the various other specifications related to WS-Security (for instance, WS-SecureConversation, WS-Trust, and so on). Chapter 7, "Infrastructure Security," and Chapter 8, "User-Level Security," cover WCF security.
- **The reliable exchange of messages** WS-ReliableMessaging (WS-RM) is a standard that defines the means through which Web services platforms can provide interoperable delivery assurances at the message level. Beyond the packet-level assurances of delivery that the given transport might make available, WS-RM provides assurances that messages sent will be delivered. WCF currently supports the Exactly Once assurance—that a message sent will be delivered exactly once—with the optional feature that a given sequence of messages can be guaranteed to be delivered in order.

MORE INFO Key WS-* specifications

For more about MTOM, visit *http://www.w3.org/TR/soap12-mtom/*.

For more about WS-Reliable Messaging, visit *http://specs.xmlsoap.org/ws/2005/02/rm/ws-reliablemessaging.pdf*.

Finally, for more about WS-Security, visit *http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss*.

So if the service uses quality-of-service mechanisms outside the scope of the WS-I Basic Profile, WCF can consume the service as long as the service provider platform implements the mechanism in compliance with the relevant standard.

Quick Check

- You need to consume an ASP.NET Web service that uses HTTP NTLM authentication. However, its WSDL has no indication that HTTP NTLM authentication is its authentication policy. Does this mean that the WCF proxy that is generated will be unable to consume this service?

Quick Check Answer

- No. As long as this policy is well documented and, therefore, known to the developers, the WCF proxy will still be able to consume the service if the developer appropriately changes the *clientCredentialType* attribute on the *transport* element in the *security* section of the configuration file. You would need to change the attribute, which would have defaulted to *clientCredentialType="None"*, to *clientCredentialType="Ntlm"* and ensure that the right credentials are created and assigned to the proxy.

Lab: Consuming a Non-WCF Mapping Service

In this lab, you will work through the details of consuming a very powerful Internet-facing mapping service, namely, the MapPoint service. It is built on the previous Microsoft Web services technology framework, ASP.NET ASMX, so it qualifies as a non-WCF service. It is also a good example of when the lack of support for WS-Policy requires the consumer to delve a little deeper into the service documentation to know how it handles authentication. This is in contrast to a WCF service which, with its full support for WS-Policy, would be able to notify consumers of its authentication policy through the service metadata it emits. In the first exercise, you focus on using the MapPoint service and then, in the second exercise, you improve the way you use it by retrieving the map images asynchronously because that retrieval can be time-consuming and, therefore, might take longer than you want to wait in a synchronous invocation setting.

▶ Exercise 1 Consume the MapPoint Service

In this exercise, you will go through the steps to consume the MapPoint service, owned and operated by Microsoft, a service that provides rich mapping capabilities to its consumers. It is the service behind the Microsoft Virtual Earth platform.

Virtual Earth resources

See *http://www.microsoft.com/virtualearth/* for more about the Virtual Earth platform in general. Note that if you are consuming this service in .NET, the most likely scenario would be that you would use the full Virtual Earth software development kit (SDK) or even the reusable MapControl. For more information about Virtual Earth resources for developers, see *http://dev.live.com/virtualearth/*.

For the purposes of this lab, because the focus is on consuming non-WCF services, you consume the MapPoint service (which is an ASP.NET ASMX service) directly by creating a WCF proxy to communicate with it. In addition to creating the proxy, you also use the technique, described in Lesson 1, of defining an agent that wraps the proxy to make using the service easier for application developers.

Before you can begin going through the required steps, you must have a developer account to access this service. The first steps of Exercise 1 walk you through getting an account if you don't already have one.

1. To access the MapPoint Web service, you must have a Windows Live ID. If you don't already have one, you can get one at *http://get.live.com/getlive/overview?wa=wsignin1.0*.

   When you have a Live ID and are signed in with it, you must ensure that you have a developer's ID and password that is specific to MapPoint. If you don't already have that, you can request it at *https://mappoint-css.live.com/mwssignup/*.

   After you go through this process, which includes receiving some confirmation e-mail messages, you should have an ID and password that provides you with developer access to the MapPoint Web service.

   MORE INFO MapPoint Web service documentation

   The detailed developer documentation for the MapPoint Web service can be found at *http://msdn.microsoft.com/en-us/library/bb507684.aspx*.

2. Navigate to the <*InstallHome*>/Chapter4/Lesson2/Exercise1/<*language*>/Before directory and double-click the Exercise1.sln file to open the solution in Visual Studio.

   The solution consists of two projects:

3. Open a Visual Studio command prompt to the directory containing the MapPoint-ServiceAgent project, which in this case is the <*InstallHome*>/Chapter4/Lesson2/Exercise1/<*language*>/Before/Microsoft.MapPoint.ServiceAgent directory.

4. Execute the following command at the command prompt to generate a proxy to the MapPoint service whose WSDL is available at *http://staging.mappoint.net/standard-30/mappoint.wsdl*. (Enter the following as a single command; it is formatted here on multiple lines to fit on the printed page.)

```
' VB
svcutil /l:VB /async /config:app.config
/namespace:*,Proxy /out:MapPointProxy.vb
http://staging.mappoint.net/standard-30/mappoint.wsdl

// C#
svcutil /async /config:app.config
/namespace:*,Microsoft.MapPoint.Proxy /out:MapPointProxy.cs
http://staging.mappoint.net/standard-30/mappoint.wsdl
```

5. Add the MapPointProxy (.cs or .vb as appropriate) file that was just created in the previous step by the svcutil tool to the Microsoft.MapPoint.ServiceAgent project.

6. From the app.config file (in the Microsoft.MapPoint.ServiceAgent project) that was just populated by the svcutil tool, copy the *system.serviceModel* element and its children

(everything within the *configuration* element) to the app.config file in the MapPointTest-Client project, placing it directly below the *appSettings* element that already exists there.

7. Next, modify the values in the *appSettings* section of this same configuration file so that the *MapPointWebServiceID* and *MapPointWebServicePassword* key-value pairs appropriately reflect your Virtual Earth Platform developer account ID and password (which you acquired to access the MapPoint service in step 2).
8. As discussed in the lesson, from the documentation (as opposed to a WS-Policy element in the service's metadata) you learn that this service authenticates using HTTP digest authentication, so you must alter the app.config file to reflect this. To do so, in each of the four binding elements in the configuration file, change the security from:

```
<security mode="None">
    <transport clientCredentialType="None"
        proxyCredentialType="None"
        realm="" />
    <message clientCredentialType="UserName"
        algorithmSuite="Default" />
</security>
```

   to the following:

```
<security mode="TransportCredentialOnly">
    <transport clientCredentialType="Digest"
        proxyCredentialType="None"
        realm="" />
    <message clientCredentialType="UserName"
        algorithmSuite="Default" />
</security>
```

9. To the Microsoft.MapPoint.ServiceAgent project, add a new class file named **MapPoint-ServiceAgent** (.cs or .vb as appropriate), which you'll now use to define an agent that wraps some logic around the autogenerated proxy and, therefore, makes it a little easier to work with the service.

   Normally, there would be more methods than this, but for the purposes here, you'll keep it simple and add only a few methods that facilitate retrieving map images by addresses. The resulting file should be as follows:

```
' VB
Imports System.Configuration
Imports System.Security.Principal
Imports System.Net

Imports Microsoft.MapPoint.ServiceAgent.Proxy

Public Delegate Sub MapRetrievedHandler( _
        ByVal mapImage As MapImage, ByVal address As Address)

Public Delegate Function GetMapDelegate( _
        ByVal address As Address, ByVal mapHeight As Double, _
        ByVal mapWidth As Double, ByVal dataSourceName As String) _
            As MapImage

Public Class MapPointServiceAgent

    Public Function GetLocationByAddress( _
            ByVal address As Address, ByVal dataSourceName As String) _
                As Location
        Dim findSvcProxy As FindServiceSoapClient
        findSvcProxy = InitFindServiceProxy()

        Dim addrSpec As FindAddressSpecification
        addrSpec = New FindAddressSpecification()
        addrSpec.InputAddress = address
        addrSpec.DataSourceName = dataSourceName

        ' Note: CustomerInfoFindHeader & UserInfoFindHeader
        ' can be null here since we are happy with defaults
        Dim results As FindResults = Nothing
        Try
            results = findSvcProxy.FindAddress(Nothing, Nothing, addrSpec)
        Catch
            results = Nothing
        End Try

        Dim res As Location = Nothing
        If Not results Is Nothing Then
            If results.NumberFound > 0 Then
                If Not results.Results(0).FoundLocation Is Nothing Then
                   res = results.Results(0).FoundLocation
                End If
            End If
        End If
        Return res
    End Function

    Public Function GetMapByLocation( _
                ByVal location As Location) As MapImage

        If location Is Nothing Then
            Return Nothing
        End If

        Dim renderSvcProxy As RenderServiceSoapClient
        renderSvcProxy = InitRenderServiceProxy()

        Dim mapSpec As New MapSpecification()
        mapSpec.DataSourceName = location.DataSourceName

        Dim views(0) As MapView
        views(0) = location.BestMapView.ByBoundingRectangle
        mapSpec.Views = views
        Dim mapImages() As MapImage
        Try
            mapImages = renderSvcProxy.GetMap(Nothing, Nothing, mapSpec)
        Catch
            mapImages = Nothing
        End Try

        Dim res As MapImage = Nothing
        If Not mapImages Is Nothing Then
            If mapImages.Length > 0 Then
                res = mapImages(0)
            End If
        End If

        Return res
    End Function


    Public Function GetMapByLocation( _
            ByVal location As Location, ByVal mapHeight As Double, _
            ByVal mapWidth As Double) As MapImage

        If location Is Nothing Then
            Return Nothing
        End If

        Dim renderSvcProxy As RenderServiceSoapClient
        renderSvcProxy = InitRenderServiceProxy()

        Dim mapSpec As New MapSpecification
        mapSpec.DataSourceName = location.DataSourceName

        ' Init view:
        Dim vbh As New ViewByHeightWidth()
        vbh.Height = mapHeight
        vbh.Width = mapWidth
        vbh.CenterPoint = location.LatLong

        Dim views(0) As MapView
        views(0) = vbh
        mapSpec.Views = views

        ' Init options:
        mapSpec.Options = New MapOptions()
        mapSpec.Options.Format = New ImageFormat()
        mapSpec.Options.Format.Height = Convert.ToInt32(mapHeight)
        mapSpec.Options.Format.Width = Convert.ToInt32(mapWidth)

        mapSpec.Options.Zoom = 0.001

        ' Init pushpin:
        Dim pin As New Pushpin()
        pin.IconDataSource = "MapPoint.Icons"
        pin.IconName = "1"
        pin.Label = location.Address.AddressLine
        pin.LatLong = location.LatLong

        Dim pins(0) As Pushpin
        pins(0) = pin
        mapSpec.Pushpins = pins

        Dim mapImages() As MapImage
        Try
            mapImages = renderSvcProxy.GetMap(Nothing, Nothing, mapSpec)
        Catch ex As Exception
            mapImages = Nothing
        End Try

        Dim res As MapImage = Nothing
        If Not mapImages Is Nothing Then
            If mapImages.Length > 0 Then
                res = mapImages(0)
            End If
        End If

        Return res
    End Function


    Public Function GetSizedMapByAddress( _
            ByVal address As Address, ByVal mapHeight As Double, _
            ByVal mapWidth As Double, ByVal dataSourceName As String) _
                As MapImage
        Dim location As Location
        location = GetLocationByAddress(address, dataSourceName)
        Return GetMapByLocation(location, mapHeight, mapWidth)
    End Function

    Public Function GetMapByAddress( _
        ByVal address As Address, ByVal dataSourceName As String) _
            As MapImage
        Dim location As Location
        location = GetLocationByAddress(address, dataSourceName)
        Return GetMapByLocation(location)
    End Function

    Public Shared Function AddressToString( _
                    ByVal address As Address) As String
        Return String.Format("{0}, {1}, {2}, {3}, {4}", _
                            address.AddressLine, _
                            address.PrimaryCity, _
                            address.Subdivision, _
                            address.PostalCode, _
                            address.CountryRegion)
    End Function

    Private Function InitFindServiceProxy() As FindServiceSoapClient
        Dim findSvcProxy As FindServiceSoapClient
        findSvcProxy = New FindServiceSoapClient()

        findSvcProxy.ClientCredentials.HttpDigest.ClientCredential= _
            New NetworkCredential( _
                ConfigurationManager.AppSettings(MapPointWebServiceIDKey), _
                ConfigurationManager.AppSettings(MapPointWebServicePasswordKey))

        findSvcProxy.ClientCredentials.HttpDigest.AllowedImpersonationLevel= _
            TokenImpersonationLevel.Impersonation

        Return findSvcProxy
    End Function

    Private Function InitRenderServiceProxy() As RenderServiceSoapClient
        Dim renderSvcProxy As RenderServiceSoapClient
        renderSvcProxy = New RenderServiceSoapClient()

        renderSvcProxy.ClientCredentials.HttpDigest.ClientCredential = _
            New NetworkCredential( _
                ConfigurationManager.AppSettings(MapPointWebServiceIDKey), _
                ConfigurationManager.AppSettings(MapPointWebServicePasswordKey))
        renderSvcProxy.ClientCredentials.HttpDigest.AllowedImpersonationLevel _
            = TokenImpersonationLevel.Impersonation

        Return renderSvcProxy
    End Function

    Private Const MapPointWebServiceIDKey As String = _
                        "MapPointWebServiceID"
    Private Const MapPointWebServicePasswordKey As String = _
                        "MapPointWebServicePassword"
End Class

Class MapRequestInfo
    Public invokedDelegate As GetMapDelegate
    Public mapRetrievedHandler As MapRetrievedHandler
    Public address As Address
End Class

// C#
using System.Configuration;
using System.Security.Principal;
using System.Net;

using Microsoft.MapPoint.Proxy;

namespace Microsoft.MapPoint.ServiceAgent
{
```

```
public delegate void MapRetrievedHandler(
    MapImage mapImage, Address address);

public delegate MapImage GetMapDelegate(Address address,
    double mapHeight, double mapWidth, string dataSourceName);

public class MapPointServiceAgent
{
    public Location GetLocationByAddress(
                    Address address, string dataSourceName)
    {
        FindServiceSoapClient findSvcProxy = InitFindServiceProxy();
        FindAddressSpecification addrSpec =
            new FindAddressSpecification();

        addrSpec.InputAddress = address;
        addrSpec.DataSourceName = dataSourceName;

        // Note: CustomerInfoFindHeader & UserInfoFindHeader can be null
        // here since we are happy with defaults
        FindResults results;
        try
        {
            results = findSvcProxy.FindAddress(null, null, addrSpec);
        }
        catch
        {
            results = null;
        }

        Location res = null;
        if (results != null && results.NumberFound > 0 &&
            results.Results[0].FoundLocation != null)
        {
            res = results.Results[0].FoundLocation;
        }

        return res;
    }

    public MapImage GetMapByLocation(Location location)
    {
        if (location == null)
            return null;

        RenderServiceSoapClient renderSvcProxy = InitRenderServiceProxy();

        MapSpecification mapSpec = new MapSpecification();
        mapSpec.DataSourceName = location.DataSourceName;
        mapSpec.Views = new MapView[]
            { location.BestMapView.ByBoundingRectangle };

        MapImage[] mapImages;
        try
        {
            mapImages = renderSvcProxy.GetMap(null, null, mapSpec);
        }
        catch
        {
            mapImages = null;
        }

        MapImage res = null;
        if (mapImages != null && mapImages.Length > 0)
            res = mapImages[0];

        return res;
    }

    public MapImage GetMapByLocation(
            Location location, double mapHeight, double mapWidth)
    {
        if (location == null)
            return null;

        RenderServiceSoapClient renderSvcProxy = InitRenderServiceProxy();

        MapSpecification mapSpec = new MapSpecification();
        mapSpec.DataSourceName = location.DataSourceName;

        // Init view:
        ViewByHeightWidth vbh = new ViewByHeightWidth();
        vbh.Height = mapHeight;
        vbh.Width = mapWidth;
        vbh.CenterPoint = location.LatLong;

        mapSpec.Views = new MapView[] { vbh };

        // Init options:
        mapSpec.Options = new MapOptions();

        mapSpec.Options.Format = new ImageFormat();
        mapSpec.Options.Format.Height = (int) mapHeight;
        mapSpec.Options.Format.Width = (int) mapWidth;

        mapSpec.Options.Zoom = 0.001;

        // Init pushpin:
        Pushpin pin = new Pushpin();
        pin.IconDataSource = "MapPoint.Icons";
        pin.IconName = "1";
        pin.Label = location.Address.AddressLine;
        pin.LatLong = location.LatLong;
        mapSpec.Pushpins = new Pushpin[] { pin };

        MapImage[] mapImages;
        try
        {
            mapImages = renderSvcProxy.GetMap(null, null, mapSpec);
        }
        catch
        {
            mapImages = null;
        }

        MapImage res = null;
        if (mapImages != null && mapImages.Length > 0)
            res = mapImages[0];

        return res;
    }

    public MapImage GetSizedMapByAddress(Address address,
            double mapHeight, double mapWidth, string dataSourceName)
    {
        Location location = GetLocationByAddress(
            address, dataSourceName);
        return GetMapByLocation(location,mapHeight,mapWidth);
    }

    public MapImage GetMapByAddress(
        Address address, string dataSourceName)
    {
        Location location = GetLocationByAddress(
            address, dataSourceName);
        return GetMapByLocation(location);
    }

    public static string AddressToString(Address address)
    {
        return string.Format(
                 "{0}, {1}, {2}, {3}, {4}",
                             address.AddressLine,
                             address.PrimaryCity,
                             address.Subdivision,
                             address.PostalCode,
                             address.CountryRegion);
    }

    private FindServiceSoapClient InitFindServiceProxy()
    {
        FindServiceSoapClient findSvcProxy = new FindServiceSoapClient();

        findSvcProxy.ClientCredentials.HttpDigest.ClientCredential =
            new NetworkCredential(
                ConfigurationManager.AppSettings[MapPointWebServiceIDKey],
                ConfigurationManager.AppSettings[
                    MapPointWebServicePasswordKey]);
            findSvcProxy.ClientCredentials.HttpDigest.AllowedImpersonationLevel =
                TokenImpersonationLevel.Impersonation;

            return findSvcProxy;
        }

        private RenderServiceSoapClient InitRenderServiceProxy()
        {
            RenderServiceSoapClient renderSvcProxy =
                                     new RenderServiceSoapClient();

            renderSvcProxy.ClientCredentials.HttpDigest.ClientCredential =
               new NetworkCredential(
                   ConfigurationManager.AppSettings[MapPointWebServiceIDKey],
                   ConfigurationManager.AppSettings[
                       MapPointWebServicePasswordKey]);

            renderSvcProxy.ClientCredentials.HttpDigest.AllowedImpersonationLevel =
                TokenImpersonationLevel.Impersonation;

            return renderSvcProxy;
        }

        private const string MapPointWebServiceIDKey =
                                     "MapPointWebServiceID";
        private const string MapPointWebServicePasswordKey =
                                     "MapPointWebServicePassword";
    }
}
```

NOTE More on the usage of HTTP digest authentication

To use the required HTTP digest authentication, in addition to the modifications you have already made to the configuration file, note that in both the *InitRenderServiceProxy* and the *InitFinderServiceProxy* methods, the *NetworkCredential* objects (created with the required MapPoint service credentials) must be assigned to the proxy's *ClientCredentials.HttpDigest.ClientCredential* property.

This completes the agent library, and it should now build. Next, you finish off the Windows Forms client that consumes the service.

10. Switching to the MapPointTestClient project, open the MainForm (.cs or .vb as appropriate) code file and add the following imports, noting that they are slightly different between the Visual Basic and C# versions because of the way the two languages handle default naming of proxy namespaces:

```
' VB
Imports Microsoft.MapPoint.ServiceAgent
Imports Microsoft.MapPoint.ServiceAgent.Proxy
```

```
// C#
using Microsoft.MapPoint.Proxy;
using Microsoft.MapPoint.ServiceAgent;
```

11. In the same MainForm file, add a private field to the *MainForm* class that is a reference to one of the *MapPointServiceAgent* objects whose class you just finished defining.

```
' VB
Private _svcAgent As MapPointServiceAgent = New MapPointServiceAgent()

// C#
private MapPointServiceAgent _svcAgent = new MapPointServiceAgent();
```

12. Implement the event handler *_btnViewMap_Click* that is invoked when the user clicks the View Map button. It should be as follows:

```
' VB
Private Sub _btnViewMap_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles _btnViewMap.Click
    Dim address As Address = New Address()

    address.AddressLine = _tboxAddressLine.Text
    address.PrimaryCity = _tboxCity.Text
    address.Subdivision = _tboxProvinceOrState.Text
    address.PostalCode = _tboxZipOrPostalCode.Text
    address.CountryRegion = CType( _
        _countriesComboBox.SelectedItem, String)

    Dim img As MapImage
    img = _svcAgent.GetSizedMapByAddress(address, _
        _mainPicBox.Height, _mainPicBox.Width, _
        DefaultDataSourceName)

    If Not img Is Nothing Then
        ' Display the map:
        Dim bmapImg As Bitmap
        bmapImg = New Bitmap( _
            New MemoryStream(img.MimeData.Bits))
        _mainPicBox.Image = bmapImg

        ' Cache the image:
        _cachedMaps.Add(address, bmapImg)

        ' And update combo box:
        _recentlyViewedComboBox.Items.Add(address)
        _recentlyViewedComboBox.SelectedIndex = _
            _recentlyViewedComboBox.Items.Count - 1
    Else
        MessageBox.Show("No map found for address:" + vbCrLf + _
            MapPointServiceAgent.AddressToString(address), _
            "Error")
    End If
    InitAddressInputFields()
End Sub

// C#
private void _btnViewMap_Click(object sender, EventArgs e)
{
    Address address = new Address();

    address.AddressLine = _tboxAddressLine.Text;
    address.PrimaryCity = _tboxCity.Text;
    address.Subdivision = _tboxProvinceOrState.Text;
    address.PostalCode = _tboxZipOrPostalCode.Text;
    address.CountryRegion = _countriesComboBox.SelectedItem as string;

    MapImage img = _svcAgent.GetSizedMapByAddress(
        address, _mainPicBox.Height,
        _mainPicBox.Width, DefaultDataSourceName);

    if (img != null)
    {
        // Display the map:
        Bitmap bmapImg = new Bitmap(
            new MemoryStream(img.MimeData.Bits));
        _mainPicBox.Image = bmapImg;

        // Cache the image:
        _cachedMaps.Add(address, bmapImg);

        // And update combo box:
        _recentlyViewedComboBox.Items.Add(address);
        _recentlyViewedComboBox.SelectedIndex =
            _recentlyViewedComboBox.Items.Count - 1;
    }
    else
    {
        MessageBox.Show(
            "No map found for address:\n" +
            MapPointServiceAgent.AddressToString(address),
            "Error");
    }

    InitAddressInputFields();
}
```

You can now build and run the application, making sure that the MapPointTestClient is the startup project and that you have an Internet connection so that the service is accessible. After it is running, you should be able to enter any valid address in the United States and Canada and display a map for that address. For example, try **1 Microsoft Way, Redmond, WA 98052, US**. You might even see the building in which WCF was built!

▶ Exercise 2 Consume the MapPoint Service Asynchronously

In this exercise, you will build on the result of Exercise 1 by improving the design so that the MapPoint service can be invoked asynchronously. This enhances the usability experience because the UI will now be responsive while the application retrieves map images asynchronously in the background.

To implement this, you must make two changes to the solution. In the UI code, you must add a status bar that provides a visual cue that some work is being done in the background. In the agent code, you'll add a method for invoking map retrieval asynchronously, with the method accepting a .NET delegate that can be used to call back the consumer when the retrieval is done. Note that to provide asynchronous invocation on your agent, you cannot simply map asynchronous calls on the agent to underlying asynchronous calls on the proxy. Why not? Because in defining your agent's interface, the choice made here was to wrap calls on the agent to two successive calls on the underlying proxy, so instead of trying to handle the coordination around two successive asynchronous calls on the proxy, you need to add the asynchronous invocation capability manually at the agent level and, in the separate thread that results, the two methods on the proxy can be called in a synchronous fashion.

1. Navigate to the <*InstallHome*>/Chapter4/Lesson2/Exercise2/<*language*>/Before directory and double-click the Exercise2.sln file to open the solution in Visual Studio.

   The solution consists of the two projects as they were completed in Exercise 1.

   Add two delegate declarations that will be used in this implementation. Add the following to the MapPointServiceAgent (.cs or .vb as appropriate) file, above the definition of the *MapPointServiceAgent* class:

```
' VB
Public Delegate Sub MapRetrievedHandler( _
    ByVal mapImage As MapImage, ByVal address As Address)

Public Delegate Function GetMapDelegate( _
    ByVal address As Address, ByVal mapHeight As Double, _
    ByVal mapWidth As Double, ByVal dataSourceName As String) _
        As MapImage

// C#
public delegate void MapRetrievedHandler(
    MapImage mapImage, Address address);

public delegate MapImage GetMapDelegate(Address address,
    double mapHeight, double mapWidth, string dataSourceName);
```

   Add a basic info class to store the state object you will pass when invoking a delegate asynchronously. To do so, define the following class below the *MapPointServiceAgent* class:

```
' VB
Class MapRequestInfo
    Public invokedDelegate As GetMapDelegate
    Public mapRetrievedHandler As MapRetrievedHandler
    Public address As Address
End Class

// C#
class MapRequestInfo
{
    public GetMapDelegate invokedDelegate;
    public MapRetrievedHandler mapRetrievedHandler;
    public Address address;
}
```

Add the following method to the *MapPointServiceAgent* class, which will be the internal callback method that is called when the asynchronously invoked operation completes.

```
' VB
Private Sub ProcessGetMapResult(ByVal ar As IAsyncResult)
    ' Access the state...
    Dim reqInfo As MapRequestInfo
    reqInfo = CType(ar.AsyncState, MapRequestInfo)
    Dim gmd As GetMapDelegate = reqInfo.invokedDelegate
    Dim mapRetrievedHandler As MapRetrievedHandler = _
        reqInfo.mapRetrievedHandler
    Dim addr As Address = reqInfo.address

    ' End the async call to get the returned image:
    Dim img As MapImage
    img = gmd.EndInvoke(ar)

    ' And use the address & the returned image to call back
    ' the handler interested in processing the retrieved map:
    If Not mapRetrievedHandler Is Nothing Then
        mapRetrievedHandler(img, addr)
    End If
End Sub

// C#
private void ProcessGetMapResult(IAsyncResult ar)
{
    // Access the state...
    MapRequestInfo reqInfo = ar.AsyncState as MapRequestInfo;
    GetMapDelegate gmd = reqInfo.invokedDelegate;
    MapRetrievedHandler mapRetrievedHandler =
                       reqInfo.mapRetrievedHandler;
    Address addr = reqInfo.address;

    // End the async call to get the returned image:
    MapImage img = gmd.EndInvoke(ar);

    // And use the address & the returned image to call back
    // the handler interested in processing the retrieved map:
    if (mapRetrievedHandler != null)
        mapRetrievedHandler(img,addr);
}
```

2. Provide clients of this *MapPointServiceAgent* class with a method to invoke a map retrieval request asynchronously, one that provides them with a means to provide a delegate that the implementation can use to call back the client when the map retrieval is complete. To do so, add the following publicly available method to the *MapPointService-Agent* class:

```
' VB
Public Sub BeginGetSizedMapByAddress( _
        ByVal address As Address, ByVal mapHeight As Double, _
        ByVal mapWidth As Double, ByVal dataSourceName As String, _
        ByVal mapRetrievedHandler As MapRetrievedHandler)
    ' Define the delegate to be invoked asynchronously:
    Dim gmd As GetMapDelegate
    gmd = New GetMapDelegate(AddressOf Me.GetSizedMapByAddress)

    ' Define the AsyncCallback delegate to be called when
    ' the asynchronous operation is completed:
    Dim cb As AsyncCallback
    cb = New AsyncCallback(AddressOf Me.ProcessGetMapResult)

    ' Create a "state" object:
    Dim reqInfo As MapRequestInfo = New MapRequestInfo()
    reqInfo.invokedDelegate = gmd
    reqInfo.mapRetrievedHandler = mapRetrievedHandler
    reqInfo.address = address

    ' Do asnyc invoke, passing our callback & state:
    gmd.BeginInvoke(address, mapHeight, mapWidth, _
                    dataSourceName, cb, reqInfo)
End Sub

// C#
public void BeginGetSizedMapByAddress(Address address,
        double mapHeight, double mapWidth,
        string dataSourceName,
        MapRetrievedHandler mapRetrievedHandler)
{
    // define the delegate to be invoked asynchronously:
    GetMapDelegate gmd = new GetMapDelegate(this.GetSizedMapByAddress);

    // Define the AsyncCallback delegate to be called when
    // the asynchronous operation is completed:
    AsyncCallback cb = new AsyncCallback(this.ProcessGetMapResult);

    // Create a "state" object:
    MapRequestInfo reqInfo = new MapRequestInfo();
    reqInfo.invokedDelegate = gmd;
    reqInfo.mapRetrievedHandler = mapRetrievedHandler;
    reqInfo.address = address;

    // Do asnyc invoke, passing our callback & state:
```

```
    gmd.BeginInvoke(address,mapHeight,mapWidth,
                    dataSourceName,cb,reqInfo);
}
```

This completes the improvements needed for the *MapPointServiceAgent*, and that project should now build. Next, you return to the UI and take advantage of these changes to make the UI more responsive as maps are asynchronously retrieved.

3. Open the design view of the *MainForm* and add a Status Strip control. Do this by simply dragging a Status Strip control from the toolbox onto the form. Rename it to **_statusStrip** and clear the Text property.
4. To this status strip, add a label control, renaming it to **_mainStatusLabel** and, again, clear the Text property.
5. Next, switch to code view for this form, which opens the MainForm (.cs or .vb as appropriate) file. To the *MainForm* class, add the following method that will be the callback handler, the method called when an asynchronously invoked map retrieval request has completed.

```
' VB
Private Sub HandleMapImageAvailableForDisplay( _
         ByVal img As MapImage, ByVal address As Address)
    If Not img Is Nothing Then
        _mainStatusLabel.Text = String.Format( _
            "Map for address {0} was retrieved", _
            MapPointServiceAgent.AddressToString(address))

        ' Display the map:
        Dim bmapImg As Bitmap
        bmapImg = New Bitmap( _
            New MemoryStream(img.MimeData.Bits))
        _mainPicBox.Image = bmapImg

        ' Cache the image:
        _cachedMaps.Add(address, bmapImg)

        ' And update combo box:
        _recentlyViewedComboBox.Items.Add(address)
        _recentlyViewedComboBox.SelectedIndex = _
            _recentlyViewedComboBox.Items.Count - 1
    Else
        _mainStatusLabel.Text = String.Format( _
            "There was an error retrieving map for address {0}", _
            MapPointServiceAgent.AddressToString(address))

        MessageBox.Show("No map found for address:" + vbCrLf + _
            MapPointServiceAgent.AddressToString(address), _
            "Error")
    End If
End Sub
```

```
// C#
private void HandleMapImageAvailableForDisplay(
        MapImage img, Address address)
{
    if (img != null)
    {
        _mainStatusLabel.Text = string.Format(
                "Map for address {0} was retrieved",
                MapPointServiceAgent.AddressToString(address));

        // Display the map:
        Bitmap bmapImg = new Bitmap(
                new MemoryStream(img.MimeData.Bits));
        _mainPicBox.Image = bmapImg;

        // Cache the image:
        _cachedMaps.Add(address, bmapImg);

        // And update combo box:
        _recentlyViewedComboBox.Items.Add(address);
        _recentlyViewedComboBox.SelectedIndex =
                        _recentlyViewedComboBox.Items.Count - 1;
    }
    else
    {
        _mainStatusLabel.Text = string.Format(
            "There was an error retrieving map for address {0}",
            MapPointServiceAgent.AddressToString(address));

        MessageBox.Show(
            "No map found for address:\n" +
            MapPointServiceAgent.AddressToString(address),
            "Error");
    }
}
```

6. Finally, update the event handler that is called when the user clicks the View Map button so that now the map retrieval is invoked asynchronously, and a delegate reference to the handler you just defined is passed when the asynchronous operation is called. The button click handler now becomes:

```
' VB
Private Sub _btnViewMap_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles _btnViewMap.Click
    Dim address As Address = New Address()

    address.AddressLine = _tboxAddressLine.Text
    address.PrimaryCity = _tboxCity.Text
    address.Subdivision = _tboxProvinceOrState.Text
    address.PostalCode = _tboxZipOrPostalCode.Text
    address.CountryRegion = CType( _
        _countriesComboBox.SelectedItem, String)
    _svcAgent.BeginGetSizedMapByAddress(address, _
        _mainPicBox.Height, _mainPicBox.Width, _
        DefaultDataSourceName, _
        AddressOf Me.HandleMapImageAvailableForDisplay)

        InitAddressInputFields()

    _mainStatusLabel.Text = String.Format( _
        "Retrieving map for address {0}", _
        MapPointServiceAgent.AddressToString(address))
End Sub

// C#
private void _btnViewMap_Click(object sender, EventArgs e)
{
    Address address = new Address();

    address.AddressLine = _tboxAddressLine.Text;
    address.PrimaryCity = _tboxCity.Text;
    address.Subdivision = _tboxProvinceOrState.Text;
    address.PostalCode = _tboxZipOrPostalCode.Text;
    address.CountryRegion = _countriesComboBox.SelectedItem as string;

    _svcAgent.BeginGetSizedMapByAddress(address,
        _mainPicBox.Height, _mainPicBox.Width,
        DefaultDataSourceName,
        this.HandleMapImageAvailableForDisplay);

    InitAddressInputFields();

    _mainStatusLabel.Text = string.Format(
        "Retrieving map for address {0}",
        MapPointServiceAgent.AddressToString(address));
}
```

Now the solution should build and run. At this point, you must also make sure that your ID and password are correct in this application's configuration file. When you run the application this time, you will notice that the application UI remains usable even when the map retrieval requests are being processed.

Lesson Summary

- You can use the svcutil command-line tool or Visual Studio to generate proxies from the WSDL of a non-WCF service.
- In some situations, when there is a lack of WS-Policy support, some of the service's policies can be found only in the documentation for the service, policies that might otherwise be found in the WSDL.
- Just as with WCF services, you can use proxy objects to call service operations either synchronously or asynchronously.

- You can use *BasicHttpBinding*, either in code or in a configuration file, to ensure that your WCF service is WS-I Basic Profile–compliant when exchanging messages with a non-WCF service.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, “Consuming Non-WCF Services.” The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the “Answers” section at the end of the book.

1. You need to consume a service built in Java. Which of the following are valid methods you can use to create a WCF proxy class to consume this service? Choose all that apply.
   - **A.** Use the *ChannelFactory* class to create a proxy object dynamically.
   - **B.** Use the svcutil command-line tool to generate a proxy class by referencing the WSDL for the service.
   - **C.** Manually define a proxy class that inherits from *ClientBase*.
   - **D.** Add a service reference in Visual Studio by referencing the WSDL for the service.

Chapter Review

To further practice and reinforce the skills you learned in this chapter, you can:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create solutions.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- You can generate a proxy class from a service's metadata, using svcutil or Visual Studio, or define a proxy class manually. For non-WCF services, the service metadata must be expressed in the standard WSDL format.
- You can create proxies to WCF services, either dynamically by using the *ChannelFactory* class or by instantiating a proxy class.
- You can use proxies to either WCF or non-WCF services to invoke operations on remote services.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- *ChannelFactory*
- proxy
- proxy class
- service agent
- service reference
- svcutil
- WS-I
- WS-I Basic Profile

Case Scenarios

In the following case scenarios, you will apply what you've learned in this chapter. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Building an e-Commerce Solution

You work for a consulting company that has just acquired a new customer that wants to enable its Web site for e-commerce. The Web site is composed mostly of static text and images to display the product line. The site will need, at the very least, shopping-cart capability as well as the ability to handle credit-card payment transactions and delivery of the product to the buyer. The company wants to do this on an extremely low budget. Your manager seeks your advice on how to handle this customer.

1. Your manager wants to know whether you can deliver a reasonably priced solution to the customer, one with the lowest possible upfront costs.
2. If so, what might the solution look like?

Case Scenario 2: Medical Imaging Application

You work on a team that is developing a medical imaging solution. For years now, you have provided a very powerful, rich-client application (a Windows Forms application) that radiologists inside the hospital use to view medical images and make diagnoses. However, you are being asked to provide medical image viewing to a Web application that referring physicians and partner insitutions can use outside of the hospital. Your team has decided that the Web application will be based on ASP.NET and that you will provide a set of common WCF services that both the Web application and the rich-client application can use. Your manager asks you the following questions:

1. Both the Web application and the rich-client application must deal with slow or lost connectivity, service operation retries, asynchronous submission, and retrieval of possibly large amounts of image data. What can you can do to avoid duplicating solutions to those challenges in both applications?
2. Can you maximize performance when retrieving large sets of image data for the rich-client application inside the enterprise?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Expand Your Knowledge of Service Agents

Improve your knowledge about service agents, and then build a more robust service agent by performing the following practices.

- **Practice 1** Read the following articles about agents and how they can be used to solve some of the problems a consumer can have when interacting with a service.
    - The ServiceConsoleHost project, a simple console application that hosts the service
    - The TaskClient project, a Windows Forms application used to consume the service
    - The Tasks.Entities project, a class library project that defines the Data contracts used by the service
    - The Tasks.Services project, a class library project that defines the Service contract and service type
    - The MapPointTestClient project, which is a Windows Forms application you use to consume the MapPoint service.
    - The Microsoft.MapPoint.ServiceAgent project, which is a class library project that initially contains only an almost empty app.config file and a text file that contains the svcutil command you'll need in a moment to generate a proxy. The project does, however, already have the appropriate references set up.
    - *Dealing with Concurrency: Designing Interaction Between Services and Their Agents*, by Maarten Mullender, which is available at *http://msdn2.microsoft.com/en-us/library/ms978508.aspx*
    - *Transparent Connectivity: A Solution Proposal for Service Agents*, by Maarten Mullender and Jon Tobey, which is available at *http://msdn2.microsoft.com/en-us/library/aa479367.aspx*
- **Practice 2** For the Task Manager service you worked with in Lesson 1, build a more capable TaskServiceAgent that can work in a completely offline mode. You can likely do this in several ways, but one feasible approach is to store any tasks that need to be submitted to the service in a local SQL server database until the agent is able to re-establish connectivity with the service.

Consume a Non-WCF Service

Consume another third-party service by performing the following practice.

- **Practice** Any of the various live.com services from Microsoft would afford good practice for consuming non-WCF services.

  Failing something that interests you among the live.com services, pick any third-party service that supports developer trial access and try to write a WCF proxy to consume it.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just one exam objective, or you can test yourself on all the 70-503 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO **Practice tests**

For details about all the practice test options available, see the "How to Use the Practice Tests" section in this book's introduction.

Chapter 10

Sessions and Instancing

When a discussion of Windows Communication Foundation (WCF) turns to the concepts and details of sessions and instancing, it seems as though you're starting to tread on common ground with Web developers. ASP.NET developers are likely to be familiar with the concepts associated with sessions. Although less directly, ASP.NET developers also deal with some of the aspects associated with instancing in relation to having a Web site hosted on a Web farm. However, the instancing issues will be more familiar to developers who have used .NET remoting in the past.

This chapter finds common ground for all developers, regardless of their background, and answers the question of how sessions function within the WCF world. It also describes the various instancing options and the implications each choice has on the available functionality. This content is definitely part of the certification exam; however, pay close attention because it is also frequently at the heart of real-world design choices.

Exam objectives in this chapter:

- Manage instances.
- Manage sessions.

Lessons in this chapter:

Before You Begin

To complete the lessons in this chapter, you must have:

- A computer that meets or exceeds the minimum hardware requirements listed in the introduction at the beginning of the book.
- Any edition of Microsoft Visual Studio 2008 (including Microsoft Visual C# 2008 Express edition or Microsoft Visual Basic 2008 Express edition) installed on the computer.

Real World

Bruce Johnson

The session side of this chapter is, again, probably familiar to those of you who have worked with ASP.NET. The idea is a simple one: By allowing the service to identify the client that made a request, it becomes possible to save state information with that client. This information can facilitate sophisticated interactions between the client and the service. What is nice about the session model that WCF provides is that the binding is, for the most part, irrelevant. Like so much else in WCF, the details are hidden from view, and it just works.

The instancing side of this chapter is a little different. It's more in the category of "what you need to know to be an expert." In most cases, you will not need to know the details of instancing beyond the need to set the mode to per call, per session, or singleton, but within that world, some strange things have been known to happen. You are moving into an area in which subtle bugs can arise, and if you have the detailed knowledge provided in this chapter, you will be able to identify the source of the problem more quickly and give off the heroic aura that experts are expected to have.

Lesson 1: Instancing Modes

Instancing should, for the most part, be a service-side implementation detail that has no effect on the client, and this is generally the case. However, the demands of the client frequently do influence the instancing that should be used. Instancing can affect scalability, throughput, transactions, and queued calls, so although the client might be oblivious to the instancing mode, the service can't reciprocate. This lesson considers the different types of possible instancing, along with how they are set up and the ramifications of the choices.

After this lesson, you will be able to:

- Identify the different instancing modes supported by WCF.
- Configure the service to preserve state information for calls from a single client.
- Share an instance of a proxy class between two or more clients.

Estimated lesson time: 50 minutes

Instancing

WCF is responsible for binding an incoming message to a particular service instance. When a request comes in, WCF determines whether an existing instance of the service class (the *service instance*) can process the request. The decision matrix for this choice is basically the instancing management that WCF provides.

When it comes to the question of which instancing mode to use, there is no correct answer. A variety of factors must be balanced to determine the most appropriate mode for the given situation. For this reason, this lesson covers all the modes in great detail and provides scenarios in which they might be the most appropriate choice. However, even with the given scenarios, the choice is seldom clear, and a small change in the importance of one factor can tip the scale to another choice. Your benefit from this discussion should be a general sense of when a particular mode is more or less likely to be chosen.

The determination of the instancing mode is done on the service side. This is to be expected because it is an implementation detail that should be hidden from the caller. The mode is defined within the service behavior. This means that the instancing mode is used across all the endpoints of a service. It can also be applied directly in the service's implementation class.

Three choices are available for the *InstanceContextMode*. They are per call mode, per session mode, and singleton mode. The meanings of these modes are described in the next few sections, but those are not the only available choices. In the original version of WCF, there was also an option in the *InstanceContextMode* enumeration called *Shareable*. Although the functionality still exists, the enumerated value does not. Instead, to share the same service instance across multiple requests, the service must intercept the request, determine which instance the requestor wants, and then provide that instance to the run time. The upcoming sections describe how this is done.

Per Call Mode

In per call mode, every single request gets its own copy of a service implementation object. Figure 10-1 illustrates the basic flow for the request

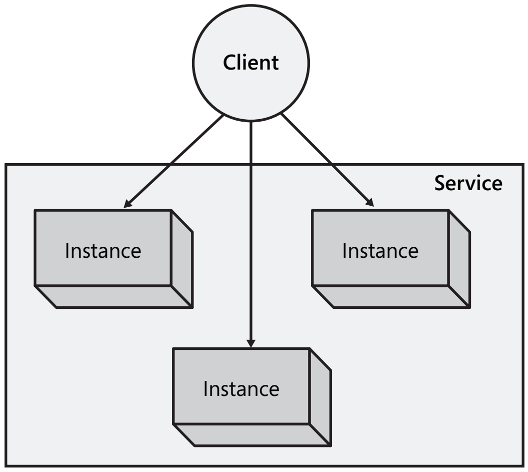


Figure 10-1 Per call instantiation

The client makes a request to the service through a proxy. When the request arrives at the service host, the host creates an instance of the service's implementation class. This class is then called to process the request. After the request is complete and the response returned to the client, the implementation object is disposed of.

NOTE IDisposable **and** Dispose

Each implementation object implements the *IDisposable* interface. When the object is finished (defined by the instancing mode), the *Dispose* method is called. Although the object has not necessarily been garbage collected immediately, it is in a state in which no further calls can be made to any method.

Per call instancing is the default mode for WCF. There are a number of reasons for making this particular choice. For the developer, per call mode requires the least amount of consideration given to concurrency. If each request has its own copy of the object, there is no need to worry about a shared value being updated in a non-atomic manner.

Historically, the instancing mode many client/server applications used was one implementation object per client. This is a simple approach, but a number of problems affect performance.

For example, consider the issue associated with a scarce resource. If the service object opens a connection to a database and keeps that connection open for its lifetime, the resource is unavailable for use by other instances, yet the period of time the resource might actually be required is quite small.

It is well understood by designers of distributed applications that this model has scalability weaknesses. One of the solutions is to reduce the time the implementation object exists. This is the genesis for the per call mode. In per call, the implementation object is instantiated as soon as it is needed, and it is disposed of as soon as the request is completed. If the object holds on to a scarce resource, the lifetime of the object has been reduced to minimize the impact holding that resource has on overall performance.

However, "simple to use" is not the same as "best." And that per call instancing hides many of the challenges associated with distributed applications doesn't mean that it should be the mode you always use. Consider some of the drawbacks associated with this approach.

A **scarce resource** is one that is expensive to allocate or is limited in the number available for use. A canonical example is a file that resides on the service system's hard drive. If the file is opened for update, only one service implementation instance can have it open at a time, so in a per call instancing mode, only the first request in can be processed through to completion. The second (and subsequent) requests will block, waiting for the physical file to become available. Although a physical file is an extreme scenario, there are many other scenarios. Database connections, network connections (used to make Web service calls), or communications ports all qualify as scarce resources.

One of the keys to making this model work is the existence of a proxy object for the service. The typical programming model that has already been discussed has the client instantiating an object and maintaining a reference to it for the life of the application. However, in per call mode, the object that is referred to *should* be disposed of. This would typically invalidate the reference, a generally undesirable outcome. However, in the world of WCF, the client is actually holding a reference to the proxy. The proxy is not disposed of with every call. Instead, it becomes part of the proxy's job to re-create the service implementation object as necessary.

An ancillary benefit to this model is how it works with transactional applications. The need to re-create the object and reconnect to scarce resources works well in an environment in which the instance state must be deterministic.

As has already been mentioned, the instancing mode is set at the service level. The following code demonstrates (in bold) how to set the mode to per call.

```
' VB
<ServiceBehavior(InstanceContextMode:=InstanceContextMode.PerCall)> _
Public Class UpdateService
   Implements IUpdateService
   ...
End Class

// C#
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerCall)]
class UpdateService : IUpdateService {...}
```

Although, theoretically, the client doesn't need to be aware of whether the service is running in per call mode, the reality is that per call means that no state can exist between calls. It becomes a design issue, but the client cannot expect that the results from one call will be preserved or used in the second call to the service. If this is a requirement, regardless of the reason, it becomes part of the service's task to ensure that state is saved across calls. This would typically be done by persisting the state into a service-local store (such as a database). Then, when subsequent requests come in, the previously saved state can be restored and used.

If the service's design calls for this pattern, there is an impact on the design of the Service contract. Specifically, each operation must include a parameter that identifies the client making the request. This allows the service method to retrieve the state associated with the client. The actual parameter that is used could be a business-level value (customer number, order number, account number) or a meaningless value (such as *guid*).

From a general design perspective, per call mode is best used when individual operations are short and the operation does not spawn any background threads that continue processing after the request is complete. The reason for this second stipulation has to do with the disposal of the implementation object. If an operation were to spin up something that isn't completed prior to the response being returned to the client, the object will not be around to receive the result. It will have been destroyed as soon as the request is finished.

Per Session Mode

Given the idea that a parameter would be passed into a service's method to retrieve state, it seems a short jump to this next mode. WCF can maintain a private session between a client and a particular instance of the service's implementation object.

The key to understanding the intricacies of per session mode is understanding what is happening internally. Each client, upon the first request to the service, gets an instance of the service's implementation object. This instance is dedicated to processing the requests that come from that client. Any subsequent calls are considered to be part of the same session (with some exceptions that will be described shortly), and the calls are processed by the same instance of the implementation object. Figure 10-2 illustrates this relationship.

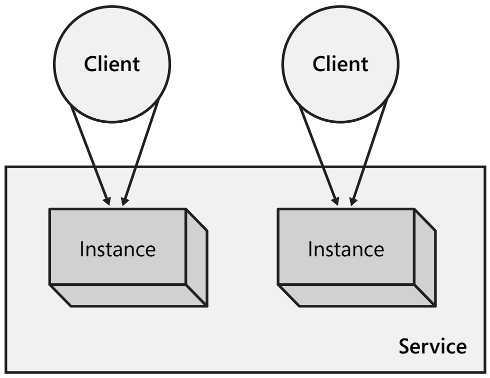


Figure 10-2 Per session mode interactions

There are two components to per session mode. The contractual piece involves letting the client know that a session is required. This is necessary because to maintain the session, the client must include an identifier to locate the appropriate implementation object in the service. To indicate to the contract that a session is to be maintained, the *ServiceContract* attribute includes a *SessionMode* property. For per session mode to be used, this Boolean value must be set to *SessionMode.Required*, as demonstrated in bold in the following code.

```
' VB
<ServiceContract(SessionMode:=SessionMode.Required)> _
Public Interface IUpdateService
   ' Interface definition code goes here
End Interface

// C#
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IUpdateService
{
   // Interface definition code goes here
}
```

The second component of the configuration is behavioral in nature. WCF needs to be told that you would like to use per session mode and that the service instance should be kept alive throughout the session. You do this by setting the *InstanceContextMode* in the service behavior as illustrated in bold in the following code.

```
' VB
<ServiceBehavior(InstanceContextMode:=InstanceContextMode.PerSession)> _
Public Class UpdateService
   Implements IUpdateService

   ' Implementation code goes here

End Class
```

```
// C#
[ServiceBehavior(InstanceContextMode = InstanceContextMode.PerSession)]
public class UpdateService : IUpdateService
{
   // Implementation code goes here
}
```

Now it's time to talk about some of the details. The relationship isn't quite between the client and the service. It is actually between a specific instance of the proxy class used by the client and the service. When you create a proxy for a WCF service, an identifier for that proxy is generated. This identifier is used by the service host to direct any requests to the appropriate instance. However, the identifier is associated with the instance of the proxy class, so if a single client creates more than one instance of the proxy class, those instances will not combine sessions. Each proxy will get its own instance of the service implementation class.

After an instance is created for a proxy, the instance remains in memory for the length of the session. For this reason, it is possible to maintain state in memory. This makes the programming model quite similar to the traditional client–server approach, but this also means that per session mode suffers from the same issues that the client server model has. It has issues with scalability, needs to be aware of state, and can have problems with transactions. The practical limit for a service is no more than a few hundred clients.

As mentioned earlier, the service instance lasts until the client no longer requires it. Again, there are caveats to that generalization. The most efficient path for session termination involves the client closing the proxy. This causes a notification to be sent to the service that the session has ended, but what happens if the client doesn't close the proxy? What happens if the client doesn't terminate gracefully or a communications issue between the client and service prevents the notification from being received? In these cases, the session will automatically terminate after ten minutes of inactivity. After the session has been terminated in such a manner, the client will receive a *CommunicationObjectFaultedException* if it attempts to use the proxy.

This ten-minute timeout is just the default value. Whether the default can be changed depends on the binding. If the binding supports a reliable session, you can set the *InactivityTimeout* property associated with the reliable session. The following code demonstrates how to do this with a *netTcpBinding* binding.

```
' VB
Dim binding As New NetTcpBinding()
binding.ReliableSession.Enabled = True
binding.ReliableSession.InactivityTimeout = TimeSpan.FromMinutes(60)

// C#
NetTcpBinding binding = new NetTcpBinding();
binding.ReliableSession.Enabled = true;
binding.ReliableSession.InactivityTimeout = TimeSpan.FromMinutes(60);
```

You can make the same setting through configuration files, as illustrated in the following segment:

```
<netTcpBinding>
    <binding name="timeoutSession">
        <reliableSession enabled="true" inactivityTimeout="01:00:00"/>
    </binding>
</netTcpBinding>
```

Exam Tip As you might surmise, it is possible for the inactivity timeout to be configured at both the client and the service. If the times are different, the shortest configured timeout prevails.

Speaking of reliable sessions, support for reliable sessions is required for a binding to support sessions. All the endpoints that expose the Service contract must use bindings that support reliable transport sessions, and the session must be enabled as shown in the earlier code example. This constraint is validated when the service is loaded; an *InvalidOperationException* is thrown if there is a mismatch.

One binding that is unable to support sessions is *basicHttpBinding*. Within the protocol that underlies this binding, there is no way to pass the information necessary to maintain the session. You can overcome the problem with the transport protocol within the format of the messages. The *wsHttpBinding* binding is capable of providing the necessary data to support sessions, for example.

There is one exception to the reliable session rule. The named pipe binding supports reliability by definition, so there is no need for the reliable messaging protocol to be implemented—and the *netNamedPipeBinding* binding does support sessions.

Singleton Mode

In this mode, only one instance of the service's implementation class is created. This instance is enlisted to handle every request that arrives at the service. The instance lives forever (or close to forever) and is disposed of only when the host process shuts down.

Singleton mode does not require any session information to be transmitted with the message. As a result, there is no restriction on the ability of the binding to support transport-level sessions. Nor is there a need for the protocol or binding to provide a mechanism that appears to emulate session behavior. If the contract exposed by the service has a session, the client must provide the session, but there is no requirement for sessions for singleton mode to work. Further, if a session is associated with the request, that session will never expire. The session identifier is maintained within the client proxy until the proxy is destroyed.

Alternatively, if no session information is exposed by the contract, the communications don't fall back to per call mode (unlike other modes). Instead, the request continues to be handled by the single instance of the singleton service.

Configure a singleton service in a manner similar to the other modes. The *InstanceContextMode* property of the *ServiceBehavior* attribute is set to *Single*. The following code demonstrates this, as shown in bold:

```
' VB
<ServiceBehavior(InstanceContextMode:=InstanceContextMode.Single)> _
Public Class UpdateService
   Implements IUpdateService
' Implementation code goes here
End Class

// C#
[ServiceBehavior(InstanceContextMode = InstanceContextMode.Single)]
public class UpdateService : IUpdateService
{
   // Implementation code goes here
}
```

One of the features the singleton behavior offers is the ability to initialize the implementation instance through the constructor. For the other behaviors, the instance object is created behind the scenes at a time determined by the host process, but for singletons, you have the option to create the singleton instance and pass it into the host process.

Naturally, this begs the question of why you would want to do this. Typically, the rationale involves performing initialization processing outside the scope of the first request. If the service instance needs to allocate some resources (such as connecting to a database), the default behavior is to have the first request pay that performance price. It might be that logic should be injected into the service instance that is not available to the client (and, therefore, couldn't be included in the request). In both of these cases (and there are other reasons as well), having the service host create the singleton instance is not the best alternative.

However, in singleton mode, you can create the service instance before the host is even started. This instance can then be passed into the host as the host is getting started. One of the constructors for the *ServiceHost* class takes a singleton instance as a parameter. When constructed in this manner, the host will direct all incoming requests to the provided instance. The following code demonstrates how this is accomplished.

```
' VB
Dim singletonInstance As New SingletonUpdateService()
Dim host As New ServiceHost(singletonInstance)
host.Open()

// C#
SingletonUpdateService singletonInstance = new SingletonUpdateService();
ServiceHost host = new ServiceHost(singletonInstance);
host.Open();
```

For the preceding code to work, the service (*SingletonUpdateService* in this example) must be defined with the *InstanceContextMode* property in the *ServiceBehavior* attribute set to *Single*.

When the service host is using a singleton instance, it is also possible for other objects to reach into the instance to call methods or set parameters. The *ServiceHost* class exposes a *Singleton-Instance* property that references the instance processing the incoming requests. The following code demonstrates how to update a member of the instance:

```
' VB
Dim instance As SingletonUpdateService = _
   TryCast(host.SingletonInstance, SingletonUpdateService)
instance.Counter += 50

// C#
SingletonUpdateService instance = host.SingletonInstance as
    SingletonUpdateService;
instance.Counter += 50;
```

Even if the local host object variable is not available, you can still gain access to the instance. The *OperationContext* class exposes a read-only *Host* property, so from within an operation, the singleton instance can be accessed.

```
' VB
Dim host As ServiceHost = TryCast(OperationContext.Current.Host, _
   ServiceHost)
If host IsNot Nothing Then
   Dim instance as SingletonUpdateService = _
      TryCast(host.SingletonInstance, SingletonUpdateService)
   If instance IsNot Nothing Then
      Instance.Counter += 1
   End If
End If

// C#
ServiceHost host = OperationContext.Current.Host as ServiceHost;
if (host != null)
{
   SingletonUpdateService instance = host.SingletonInstance
      as SingletonUpdateService;
   if (instance != null)
      instance.Counter += 1;
}
```

That every request is handled by a single instance of the implementation class has potential implications for contention issues. If multiple requests arrive at the service, they will be processed, in many cases, by the same instance but in a different worker thread. This means that any variable scoped outside of the current method (that is a class-level variable) can be corrupted if the value is updated by two worker threads at once. You must ensure that updates are performed using concurrency techniques such as locking.

The side effect of dealing with concurrency is that, at least in areas that are synchronized, only one request can be processed at a time. If the singleton service has a number of areas that require synchronization, or even if there is only one but it is in a frequently used method, performance can be negatively affected.

From a design perspective, singleton services are best used when they are modeling a singleton resource—a log file, perhaps, that allows a single writer only or, as has recently happened in the real-world job mentioned earlier, communicating with a single robot. If there is a possibility that, in the future, the service might no longer be a singleton, think hard before using this model. Subtle dependencies can be introduced while using the singleton model. The client might come to expect that state will be shared across multiple requests. Although the change to reconfigure the service to be something other than a singleton is simple, the challenge of tracking down dependency bugs can be much worse.

Sharing Instances

As has been mentioned, the mechanism for sharing a service instance between multiple clients has changed from the original approach. By creating a class that implements the *IInstanceContextProvider* interface and then injecting the class into the dispatch pipeline, you can have a great deal of control over which instance of the service class will be used to service each request.

The starting point must come from the client. For the service to distinguish between the different clients, it examines each incoming request. Based on information that exists within the request, an existing instance is provided (or a new one is created). This generally means that the client needs to place something in the request, such as a message header. The easiest way to accomplish this is to use the *MessageHeader* class factory to create an instance of a *MessageHeader* object. That object can then be added to the message headers that are sent with the request. The following code demonstrates how to do this.

NOTE Import the *System.ServiceModel.Channels* namespace

The *MessageHeader* class used in this code exists in the *System.ServiceModel.Channels* namespace. Unless this namespace is imported into the code file, you might receive an error message indicating that *MessageHeader* is a generic type that expects a parameter.

```
' VB
Dim header As MessageHeader = _
   MessageHeader.CreateHeader("headerName", "headerNamespace", _
   "instanceId")

Using SessionClient proxy As NewSessionClient()
   Using (New OperationContextScope(proxy.InnerChannel))
      OperationContext.Current.OutgoingMessageHeaders.Add(header)
      ' use the proxy object
   End Using
End Using

// C#
MessageHeader header = MessageHeader.CreateHeader("headerName",
   "headerNamespace", "instanceId");

using (SessionClient proxy = new SessionClient())
{
   using (new OperationContextScope(proxy.InnerChannel))
   {
       OperationContext.Current.OutgoingMessageHeaders.Add(header);
        // Use the proxy object
   }
}
```

The idea is that any client making a request to the service will use this pattern of code. If two clients must share an instance, the instance ID from one client will be sent to the second client, which would then include that in the message header it sends to the service.

Sending the header information is just the starting point. On the service side, the presence of the instance ID must be recognized and extracted from the request. This ID is then used as the key to a collection of previously created instances. If the corresponding instance already exists in the collection, it must be used to process the request. If the instance ID does not exist, a new instance must be created and then added to the collection to handle future requests.

The mechanism to implement the preceding scenario might not be obvious. Fortunately, Microsoft uses a provider model for the creation of instances to process requests. The interface for this is named *IInstanceContextProvider*. This interface exposes four methods: *GetExisting-InstanceContext*, *InitializeInstanceContext*, *IsIdle*, and *NotifyIdle*. These four methods actually work in two groups.

GetExistingInstanceContext and *InitializeInstanceContext* work in concert to determine which instance of the service's implementation object will be used to create the response. The *Get-ExistingInstanceContext* method is invoked as part of the process of handling an incoming request. The result from this method is either an existing instance context or a value of *null/ Nothing*. In the latter case, WCF recognizes that no instance has been previously created, so it creates a new instance and then invokes the *InitializeInstanceContext* method. The idea is that any setup that must be performed on the new instance will be done in the *InitializeInstance-Context* method. In the case of the instance-sharing mode, this would normally include saving the new instance so that it can be retrieved in a future call to *GetExistingInstanceContext*.

WCF uses the *IsIdle* and *NotifyIdle* methods when it believes that all the activities associated with an instance have been completed. At this point, the *IsIdle* method is invoked. It is up to this method to determine whether the client (or clients) no longer needs the instance. The method returns a Boolean value, and if it returns *true*, then WCF will close the context.

Alternatively, if *IsIdle* returns *False*, that is a signal to WCF that the client might still need the particular instance. At this point, WCF invokes the *NotifyIdle* method. This method includes as one of the parameters a callback method. The idea is that, after the instance is no longer required (as determined by the provider), the method reference by the callback parameter will be invoked. This notifies WCF that the instance is no longer required. It will then start the instance deactivation process (including a call to the *IsIdle* method) once again.

Lab: Instance Modes

In this lab, you will focus on experimenting with the different instancing modes available in WCF. The first exercise looks at the *InstanceContextMode* enumeration, illustrating the different possible behaviors. The second exercise walks you through the creation of an instance context provider and illustrates how it can be used to share instances between clients.

▶ Exercise 1 Per Session, Per Call, and Singleton Modes

In this first exercise, you will use the *InstanceContextMode* value to determine the instancing WCF uses as well as to demonstrate the behavior of each mode by using a variable that is private to the implementation class.

1. Navigate to the <*InstallHome*>/Chapter10/Lesson1/Exercise1/<*language*>/Before directory and double-click the Exercise1.sln file to open the solution in Visual Studio.

   The solution consists of two projects. They are as follows:

   - The DemoService project, a simple WCF service library that implements the *ISession* interface. This interface consists of a single method (*GetSessionStatus*) that returns a string indicating the number of times the method has been called within the current service instance.
   - The TestClient project, a Console application that generates a request for the service and displays the result in the Console window.

2. In Solution Explorer, double-click the Program.cs or Mobile1.vb file in the TestClient project.

   In this file, you can see the lines of code that send requests to the service. Initially, there are two calls, back to back. First, set up the service to use the *PerCall* instance method. This is actually redundant because that is the default mode, but it does set up for the other modes.

3. To start, in Solution Explorer, double-click the SessionService file.

   The declaration for the *SessionService* class includes the *ServiceBehavior* attribute. One of the properties for that class is named *InstanceContextMode*. You can assign this value through the attribute by using a named parameter format.

   Change the class declaration to be the following:

   ```
   ' VB
   <ServiceBehavior(InstanceContextMode:=InstanceContextMode.PerCall)> _
   ```

```
Public Class SessionService
   Implements ISession

// C#
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall)]
public class SessionService : ISession
```

4. Ensure that TestClient is set as the startup project and launch the application by pressing F5.

   After a few moments, you will see that two messages appear. Each message indicates that the instance has been called only one time, even though the same proxy object is being used. This is to be expected when the instance is created once per call.

5. Press Enter to stop running the application.

6. In the SessionService file, change the instance context mode from *PerCall* to *PerSession*. When you are finished, the class declaration will look like the following (changes shown in bold):

```
' VB
<ServiceBehavior(InstanceContextMode:=InstanceContextMode.PerSession)> _
Public Class SessionService
   Implements ISession

// C#
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerSession)]
public class SessionService : ISession
```

   For session mode to work, the service interface must be marked as requiring an interface.

7. In Solution Explorer, double-click the ISession file.

   The declaration for the *ISession* interface includes a *ServiceContract* attribute. The attribute includes a *SessionMode* property, which you must set to *Required*.

8. Modify the interface's declaration as shown in bold to look like the following:

```
' VB
<ServiceContract(SessionMode:=SessionMode.Required)> _
Public Interface ISession

// C#
[ServiceContract(SessionMode=SessionMode.Required)]
public interface ISession
```

9. Launch the application by pressing F5.

   After a few moments, you will see that two messages appear. The messages indicate that a single instance of the service class has been called twice. Again, this is the expectation when the instance is created once per session.

10. Press Enter to terminate the application.

    To simulate two clients, the client application can create two separate *using* blocks.

11. In the Program.cs or Module1.vb file, add a second *using* block that creates a new proxy object and invokes the service. Change the *Main* method so that the body looks like the following:

```
' VB
Using proxy As New DemoService.GetSessionStatusClient()
   Console.WriteLine("First call: " + proxy.GetSessionStatus())
End Using

Using proxy As New DemoService.GetSessionStatusClient()
   Console.WriteLine("Second call: " + proxy.GetSessionStatus())
End Using
Console.ReadLine()

// C#
using (DemoService.GetSessionStatusClient proxy = new
   DemoService.GetSessionStatusClient())
{
   Console.WriteLine("First call: " + proxy.GetSessionStatus());
}

using (DemoService.GetSessionStatusClient proxy = new
   DemoService.GetSessionStatusClient())
{
   Console.WriteLine("Second call: " + proxy.GetSessionStatus());
}
Console.ReadLine();
```

12. Launch the application by pressing F5.

    In a few moments, the messages will appear on the console. The messages indicate that even though the instance context mode is set to *PerSession*, the different *using* blocks result in two different sessions.

13. Press Enter to terminate the application.

14. In the SessionService file, change the instance mode to Single.

    The declaration for the *SessionService* class should read as follows (changes shown in bold):

```
' VB
<ServiceBehavior(InstanceContextMode:=InstanceContextMode.Single)> _
Public Class SessionService
   Implements ISession

// C#
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
public class SessionService : ISession
```

15. Launch the application one last time by pressing F5.

In a few moments, the console messages appear. In this case, they indicate that even though two different sessions have been created (there are still two *using* blocks), they both use the same session instance.

16. Press Enter to terminate the application

▶ Exercise 2 Share Service Instances

The fourth instancing mode for WCF services used to be known as *Shareable*. WCF uses a provider model to determine which instance of a service implementation class should be used. In this exercise, you will create a custom provider for instances and inject it into the WCF pipeline. The instance ID will be a number typed into the client to emulate the sharing process.

1. Navigate to the <*InstallHome*>/Chapter10/Lesson1/Exercise2/<*language*>/Before directory and double-click the Exercise2.sln file to open the solution in Visual Studio.

   The solution consists of two projects. They are as follows:

   ❑ The DemoService project, a simple WCF service library that implements the *ISession* interface. This interface consists of a single method (*GetSessionStatus*) that returns a string indicating the number of times the method has been called within the current service instance.

   ❑ The TestClient project, a Console application that generates a request for the service and displays the result in the Console window.

2. In Solution Explorer, double-click the DemoContextInfo file.

   This file will store information about an individual instance context. The provider will maintain a dictionary of DemoContextInfo files. This class implements the *IExtension* interface. The interface facilitates the aggregation of classes into the WCF pipeline, although in this particular case, the methods associated with this interface (*Attach* and *Detach*) are not needed for the implementation.

3. In Solution Explorer, double-click the DemoContextProvider file.

   This file will provide the implementation for the provider. This class must implement the *IInstanceContextProvider* interface.

4. Change the class declaration to be as follows:

```
' VB
Public Class DemoContextProvider
   Implements IInstanceContextProvider

// C#
public class DemoContextProvider : IInstanceContextProvider
```

   The interface requires four methods to be added.

5. Add the following method blocks to fulfill this requirement:

```
' VB
Public Function GetExistingInstanceContext(message As Message, _
    channel As IContextChannel) As InstanceContext _
    Implements IInstanceContextProvider.GetExistingInstanceContext
End Function

Public Sub InitializeInstanceContext(instanceContext As InstanceContext, _
    message As Message, channel As IContextChannel) _
    Implements IInstanceContextProvider.InitializeInstanceContext
End Sub

Public Function IsIdle(instanceContext As InstanceContext) As Boolean _
   Implements IInstanceContextProvider.IsIdle
End Function

Public Sub NotifyIdle(callback As InstanceContextIdleCallback, _
   instanceContext As InstanceContext) _
  Implements IInstanceContextProvider.NotifyIdle
End Sub

// C#
public InstanceContext GetExistingInstanceContext(Message message,
   IContextChannel channel) { }

public void InitializeInstanceContext(InstanceContext instanceContext,
   Message message, IContextChannel channel) { }

public bool IsIdle(InstanceContext instanceContext)
{
    return false;
}

public void NotifyIdle(InstanceContextIdleCallback callback,
   InstanceContext instanceContext) { }
```

In the *GetExistingInstanceContext* method, the first step is to retrieve the instance ID from the request.

6. Add the following code to the *GetExistingInstanceContext* method.

```
' VB
Dim headerIndex As Integer = message.Headers.FindHeader(headerName, _
    headerNamespace)

Dim _instanceId As String = String.Empty
If headerIndex <> -1 Then
   _instanceId = message.Headers.GetHeader(Of String)(headerIndex)
End If

// C#
int headerIndex = message.Headers.FindHeader(headerName, headerNamespace);

string instanceId = String.Empty;
if (headerIndex != -1)
   instanceId = message.Headers.GetHeader<string>(headerIndex);
```

7. If the request is associated with a session, the information about the instance will have been added as one of the extensions in the channel. If so, retrieve it. Add the following code below the newly added lines.

```
' VB
Dim info As DemoContextInfo = Nothing
Dim hasSession As Boolean = (channel.SessionId IsNot Nothing)
If hasSession Then
   info = channel.Extensions.Find(Of DemoContextInfo)()
End If

// C#
DemoContextInfo info = null;
bool hasSession = (channel.SessionId != null);
if (hasSession)
    info = channel.Extensions.Find<DemoContextInfo>();
```

8. If the request has an instance ID associated with it, there might already be a context to use. If so, retrieve it from the dictionary. Otherwise, instantiate a new *DemoContextInfo* object and add it to the dictionary. Add the following code to the *GetExistingInstance-Context* method below the lines added in the previous step.

```
' VB
Dim isNew As Boolean = False
If String.IsNullOrEmpty(_instanceId) OrElse Not _
   contextMap.TryGetValue(_instanceId, info) Then
   info = New DemoContextInfo(_instanceId)
   isNew = True
   contextMap.Add(_instanceId, info)
   If hasSession Then
      channel.Extensions.Add(info)
   End If
End If

// C#
bool isNew = false;
if (String.IsNullOrEmpty(instanceId) ||
   ! contextMap.TryGetValue(instanceId, out info))
{
   info = new DemoContextInfo(instanceId);
   isNew = true;
   contextMap.Add(instanceId, info);
   if (hasSession)
      channel.Extensions.Add(info);
}
```

At the end of the *GetExistingInstanceContext* method, the choice is to return a *null/Nothing* value (if there was no existing instance context) or return the instance context the provider found. In the latter case, information about the channel is added to the channels associated with the instance. This enables the instance to track the different channels with which it is operating.

9. Add the following code at the bottom of the *GetExistingInstanceContext* method:

```
' VB
If isNew Then
   Return Nothing
Else
   Dim _instance As InstanceContext = info.Instance
   If hasSession Then
      _instance.IncomingChannels.Add(channel)
   End If
   Return _instance
End If

// C#
if (isNew)
{
   return null;
}
else
{
   InstanceContext instanceContext = info.Instance;
   if (hasSession)
      instanceContext.IncomingChannels.Add(channel);

   return instanceContext;
}
```

In this interface, the other method of importance is *InitializeInstanceContext*. This method is called when the *GetExistingInstanceContext* returns *null/Nothing* and a new instance context has to be created. For this exercise, the code in this method will add the new instance to the dictionary of instances.

10. To start, check whether there is an existing session because, if so, the instance is already associated with the channel through the *Extensions* collection. Add the following code to the beginning of the *InitializeInstanceContext* method:

```
' VB
Dim info As DemoContextInfo = Nothing
Dim hasSession As Boolean = (channel.SessionId IsNot Nothing)

If hasSession Then
   instanceContext.IncomingChannels.Add(channel)
   info = channel.Extensions.Find(Of DemoContextInfo)()
End If

// C#
DemoContextInfo info = null;
bool hasSession = (channel.SessionId != null);

if (hasSession)
{
   instanceContext.IncomingChannels.Add(channel);
   info = channel.Extensions.Find<DemoContextInfo>();
}
```

11. If there is no existing session, get the instance ID from the headers in the request and see whether the ID can be found in the dictionary of previously used instances. Add the following *else* clause to the just-added *if* statement.

```
' VB
Else
   Dim headerIndex As Integer = message.Headers.FindHeader(headerName, _
      headerNamespace)
   If headerIndex <> -1 Then
      Dim instanceId As String = _
          message.Headers.GetHeader(Of string)(headerIndex)
      If instanceId IsNot Nothing Then
         contextMap.TryGetValue(instanceId, info)
      End If
   End If

// C#
else
{
   int headerIndex = message.Headers.FindHeader(headerName,
      headerNamespace);
   if (headerIndex != -1)
   {
      string instanceId = message.Headers.GetHeader<string>(headerIndex);
      if (instanceId != null)
         this.contextMap.TryGetValue(instanceId, out info);
   }
}
```

If, for any reason, the instance context was found, it must be added to the *DemoContextInfo* object that will be used to process the request.

12. Add the following lines to the bottom of the *InitializeInstanceContext* method:

```
' VB
If info IsNot Nothing Then
   Info.Instance = instanceContext
End If

// C#
if (info != null)
   info.Instance = instanceContext;
```

There are a number of ways to inject this functionality into the WCF pipeline. They are described in Chapter 9, "When Simple Is Not Sufficient," in the discussion of the details surrounding the *DispatchRuntime* object. For this exercise, you create an attribute to decorate the implementation class. The file for the attribute already exists.

13. In Solution Explorer, double-click the ShareableAttribute file.

The class is already decorated with the *IServiceBehavior* interface. This requires the three methods in the class to be defined. To add the *InstanceContextProvider*, the only method that must have code is *ApplyDispatchBehavior*. In this method, every endpoint dispatcher on every channel will set the *InstanceContextProvider* property to a new instance of the *DemoContextProvider* class.

14. Add the following code to the *ApplyDispatchBehavior* method:

```
' VB
Dim extension As New DemoContextProvider()
Dim dispatcherBase As ChannelDispatcherBase
For Each dispatcherBase In serviceHostBase.ChannelDispatchers
   Dim dispatcher As ChannelDispatcher = TryCast(dispatcherBase, _
      ChannelDispatcher)
   Dim _endpointDispatcher As EndpointDispatcher
   For Each _endpointDispatcher in dispatcher.Endpoints
      _endpointDispatcher.DispatchRuntime.InstanceContextProvider = _
         extension
   Next
Next

// C#
DemoContextProvider extension = new DemoContextProvider();
foreach (ChannelDispatcherBase dispatcherBase in
   serviceHostBase.ChannelDispatchers)
{
   ChannelDispatcher dispatcher = dispatcherBase as ChannelDispatcher;
   foreach (EndpointDispatcher endpointDispatcher in dispatcher.Endpoints)
   {
      endpointDispatcher.DispatchRuntime.InstanceContextProvider =
         extension;
   }
}
```

Now that the attribute has been created, the service's implementation class must be decorated with it.

15. First, in Solution Explorer, double-click *SessionService*.
16. In the class declaration, add the *Shareable* attribute. When you're finished, the class *declaration* should look like the following:

```
' VB
<ServiceBehavior(InstanceContextMode:=InstanceContextMode.Single)> _
<Shareable> _
Public Class SessionService
   Implements ISession

// C#
[ServiceBehavior(InstanceContextMode=InstanceContextMode.Single)]
[Shareable]
public class SessionService : ISession
```

17. Before starting the demo, in Solution Explorer, double-click the Program.cs or Module1.vb file in TestClient.

    Notice that there is a loop that prompts for an instance ID. Within that loop, there is a *using* block for the proxy to the service. This means that the same session will not be used for each call and that the only way for the instances to be maintained is through the provider that you have just written.

18. Ensure that TestClient is set to be the startup project, and then launch the application by pressing F5.

    You will prompted for an instance ID.

19. Enter the instance ID of your choice (say, 123, to keep it simple).

    The returned message indicates that this method has been called once.

20. Enter the same instance ID, and the instance has been called twice. Enter a different instance ID, and the counter restarts; if you later duplicate an earlier instance ID, you will see the previous counter incremented in the output on the console. When you have finished exercising the application, press Enter to terminate.

Lesson Summary

- The instance mode determines the relationship between the client and the instance of the service's implementation class.
- Along with the standard modes, WCF also provides a provider model to determine the instance context that should be used to process a request.
- *PerCall* is the default mode, and it maintains a one-to-one association between method calls and instances.
- *PerSession* creates an instance for each client proxy whereas an instance mode of *Single* results in one instance handling every request.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 1, "Instancing Modes." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. You have created a WCF application by which the client communicates with the service, using the *netTcpBinding*. You would like to minimize any possible threading and synchronization issues in the service. Which instance mode should you use?
   - **A.** Per call
   - **B.** Per session
   - **C.** Singleton
   - **D.** Instance context provider
2. You have created a WCF application by which the client communicates with the service, using the *wsTcpBinding*. A number of methods in the service retrieve a large quantity of relatively static data. You would like to minimize the processing time spent retrieving the data (and keep the data in a cache within the service object). Which instance mode should you use?
   - **A.** Per call
   - **B.** Per session
   - **C.** Singleton
   - **D.** Instance context provider

Lesson 2: Working with Instances

The instance mode WCF uses is just the start of working with instances. You can manipulate a number of details to improve the performance and scalability of a WCF service. WCF provides throttling and quota capabilities that can help prevent denial of service (DoS) attacks as well as ensure that the servers aren't overloaded by handling requests. Along the same lines, you can control the activation and deactivation of the instances used to process requests to a degree that is finer than the default functionality.

Not only does WCF allow for performance to be protected, some attributes can be set to demarcate operations. The demarcation ensures that, where necessary, some operations cannot be completed before or after other operations. This is not a complete workflow management function, but it does allow a service to ensure that a particular operation is called first and that no operations can be called after a finalize operation has been performed.

After this lesson, you will be able to:

- Protect a WCF service by setting the throttling and quota parameters.
- Demarcate service operations.
- Manage instance activation and deactivation at a very granular level.

Estimated lesson time: 50 minutes

Protecting the Service

When WCF is deployed in the real world (where *real* is defined as a distributed environment in which requests arrive at a pace that is outside of your control), a number of potential problems can arise. Some of the performance differences associated with the different instancing modes have already been covered. However, beyond pure performance problems, WCF services have to contend with some of the same problems that a Web site has to contend with. This includes the potential for being flooded with client requests, similar to a denial of service attack.

Denial of service attacks are attempts to deplete the resources required by the service to process incoming requests to the point that no additional resources are available. The type of depleted resources can include any scarce resource the service uses. WCF provides a number of ways to mitigate the problem through either throttling requests or applying quotas to the resource.

Throttling

The goal of throttling is twofold. First, it prevents the service host from being overrun by a flood of requests. Second, it enables the load on the WCF service (and the server on which the service is running) to be smoothed out. In both cases, the intent is to place a limit on the number of incoming requests so that the service will be able to handle them in a timely manner.

The default WCF setting for throttling is to have none at all. When throttling is engaged, WCF will check the current counters for each request that arrives. If the configured settings are exceeded, WCF automatically places the request in a queue. As the counters come down below the threshold, the requests are then retrieved from the queue in the same order and presented to the service for processing. The result of this is that, in many cases, the observed behavior for a service that has reached its maximum is to have the client request time out.

Three settings in the service behavior control the number of requests the service host will be allowed to process simultaneously. Each of these is defined in the *ServiceThrottlingBehavior* section of the configuration file. The following paragraphs describe the three settings and are followed by an example of how you can configure them.

MaxConcurrentCalls The *MaxConcurrentCalls* value specifies the number of simultaneous calls the service will accept. The default value is 16 calls. Of the three settings, this is the only one that covers all the types of requests that arrive.

MaxConcurrentSessions The *MaxConcurrentSessions* value determines the maximum number of channels requiring sessions that the service will support. The default value for this setting is 10 session-aware channels. Any attempt to create a channel beyond this maximum will throw a *TimeoutException*. Because this setting is concerned with session-aware channels only, if the binding is not session-aware (such as the *basicHttpBinding*), this setting has no impact on the number of requests that can be processed.

MaxConcurrentInstances The *MaxConcurrentInstances* setting sets the maximum number of instances of the service implementation object that will be created. The default value for this setting is *Int32.MaxValue*, and the impact this value has on the service depends on the mode. If the mode is per call, this is the same as *MaxConcurrentCalls* because each call gets its own instance. If the mode is per session, the setting works the same as *MaxConcurrentSessions*. For singleton mode, the value of the number of instances is always 1, so the setting is really only useful when the *IInstanceContextProvider* is being used.

The following segment from a configuration file demonstrates how you can configure these settings:

```
<behaviors>
   <serviceBehaviors>
      <behavior name="throttlingBehaviort">
         <serviceThrottling maxConcurrentCalls="10"
            maxConcurrentInstances="10"
            maxConcurrentSessions="5"/>
      </behavior>
   </serviceBehaviors>
</behaviors>
```

You can set the same configuration through code. The following segments demonstrate the technique:

```
' VB
Dim host As New ServiceHost(GetType(UpdateService), _
   New Uri("http://localhost:8080/UpdateService"))
host.AddServiceEndpoint("IUpdateService", _
   New WSHttpBinding(), String.Empty)
Dim throttlingBehavior As New ServiceThrottlingBehavior()
throttlingBehavior.MaxConcurrentCalls = 10
throttlingBehavior.MaxConcurrentInstances = 10
throttlingBehavior.MaxConcurrentSessions = 5
host.Description.Behaviors.Add(throttlingBehavior)
host.Open()

// C#
ServiceHost host = new ServiceHost( typeof(UpdateService),
   new Uri("http://localhost:8080/UpdateService"));
host.AddServiceEndpoint( "IUpdateService",
   new WSHttpBinding(), String.Empty);
ServiceThrottlingBehavior throttlingBehavior = new ServiceThrottlingBehavior();
throttlingBehavior.MaxConcurrentCalls = 10;
throttlingBehavior.MaxConcurrentInstances = 10;
throttlingBehavior.MaxConcurrentSessions = 5;
host.Description.Behaviors.Add(throttlingBehavior);
host.Open();
```

As has been mentioned, when the throttling limits are reached, the client will throw an exception. Specifically, the exception the client receives is the previously mentioned *TimeoutException*. Because this one exception fits all scenarios (that is, the same exception is raised regardless of which of the throttling settings caused the problem), it is left up to you to discover the cause. A couple of hints can help. If the problem is caused by the concurrent sessions limit, you will most likely see the exception raised within the *SendPreamble* method. If it turns out that the *Send* method is the source of the time out, it is more likely to be caused by the maximum concurrent calls limit.

NOTE No need to use code

There is little reason to configure the throttling behavior in code. By keeping it in the configuration file, you enable administrators to adjust the service's performance on an as-needed basis.

Reading Throttling Settings

It is possible to read (but not update) the current throttling settings after the service host has been opened. Applications do this, typically to provide diagnostic information about the service. You do this by accessing the dispatcher for the service, which is responsible for implementing the throttling, so it makes sense that the dispatcher would have all the information close at hand.

The *ServiceHost* class exposes a collection of dispatchers in the *ChannelDispatchers* property. This is a strongly typed collection of *ChannelDispatched* objects. The *ChannelDispatcher* object has a property called *ServiceThrottle*. Through the *ServiceThrottle* object, you have access to all the throttling properties, including *MaxConcurrentCalls*, *MaxConcurrentInstances*, and *MaxConcurrentSessions*. The following code demonstrates this technique:

```
' VB
Dim dispatcher As ChannelDispatcher = _
   TryCast(OperationContext.Current.Host.ChannelDispatchers(0), _
   ChannelDispatcher)

Dim throttle as ServiceThrottle = dispatcher.ServiceThrottle

Trace.WriteLine(String.Format("MaxConcurrentCalls = {0}", _
   throttle.MaxConcurrentCalls))
Trace.WriteLine(String.Format("MaxConcurrentSessions = {0}", _
   throttle.MaxConcurrentSessions))
Trace.WriteLine(String.Format("MaxConcurrentInstances = {0}", _
   throttle.MaxConcurrentInstances))

// C#
ChannelDispatcher dispatcher =
   OperationContext.Current.Host.ChannelDispatchers[0] as ChannelDispatcher;

ServiceThrottle throttle = dispatcher.ServiceThrottle;

Trace.WriteLine(String.Format("MaxConcurrentCalls = {0}",
   throttle.MaxConcurrentCalls));
Trace.WriteLine(String.Format("MaxConcurrentSessions = {0}",
   throttle.MaxConcurrentSessions));
Trace.WriteLine(String.Format("MaxConcurrentInstances = {0}",
   throttle.MaxConcurrentInstances));
```

Quotas

The quota mechanism available through WCF involves controlling the amount of memory used by the service host and the various service implementation objects. The premise behind a DoS attack that is aimed at memory is to find a way to make the processing of the request(s) allocate an inordinately large amount of memory. As additional requests arrive (whether good ones or malicious ones), an *OutOfMemoryException* or a *StackOverflowException* might be raised.

When you apply a quota to a WCF service, the *QuotaExceededException* is raised. However, instead of this exception causing the service to terminate (as the out of memory or stack overflow condition might), the message being processed is simply discarded. The service then processes the next request and carries on.

A number of settings affect the level of quota.

MaxReceivedMessageSize The *MaxReceivedMessageSize* value (along with the other settings associated with quotas) is set on the binding directly. It controls how large a message size can be. The default value is 65,536 bytes, which should be sufficient for most messages. You can set this value through either code or configuration. The following demonstrates a configuration element that will set the value of the maximum message size to 128,000 bytes:

```
<bindings>
    <netTcpBinding>
        <binding name="netTcp"
            maxReceivedMessageSize="128000" />
    </netTcpBinding>
</bindings>
```

CAUTION Setting the *MaxReceivedMessageSize* value

Setting this value (or leaving it to the default) can have a number of unintended consequences. Specifically, if you legitimately have an occasional large message, ensure that you configure the *maxReceivedMessageSize* to accommodate such large messages. Otherwise, the message will be rejected.

You can set this value imperatively also, as demonstrated in the following code sample:

```
' VB
Dim binding As New NetTcpBinding()
binding.MaxReceivedMessageSize = 128000
Dim host As New ServiceHost(GetType(UpdateService), _
   New Uri("net.tcp://localhost:1234/UpdateService"))
host.AddServiceEndpoint("IUpdateService", _
   binding, String.Empty)
host.Open()

// C#
NetTcpBinding binding = new NetTcpBinding();
binding.MaxReceivedMessageSize = 128000;
ServiceHost host = new ServiceHost( typeof(UpdateService),
   new Uri("net.tcp://localhost:1234/UpdateService"));
host.AddServiceEndpoint( "IUpdateService",
   binding, String.Empty);
host.Open();
```

ReaderQuotas The *ReaderQuotas* property of the binding sets limits on the complexity of the messages received by the service. They protect that service from memory-based denial of service by specifying a set of criteria within which all messages must fall. Table 10-1 contains a list of the properties that can be set on the *ReaderQuotas* object and their meanings.

Table 10-1 *ReaderQuotas* Properties

Property	Default	Description
MaxDepth	32	The maximum depth to which the nodes in the message can go. This is like saying that the XML that represents the message can have no more than 32 generations (where a parent node and a child node make up a generation) at the deepest point in the schema.
MaxStringContentLength	8192	The longest that any string value in the message can be. A string value would be the value of an attribute or the value of the inner text for any node.
MaxArrayLength	16384	The maximum number of elements that can appear in a single array.
MaxBytesPerRead	4096	The maximum number of bytes returned by each call to *Read* while the message is processed.
MaxNameTableCharCount	16384	The maximum number of characters that can appear in a table name.

NOTE DoS protection

It might seem a little odd to restrict the number of bytes returned by a *Read* method. However, for an XML file to be processed, the entire starting tag must be loaded into memory. It is a common attack to provide an XML document with an extraordinarily long starting tag. Because this tag would need to be loaded, limiting it is an obvious way to prevent DoS attacks.

Demarcating Operations

Conceptually, a session simply means that the service can determine which client a request is coming from. This enables the service to maintain state between the individual requests. However, there are times when the order in which the operations are executed actually matters, and this requirement calls for an extension to the sessioning mechanism.

The idea of needing to maintain the order in which methods are called might seem a little bizarre. After all, in the vast majority of business applications, the client is quite capable of ensuring this, but in many cases, the ability of the client to dictate the order of operations is not as solid as you might think.

Consider, for example, any HTTP-based binding. Although it would seem that if *MethodA* is invoked before *MethodB*, then in every case, *MethodA* will be executed on the service before *MethodB*. However, suppose *MethodA* and *MethodB* are executed on different threads. Still, isn't it possible to ensure that the two threads are synchronized to the point that the client can guarantee execution order?

The answer is no. When using an HTTP-based binding, there is no guarantee of the order of arrival. Even though the client executes *MethodA* before *MethodB* (on different threads; this doesn't apply to synchronous calls), HTTP will not guarantee that the request associated with *MethodA* will arrive at the service prior to *MethodB*. Unless the service is enlisted in the mechanism to guarantee operation order, no such guarantee can be made.

Consider the following Service contract:

```
' VB
<ServiceContract(SessionMode:=SessionMode.Required)> _
Public Interface IProcessOrders
   <OperationContract> _
   Sub InitializeOrder(customerId As Integer)
   <OperationContract> _
   Sub AddOrderLine(productId As String, _
      Quantity As Integer)
   <OperationContract> _
   Function GetOrderTotal() As Double
   <OperationContract> _
   Function SubmitOrder() As Boolean
End Interface

//C#
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IProcessOrders
{
    [OperationContract]
    void InitializeOrder(int customerId);
    [OperationContract]
    void AddOrderLine(string productId, int quantity);
    [OperationContract]
    double GetOrderTotal();
    [OperationContract]
    bool SubmitOrder();
}
```

The business rules associated with this interface are that the first method to be called has to be *InitializeOrder*. This instantiates an *Order* object and populates the fields with default values. Then the *AddOrderLine* method must be called at least once (although it can be called multiple times). Next, *GetOrderTotal* is called to calculate the order totals. Finally, the *SubmitOrder* method is called. This last method also closes the session.

WCF provides a mechanism that enables contract designers to indicate operations, which cannot be the first or last method, by setting the *IsInitiating* and *IsTerminating* properties on the *OperationContract* attribute. If *IsInitiating* is set to *true* for a method and no session has been established when that method is called, a session is created. If a session already exists, the method is called within that session.

If *IsTerminating* is set to *true* for a method, when the method completes, the session is closed. This is not the same as disposing of the service instance, however. The client still needs to execute the *Close* method on the proxy to close the connection. However, any subsequent methods on this proxy will be rejected with an *InvalidOperationException*.

By using these properties, it is possible to mark the start and end of an operation. The default value for *IsInitiating* is *true*, and the default value for *IsTerminating* is *false*. Because of this, the settings that are required in the sample interface should be set as follows (changes shown in bold):

```
' VB
<ServiceContract(SessionMode:=SessionMode.Required)> _
Public Interface IProcessOrders
   <OperationContract> _
   Sub InitializeOrder(customerId As Integer)
   <OperationContract(IsInitiating:=False)> _
   Sub AddOrderLine(productId As String, _
      Quantity As Integer)
   <OperationContract(IsInitiating:=False)> _
   Function GetOrderTotal() As Double
   <OperationContract(IsInitiating:=False, IsTerminating:=True)> _
   Function SubmitOrder() As Boolean
End Interface

//C#
[ServiceContract(SessionMode = SessionMode.Required)]
public interface IProcessOrders
{
    [OperationContract]
    void InitializeOrder(int customerId);
    [OperationContract(IsInitiating=false)]
    void AddOrderLine(string productId, int quantity);
    [OperationContract(IsInitiating=false)]
    double GetOrderTotal();
    [OperationContract(IsInitiating=false, IsTerminating=true)]
    bool SubmitOrder();
}
```

Consider how these settings will work. Of the four methods, only *InitializeOrder* can start a session. So, if one of the other methods is called prior to *InitializeOrder*, it throws an *InvalidOperationException*. The remaining methods can then be called in any order required, with one exception. If *SubmitOrder* is called because of the *IsTerminating* property, the session is closed.

NOTE Demarcated services must be session aware

To use this demarcating technique, either the service must be session aware (such as having a *Per-Session* instancing mode) or the service must be a singleton.

Instance Deactivation

The details of the issues associated with sessions and service instances are, not surprisingly, more complicated. Consider Figure 10-3, which represents a view closer to reality of a service.

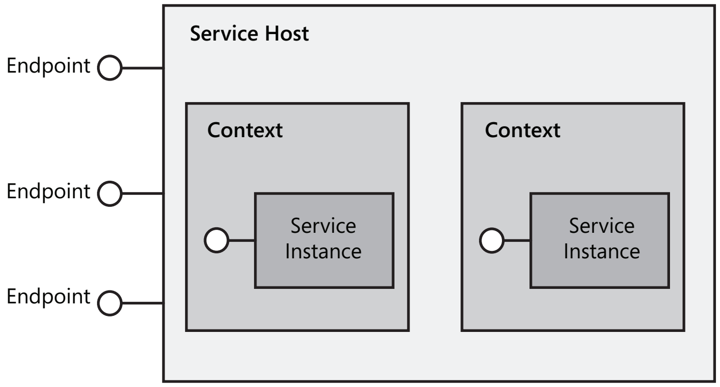


Figure 10-3 How instances and hosts are related

As you can see in Figure 10-3, the service instance is actually loaded into a *Context* object, and the session information routes client messages not to the specific instance but to the context.

When a session is created, the service host creates a new context. This context is terminated when the session ends. This means that the lifetime of the context matches the instance hosted within it by default. WCF enables the developer of the service to separate the two lifetimes.

WCF goes a step further in that you can create a context that has no instance at all. The way to control context deactivation is through the *ReleaseInstanceMode* property of the *Operation-Behavior* attribute.

You can set various values in that *ReleaseInstanceMode* to identify when the service instance should be released in relation to a particular method control. The choices are *BeforeCall*, *After-Call*, *BeforeAndAfterCall*, or *None*.

The default release mode is *None*. This means that the service instance continues to exist as method requests arrive and processes are returned. This is the mode that you have come to expect from a service instance.

If the release mode is set to *BeforeCall*, a new instance is created with the beginning of the call. If a service instance already exists, it is deactivated and the *Dispose* method called on it. The client is blocked while this is going on because it is assumed to be important to have the new service instance available to process the request. This style is normally used when the method allocates a scarce resource, and it must to be certain that any previous use has been cleaned up.

If the release mode is set to *AfterCall*, the current service instance is deactivated and disposed of when the method is completed. This would normally be set when the method is deallocating a scarce resource. The idea is that, after the method is finished, the service instance is disposed of immediately, ensuring that the resource will be available for the next caller.

The last release mode is *BeforeAndAfterCall*. This mode disposes of any existing service instance prior to executing the method and then disposes of the just created service instance after the method is finished. You might recognize this as the same as the per call instance mode. The difference, however, is that this mode can be set on an individual method, so you can configure one method to be (basically) per call instancing, whereas the other methods in the service can use a different instancing model.

You can define the release mode declaratively, using code that looks like the following:

```
' VB
Public Class UpdateService
   Implements IUpdateService

   <OperationBehavior( _
      ReleaseInstanceMode:=ReleaseInstanceMode.BeforeAndAfterCall)> _
   Public Sub Update()
      ' Implementation code goes here
   End Sub

End Class

// C#
public class UpdateService : IUpdateService
{
   [OperationBehavior(ReleaseInstanceMode=ReleaseInstanceMode.BeforeAndAfterCall)]
   public void Update()
   {
      // Implementation code goes here
   }
}
```

You also have the option of making a run-time decision to deactivate the current service instance (when the method is complete). The instance context exposes a *ReleaseServiceInstance* method. When called, the current instance is marked to be deactivated and disposed of after the method is finished. The instance context is part of the operation context, so the way to call this method looks like the following:

```
' VB
OperationContext.Current.InstanceContext.ReleaseServiceInstance()

// C#
OperationContext.Current.InstanceContext.ReleaseServiceInstance();
```

This technique is intended to provide a high level of granularity to optimize the service. However, as is true with many such techniques, the normal course of events doesn't require this level of effort. It is better to design and develop your application using more standard techniques, falling back on these only if performance and scalability goals are not being met.

Lab: Throttling and Demarcation

In this lab, you will work with two separate functions. The first exercise will illustrate some of the throttle configuration you can perform. The effect that some of the settings (such as large request limits, maximum levels in deserialization, and so on) have on incoming requests can be a little challenging to illustrate. As a result, the exercise shows the ones that can be easily demonstrated.

The second exercise will create a service with demarcated operations and demonstrate the exceptions raised when the specified order is violated.

▶ Exercise 1 Throttle WCF Requests

In this first exercise, you will restrict the number of simultaneous instances a service can create. You will use a service similar to the one constructed in the lab for Lesson 1 in this chapter.

1. Navigate to the <*InstallHome*>/Chapter10/Lesson2/Exercise1/<*language*>/Before directory and double-click the Exercise1.sln file to open the solution in Visual Studio.

   The solution consists of two projects. They are as follows:

   ❑ The DemoService project, a simple WCF service library that implements the *ISession* interface. This interface consists of a single method (*GetSessionStatus*) that returns a string indicating the number of times the method has been called within the current service instance.

   ❑ The TestClient project, a Console application that generates a request for the service and displays the result in the Console window.

   You can find the settings for throttling a service in the configuration file for the service.

2. In Solution Explorer, double-click the App.config file in the DemoService project.
3. Locate the behavior named *ThrottleBehavior* in the *serviceBehaviors* element.

   The throttle is set in the *serviceThrottling* element.

4. Set the maximum number of concurrent instances to 2 by adding the following XML to the *behavior* element within the *serviceBehaviors* element:

   ```
   <serviceThrottling maxConcurrentInstances="2" />
   ```

5. Ensure that TestClient is set as the startup project, and then launch the application by pressing F5.
6. When prompted for an instance ID, enter a value of 123 and press Enter.

   This creates the first instance.
7. When prompted for the instance ID again, enter a value of 456 and press Enter.

   This creates the second instance.
8. Finally, when prompted for the instance ID again, enter a value of 789.

   This is the third instance and, rather than displaying a string on the console, it will wait. In fact, it will wait until the timeout value has been exceeded and an exception is thrown.
9. Choose Stop Debugging from the Debug menu to end the application.

▶ Exercise 2 Demarcate Operations

As mentioned, WCF provides some functionality aimed at regulating the order in which operations can be executed. In this exercise, you will configure the service to use this capability. Then you will modify the client to test not only the successful path but also an execution order that would violate the configured order.

1. Navigate to the <*InstallHome*>/Chapter10/Lesson2/Exercise2/<*language*>/Before directory and double-click the Exercise2.sln file to open the solution in Visual Studio.

   The solution consists of two projects. They are as follows:
   - The DemoService project, a simple WCF service library that implements the *ISession* interface. This interface consists of three methods (*FirstMethod*, *GetSessionStatus*, and *LastMethod*), each of which returns a string indicating which method has been called.
   - The TestClient project, a Console application. The application generates a request for the service and displays the result in the Console window.
2. In Solution Explorer, double-click the ISession file.

   You will notice that three methods are defined within the contract.
3. To start, configure *FirstMethod* to be the first operation called, by setting the *IsInitiating* property on the *OperationContract* attribute to *true*.

   The *IsInitiating* property for the other methods in the interface also must be set to *false*, but you do that shortly.
4. Change the declaration of *FirstMethod* (as shown in bold) to look like the following:

```
' VB
<OperationContract(IsInitiating:=True)> _
Function FirstMethod() As String

// C#
[OperationContract(IsInitiating = true)]
string FirstMethod();
```

Now the *IsInitiating* property for the *GetSessionStatus* method must be set to *false*.

5. Change the method declaration (as shown in bold) to the following:

```
' VB
<OperationContract(IsInitiating:=False)> _
Function GetSessionStatus() As String

// C#
[OperationContract(IsInitiating = false)]
string GetSessionStatus();
```

The third method also must have *IsInitiating* set to *false*. However, the intent is for this method to be the last method called. For this reason, the *IsTerminating* property must be set to *true*.

Change the method declaration (as shown in bold) to the following:

```
' VB
<OperationContract(IsInitiating:=False, IsTerminating:=True)> _
Function LastMethod() As String

// C#
[OperationContract(IsInitiating = false, IsTerminating = true)]
string LastMethod();
```

6. In Solution Explorer, double-click Program.cs or Module1.vb.

   Notice the order in which the methods are called. This is what is expected by the configuration in the service.

7. Ensure that TestClient is set as the startup project, and then launch the application by pressing F5.

   Note that the messages appear as expected.

8. End the application.

9. To modify the order of the method calls, move the call to *GetSessionStatus* so that it occurs before the call to *FirstMethod*.

   The body of the *using* block in the *Main* method should look like the following:

```
' VB
Console.WriteLine(proxy.GetSessionStatus())
Console.WriteLine(proxy.FirstMethod())
Console.WriteLine(proxy.LastMethod())

// C#
Console.WriteLine(proxy.GetSessionStatus());
Console.WriteLine(proxy.FirstMethod());
Console.WriteLine(proxy.LastMethod());
```

10. Launch the application by pressing F5.

You will find that an *InvalidOperationException* or an *ActionNotSupportedException* is raised with the *GetSessionStatus* call. The message in the exception indicates that *GetSessionStatus* was invoked before a method in which *IsInitiating* has been set to *true*.

11. Choose Stop Debugging from the Debug menu to end the application.
12. Finally, move the call to *GetSessionStatus* so that it occurs after the call to *LastMethod*.

    The body of the *using* block in the *Main* method should look like the following:

```
' VB
Console.WriteLine(proxy.FirstMethod())
Console.WriteLine(proxy.LastMethod())
Console.WriteLine(proxy.GetSessionStatus())

// C#
Console.WriteLine(proxy.FirstMethod());
Console.WriteLine(proxy.LastMethod());
Console.WriteLine(proxy.GetSessionStatus());
```

13. Launch the application by pressing F5.

    Again, you will find that an *InvalidOperationException* is raised with the *GetSessionStatus* call. This time, the message in the exception indicates that *GetSessionStatus* was invoked after a method in which *IsTerminating* has been set to *true* was called.
14. Choose Stop Debugging from the Debug menu to end the application.

Lesson Summary

- Client endpoint configuration starts from the same address, binding, and contract bases as services do.
- If one of the standard bindings is specified, the default values for that binding are used.
- You define additional binding behaviors through a *behaviorConfiguration* section.
- If the client supports callbacks, you can define a number of client behaviors through the *endpointBehavior* section.
- All of the configuration that can be performed declaratively can also be performed imperatively.
- You can instantiate all the bindings by using the name of a configuration section. Alternatively, you can instantiate the binding separately, assign the desired properties, and then associate it with the proxy.

Lesson Review

You can use the following questions to test your knowledge of the information in Lesson 2, "Working with Instances." The questions are also available on the companion CD if you prefer to review them in electronic form.

NOTE Answers

Answers to these questions and explanations of why each answer choice is correct or incorrect are located in the "Answers" section at the end of the book.

1. Consider the following segment from a configuration file.

```
<behaviors>
   <serviceBehaviors>
      <behavior name="throttlingBehaviort">
         <serviceThrottling maxConcurrentCalls="15"
            maxConcurrentInstances="10"
            maxConcurrentSessions="5"/>
      </behavior>
   </serviceBehaviors>
</behaviors>
```

   Which of the following statements is true?

   A. The service can accept no more than fifteen simultaneous requests.

   B. The service can accept no more than ten simultaneous requests.

   C. The service can accept no more than five simultaneous requests.

   D. There is no limit to the number of simultaneous requests the service can accept.

2. Consider the properties of the OperationContract that demarcate an operation. Which of the following statements is false?

   A. You can specify which method must be the first one called within the service.

   B. You can ensure that no methods in the service can be called after a specific method is called.

   C. You cannot ensure the order of all the methods in a service unless two or fewer methods are exposed.

   D. You can ensure that a particular method will always be called when the service is finished.

Chapter Review

To further practice and reinforce the skills you learned in this chapter, you can:

- Review the chapter summary.
- Review the list of key terms introduced in this chapter.
- Complete the case scenarios. These scenarios set up real-world situations involving the topics of this chapter and ask you to create a solution.
- Complete the suggested practices.
- Take a practice test.

Chapter Summary

- The available instance modes offer a wide range of options. The decision regarding which mode to use should consider both performance and threading issues.
- By using throttling settings and quotas, WCF can help prevent denial of service attacks or simply ensure that servers do not become overloaded with requests.
- The developer can control when service instances are created and destroyed even within the instance modes.

Key Terms

Do you know what these key terms mean? You can check your answers by looking up the terms in the glossary at the end of the book.

- scarce resource
- service instance

Case Scenarios

In the following case scenarios, you will apply what you've learned about instancing modes and working with instances. You can find answers to these questions in the "Answers" section at the end of this book.

Case Scenario 1: Choosing the Appropriate Instancing Mode

Your company has developed a WCF application that will be distributed to your clients. You would like to arrange for a single instance of the service implementation class to be active for each client. The service, therefore, should act as a singleton for each client.

Answer the following question for your manager:

- Which type of instancing should be used in the application?

Case Scenario 2: Protecting Your WCF Application

Your company has developed a WCF application that will be distributed to a large number of clients. Because the service portion of the application is being exposed on a publicly accessible server, you are concerned about the possibility of denial of service attacks. You want to ensure also that, in the case of heavy usage, the client experience is acceptable.

Answer the following questions for your manager.

1. Which type of instancing should be used in the application?
2. Which changes should be made to the throttling settings?
3. Which changes should be made to the quota settings?

Suggested Practices

To help you successfully master the exam objectives presented in this chapter, complete the following tasks.

Working with Service Instances

Create an application to practice instance sharing.

- **Practice** Create a WCF application that uses the IP address from which the request originated to determine which instance context should be used.

WCF Protection

Create an application to practice setting quotas.

- **Practice** Create a WCF application that is configured to reject messages that are over 4 KB in size. Test the application by sending both large and small requests to the service.

Watch a Webcast

Watch a webcast about configuring WCF.

- **Practice** Watch the MSDN webcast, “Instancing Modes,” by Michele Leroux Bustamante, available on the companion CD in the Webcasts folder.

Take a Practice Test

The practice tests on this book's companion CD offer many options. For example, you can test yourself on just one exam objective, or you can test yourself on all the 70-503 certification exam content. You can set up the test so that it closely simulates the experience of taking a certification exam, or you can set it up in study mode so that you can look at the correct answers and explanations after you answer each question.

MORE INFO **Practice tests**

For details about all the practice test options available, see the "How to Use the Practice Tests" section in this book's introduction.

Index