Microsoft®

Windows® Communication Foundation 4

John Sharp
content•master

online book + practice files

Step by Step

Microsoft®

Windows® Communication Foundation 4 Step by Step

John Sharp

Printed and bound in the United States of America.

ISBN: 978-0-735-64556-1

Third Printing: July 2014 with corrections November 2014

Acquisitions and Development Editor: Russell Jones
Production Editor: Kristen Borg
Production Services: Octal Publishing, Inc.
Technical Reviewer: Ashish Ghoda and Kenn Scribner
Indexing: Potomac Indexing, LLC
Cover: Karen Montgomery
Illustrator: Robert Romano

This product is printed digitally on demand.

Contents at a Glance

Table of Contents

Acknowledgments

On the back cover of his book, "Dirk Gently's Holistic Detective Agency," Douglas Adams depicts an invoice presented by Mr. Gently to his client for attempting to find her missing cat. It contains the following items:

Item	Charge
Finding cat (deceased)	£50.00
Detecting and triangulating vectors of interconnectedness of all things	£150.00
Tracing same to beach on Bahamas, fare and accommodation	£1500.00
Struggling on in face of draining skepticism from client, drinks	£327.00
Saving the human race from total extinction	No charge

Douglas Adams's book was published in 1987, but 23 years later I find myself empathizing with Dirk Gently. Happily, my own beloved feline, Ginger, is still very much with us, but in common with many service-oriented developers these days, I spend more and more time searching for solutions that enable me to connect all things together. Clearly, my home office is not quite a beach on the Bahamas, but I do admit to enjoying a decent amount of time sunning myself in the stands at Edgbaston (the home of Warwickshire County Cricket Club) watching batsmen attempting to endanger workmen building the new pavilion with lofted drives over the boundary, while I contemplate how to configure pieces of software to get them to interoperate and communicate correctly. My wife is always a little skeptical of how our jaunts to see how Warwickshire fares against other county cricket teams amounts to "work," but she enjoys the cricket as much as I do, so she does not complain.

In the world of connected solutions, Microsoft Windows Communication Foundation has proved an absolute boon, and although I am yet to be convinced that it has saved the human race from extinction, I have authored papers and even produced a video on how using WCF can help to save your organization (this may be hyperbole, but you know what we technophiles are like when we desperately want to convince management of the need to invest in new software and machinery!). To this end, I always count it an absolute privilege whenever I get the chance to write in depth about fun, new technology; as I mentioned in the previous edition of this book, I thank all at Content Master for allowing me to spend a significant amount of my time doing it.

It would also be very remiss of me not to thank Russell Jones at O'Reilly Media, who badgered me and patiently waited while I found the time to get started on this project as well as for all his support and help in editing and correcting my grammar during the initial drafts of each chapter, and to Bob Russell at Octal Publishing, who had the unrewarding job of having to wade through every chapter seeking out any remaining "British-isms." Additionally, sincere thanks are due to Ashish Ghoda and Kenn Scribner who took on the daunting task of checking the technical accuracy of my work and who provided valuable advice, guidance, and corrections whenever I was wrong (any remaining technical errors in the book are my own, of course). Also, thanks to Lin Joyner at Content Master, who expended a significant effort at great personal risk to her own sanity, for agreeing to test many of the exercises for me.

Finally, I must pay the greatest tribute to my long-suffering family: to Diana, my wife and fellow cricket-watcher, who is never short of advice when seeing a batsman struggle against a short-pitched delivery ("Just hit the ball!"); to James who grew up and left home to go to university while I was writing Chapter 17; and to Francesca who has learned from my wife what it takes to make a truly excellent cup of tea.

And to Ginger—please stop trying to walk across my keyboard as I type.

—John Sharp

Introduction

Microsoft Windows Communication Foundation (WCF), alongside Windows Workflow Foundation (WF) and Windows Presentation Foundation (WPF), has become part of the primary framework for building the next wave of business applications for the Microsoft Windows operating system. WCF provides the underpinning technology driving distributed solutions based on the Microsoft platform; with it, you can build powerful service-oriented systems designed to address connected services and applications. WCF is also an integral technology for building and accessing services running in the cloud under Windows Azure.

You can use WCF to create new services as well as augment and interoperate with many existing services created by using other technologies. When designing distributed applications in the past, you frequently had to choose a specific technology, such as Web services, COM+, Microsoft Message Queue, or .NET Framework Remoting. That choice often had a fundamental impact on the architecture of your solutions. In contrast, WCF provides a consistent model for implementing scalable and extensible systems that employ a variety of technologies with which you can design and architect your solutions without being restricted by a specific connectivity mechanism.

In short, if you are building professional, service-oriented solutions for Windows, you need to learn about WCF.

Who This Book Is For

This book will show you how to build connected applications and services using WCF. If you are involved in designing, building, or deploying applications for the Microsoft Windows operating system, sooner or later you are going to need to become familiar with WCF. This book will give you the initial boost you need to quickly learn many of the techniques required to create systems based on WCF. The book takes a pragmatic approach, covering the concepts and details necessary to enable you to build connected solutions.

Assumptions

To get the most from this book, you should meet the following profile:

- You should be an architect, designer, or developer who will be creating solutions for the Microsoft Windows family of operating systems.
- You should have experience developing applications using Visual Studio and C#.
- You should have a basic understanding of concepts such as transactions, Web services, security, and message queuing.

Finding Your Best Starting Point in This Book

This book is designed to help you build skills in a number of essential areas. It assumes that you are new to WCF and takes you step by step through the fundamental concepts of WCF, feature by feature. The techniques and ideas that you see in one chapter are extended by those in subsequent chapters; therefore, most readers should follow the chapters in sequence and perform each of the exercises. However, if you have specific requirements or are only interested in certain aspects of WCF, you can use the table below to find your best route through this book.

If you are	Follow these steps
New to Web services and distributed applications and need to gain a basic understanding of WCF.	1. Install the code samples as described in the "Code Samples" section of this Introduction. 2. Work through Chapters 1 to 5 sequentially and perform the exercises. 3. Complete Chapters 6 to 18 as your level of experience and interest dictates.
New to Web services and distributed applications and need to learn how to use WCF to implement solutions using common Web services features such as sessions, transactions, and reliable messaging.	1. Install the code samples as described in the "Code Samples" section of this Introduction. 2. Work through Chapters 1 to 10 sequentially and perform the exercises. 3. Complete Chapters 11 to 18 as your level of experience and interest dictates.
Familiar with Web services and distributed applications, and need to learn about WCF quickly, including its advanced features.	1. Install the code samples as described in the "Code Samples" section of this Introduction. 2. Skim the first chapter for an overview of WCF, but perform the exercises. 3. Read Chapter 2 and perform the exercises. 4. Skim Chapter 3. 5. Read Chapters 4 and 5 and complete the exercises. 6. Skim Chapters 6 to 10, performing the exercises that interest you. 7. Complete the remaining chapters and exercises.
Familiar with security concepts but need to understand how to use the security features that WCF provides.	1. Install the code samples as described in the "Code Samples" section of this Introduction. 2. Skim the first three chapters. 3. Read Chapters 4 and 5 and perform the exercises. 4. Skim Chapters 6 to 15. 5. Read Chapter 17 and complete the exercises. 6. Skim Chapter 18.

If you are	Follow these steps
Referencing the book after working through the exercises.	1. Use the index or the Table of Contents to find information about particular subjects. 2. Read the Summary sections at the end of each chapter to find a brief review of the concepts and techniques presented in the chapter.

Conventions and Features in This Book

This book presents information using conventions designed to make the information readable and easy to follow. Before you start, read the following list, which explains conventions you'll see throughout the book and points out helpful features that you might want to use:

- Each exercise is a series of tasks. Each task is presented as a series of numbered steps (1, 2, 3, and so on). A bullet (■) indicates an exercise that has only one step.
- Reader aids labeled "Note" and "Tip" provide additional information or alternative methods for completing a step successfully.
- Reader aids labeled "Important" alert you to information you need to check before continuing.
- Text that you type appears in **bold**.
- A plus sign (+) between two key names means that you must press those keys at the same time. For example, "Press Alt+Tab" means that you hold down the Alt key *while* you press the Tab key.

System Requirements

You'll need the following hardware and software to complete the practice exercises in this book:

- Microsoft Windows 7 Professional, Enterprise, or Ultimate editions.

Note Some of the exercises require you to create local users and security groups. This feature is not available with Windows 7 Home Basic or Home Premium editions.

- Microsoft Visual Studio 2010 Professional, Premium, Ultimate, or Test Professional editions, including SQL Server 2008 Express.

Note The exercises in this book are not intended to work with Visual Studio 2010 Express edition.

- 1.6 GHz or faster 32-bit (x86) or 64-bit (x64) processor.
- 1 GB RAM (32-bit) or 2 GB RAM (64-bit).
- 20 GB available hard disk space (32-bit) or 25 GB (64-bit).
- DirectX 9 graphics device with WDDM 1.0 or higher driver.
- Microsoft mouse (or compatible) pointing device.

Some of the exercises require that you have installed Internet Information Services (IIS) and Message Queuing (MSMQ). You will also need the *AdventureWorks* database provided with the code samples for this book. Download and installation instructions are provided later in this introduction.

Important If you have other tools or services that establish network connections, you may need to temporarily halt them if they use the same ports required by the exercises in this book (alternatively, you can replace the port numbers referenced by the exercises with others of your own choice). For example, some of the exercises reference port 8080. If you have the Apache Web server running on your development computer, it defaults to using port 8080, so you may need to halt or reconfigure this service.

Code Samples

Follow these steps to download and install the code samples and other companion content on your computer so that you can use them with the exercises:

1. Navigate to *http://aka.ms/645561/files*.
2. Click the Downloads tab.

   You'll see instructions for downloading the .zip archive containing the companion content files.
3. Unpack the .zip archive into your Documents folder. This action creates the following folder containing the exercise and solution files for each chapter:

   Microsoft Press\WCF Step By Step

Installing and Configuring Internet Information Services and Microsoft Message Queue

Many of the exercises in this book require you to build WCF services hosted by using Internet Information Services (IIS). You must make sure that you have installed and configured IIS on your computer, and you must have installed ASP.NET version 4.0 with IIS. Additionally, some exercises use Microsoft Message Queue (MSMQ) as the transport for connecting client applications to services, so you must also install the MSMQ Server Core. The following instructions describe how to do this. Note that you require administrative access to your computer to install and configure IIS and MSMQ.

1. Log on to Windows as an account that has Administrator access.
2. On the Windows Start menu, click Control Panel, and then click Programs. In the Programs pane, under Programs And Features, click Turn Windows Features On Or Off.
3. In the Windows Features dialog box, expand Internet Information Services, and then select the following features:
   - ❑ Web Management Tools | IIS Management Console
   - ❑ Web Management Tools | IIS 6 Management Compatibility | IIS 6 Metabase and IIS 6 Configuration Compatibility
   - ❑ World Wide Web Services | Application Development Features | ASP.NET (this will also select .NET Extensibility, ISAPI Extensions, and ISAPI Filters)
   - ❑ World Wide Web Services | Common Http Features | Directory Browsing (Default Document should already be selected)
   - ❑ World Wide Web Services | Security | Basic Authentication and World Wide Web Services | Security | Windows Authentication (Request Filtering should already be selected)
4. Expand Microsoft Message Queue (MSMQ) Server, and then select Microsoft Message Queue (MSMQ) Server Core (do not select the individual items in the Microsoft Message Queue (MSMQ) Server Core folder).
5. Click OK, and then wait for the features to be installed and configured.

Installing ASP.NET Version 4.0

The exercises in this book rely on ASP.NET Version 4.0 being installed and configured with IIS. To do this, perform the following tasks:

1. On the Windows Start menu, click All Programs, click Microsoft Visual Studio 2010, click Visual Studio Tools, right-click Visual Studio Command Prompt (2010), and then click Run As Administrator. In the User Account Control dialog box, click Yes.
2. In the Visual Studio Command Prompt window, type the following command:

   ```
   aspnet_regiis -iru
   ```

3. When the command has completed, leave the Visual Studio Command Prompt window open; you will use it again after installing the *AdventureWorks* database.

Installing and Configuring the *AdventureWorks* Database

The exercises and examples in this book make use of the *AdventureWorks* sample database. If you don't already have this database installed on your computer, a copy of the database installation program is supplied with the companion content for this book. Follow these steps to install and configure the database:

1. Log on to Windows as an account that has administrator access if you have not already done so.
2. Verify that the SQL Server (SQLEXPRESS) service is running.

> **Tip** Start the SQL Configuration Manager utility in the Configuration Tools folder, located in the Microsoft SQL Server 2008 program group. In the left pane, click SQL Server Services. In the right pane, examine the status of the SQL Server (SQLEXPRESS) service. If the status is Stopped, right-click the service, and then click Start. Wait for the status to change to Running, and then close SQL Configuration Manager.

3. Using Windows Explorer, move to the Microsoft Press\WCF Step By Step\Setup folder located within your Documents folder.
4. Double-click the file AdventureWorks2008_SR4.exe. If the User Account Control dialog box appears, click Yes.
5. Wait while the WinZip Self-Extractor tool unzips the installation program.
6. When the SQL Server 2008R2 Database Installer dialog box appears, read the license agreement. If you agree with the license terms, select the I Accept The License Terms check box, and then click Next.

7. On the AdventureWorks 2008 Community Sample Database SR4 page, set the Installation Instance to SQLEXPRESS, select the *AdventureWorks OLTP* database, deselect all other databases, and then click Install.

> **Note** Make sure that you select the AdventureWorks OLTP database and *not* Adventure-Works OLTP 2008. Depending on how you have configured SQL Server, not all databases will be available anyway, and you may see a warning icon against some of these databases. You can ignore these warnings because these databases are not required.

8. On the Installation Execution page, wait while the database is installed and configured, and then click Finish.

8. Return to the Visual Studio Command Prompt window running as Administrator in the Microsoft Press\WCF Step By Step\Setup folder.

9. Type the following command:

```
osql -E -S .\SQLEXPRESS -i aspnet.sql
```

   This command should complete without any errors (it will display a series of prompts, "1> 2> 1> 2> 1> 2> 1> 1> 2> 1> 2> 1>").

> **Note** The script aspnet.sql creates user accounts for the DefaultAppPool and ASP.NET v4.0 applications pools used by IIS and grants these accounts access to the *AdventureWorks* database.

10. Close the Visual Studio Command Prompt window.

Using the Code Samples

Each chapter in this book explains when and how to use any code samples for that chapter. When it's time to use a code sample, the book will list the instructions for how to open the files. The chapters are built around scenarios that simulate real programming projects, so you can easily apply the skills you learn to your own work.

For those of you who like to know all the details, following is a list of the code sample, Visual Studio projects, and solutions, grouped by the folders where you can find them.

> **Important** Many of the exercises require administrative access to your computer. Make sure you perform the exercises using an account that has this level of access.

Solution Folder	Description
Chapter 1	
Completed\ProductsService	This solution gets you started. Creating the ProductsService project leads you through the process of building a simple WCF service hosted by IIS. You can use the service to query and update product information in the *AdventureWorks* database. The ProductsClient project is a console-based WCF client application that connects to the *ProductsService* service. You use this project for testing the WCF service.
Chapter 2	
ProductsClient	This solution is the starting point for the exercises in this chapter. It contains a copy of the completed client application from Chapter 1.
Completed\ProductsClient	This solution contains a version of the client application that connects to the *ProductsService* service by using a TCP connection.
Completed\HostedProducts ServiceHost	This solution contains Windows Presentation Foundation application that provides a host environment for the *ProductsService* service. You use this application to manually start and stop the service. You configure the ProductsClient application to connect to the service hosted by this application by using an HTTP endpoint.
Completed\WindowsProduct Service	This solution contains a Windows Service that hosts the *Products Service* service. You can start and stop the service from the Services applet in the Windows Control Panel. You reconfigure the ProductsClient application to connect to this service by using an endpoint based on the Named Pipe transport.
Chapter 3	
ProductsServiceFault	This solution contains a copy of the ProductsServiceLibrary, Products ServiceHost, and ProductsClient applications from Chapter 2. It is used as a starting point for the exercises in this chapter.
Completed\UntypedProducts ServiceFault	The *ProductsService* service in this solution traps exceptions and reports them back to the client application as untyped SOAP faults, which are caught and handled by the ProductsClient application.
Completed\StronglyTyped ProductsServiceFault	The *ProductsService* service in this solution reports exceptions as typed SOAP faults, defined by using fault contracts. The Products Client application catches these strongly typed SOAP faults as exceptions.
Chapter 4	
ProductsService	This solution contains three projects: the *ProductsService* service, the ProductsServiceHost application, and the ProductsClient. These projects are configured to catch and handle SOAP faults, as described in Chapter 3. This solution forms the starting point for the exercises in this chapter.

Solution Folder	Description
Chapter 4 (continued)	
Completed\NetTcpProducts ServiceWithMessageLevelSecurity	This solution contains an implementation of the *ProductsService* service, the ProductsServiceHost application, and the Products Client applications that applies message-level security over a TCP binding.
Completed\BasicHttpProducts ServiceWithTransportLevel Security	This solution shows how to implement transport-level security over an HTTP binding.
Completed\WS2007Http ProductsServiceWithMessage LevelSecurity	This version of the solution contains a host that implements message-level security over an HTTP binding.
Completed\ProductsService WithBasicAuthentication	This solution contains a version of the *ProductsService* service that implements basic authentication and displays the name of the user calling the *ListProducts* operations. The client application explicitly provides the name and password of the user connecting to the service.
Completed\ProductsService WithWindowsAuthentication	This solution is similar to the previous one, except that the *Products Service* service implements Windows authentication. The credentials for the client application are picked up from the user's login session.
Completed\ProductsService WithAuthorization	The *ProductsService* service in this solution authorizes users according to the Windows security group to which they belong. Users that do not belong to a specified security group are denied access when they attempt to invoke operations.
Chapter 5	
ProductsClient	This folder contains a copy of the client application that is used for testing the various configurations of the *InternetProductsService* service in this chapter.
Completed\ASPNETMembership	This solution contains the *InternetProductsService* service that is deployed to IIS and authenticates users by using the ASP.NET Role Provider rather than Windows security groups.
Completed\ASPNETMemberShip UsingCertificates	The *InternetProductsService* service in this solution uses the ASP. NET Role Provider in conjunction with certificates to authenticate users.
Completed\MutualAuthentication UsingCertificates	The *InternetProductsService* service in this solution uses a certificate to authenticate itself to the client application.
Chapter 6	
ProductsService	This solution contains an amended copy of the ProductsClient, ProductsServiceLibray, and ProductsServiceHost projects from Chapter 4. The service implements message-level security and authenticates users by using Windows tokens. This solution is used as the starting point for the exercises in this chapter.

Solution Folder	Description
Chapter 6 (continued)	
ProductsServiceWithVersioned ServiceContract	This solution contains an implementation of the *ProductsService* service and a client application that provides these two versions of the service contract. It is used by some of the exercises in the second part of the chapter.
Completed\ProductsService WithProtectedOperations	This solution contains a version of the *ProductsService* service in which client applications are required to encrypt and sign request messages for some operations, but only sign requests for others. The proxy class in the ProductsClient application has been updated to encrypt and sign, or just sign messages, as appropriate. The purpose of this solution is to show how changing security requirements for operations can break a service contract.
Completed\ProductsService WithAdditionalBusinessLogic	The *ProductsService* service in this solution contains additional methods. However, because these methods implement internal logic for the service and are not exposed as part of the service contract, they do not require that existing client applications are updated.
Completed\ProductsService WithModifiedServiceContract	This solution contains a version of the *ProductsService* service with an additional operation and a modified service contract. The client application has not been updated, but it still works although it cannot invoke the new operation.
Completed\ProductsService WithVersionedServiceContract	The *ProductsService* service in this solution exposes two versions of the service contract, enabling existing client applications to use the old contract while exposing the additional operation to new client applications.
Completed\ProductsServiceWith AdditionalFieldsInDataContract	This solution shows the effects that modifying a data contract can have on client applications and how you can implement a data contract that supports clients expecting different versions of a data contract.
Chapter 7	
Completed\ShoppingCart	This solution contains a completed version of the initial *Shopping CartService* service that implements shopping cart functionality and a client application that exercises this functionality. This solution is used as the basis for subsequent exercises in this chapter.
Completed\ ShoppingCartContextModes	The *ShoppingCartService* service in this solution demonstrates the use of the Single instance context mode.
Completed\ShoppingCartWith State	The *ShoppingCartService* service in this solution uses the *PerCall* instance context mode and contains code that saves the instance state to XML files.
Completed\ShoppingCart WIthSequencedOperations	This solution shows how to control the sequence in which a client application can invoke operations and control the lifetime of a session.

Solution Folder	Description
Chapter 7 (continued)	
DurableShoppingCart	This solution contains a version of the *ShoppingCartService* service that implements the *PerSession* instance context mode. The solution also contains a GUI client application called Shopping CartGUIClient. This solution is used by exercises that convert the *ShoppingCartService* service into a durable service.
Completed\DurableShoppingCart	This solution contains a completed implementation of the durable version of *ShoppingCartService* service.
Chapter 8	
Completed\ProductsWorkflow	This solution contains a workflow service called *ProductsWork flowService* that retrieves the details of a specified product. The solution also includes a basic console client application to test the service.
Completed\ProductsWorkflow WithFaultHandling	The *ProductsWorkflowService* service in this solution shows how to catch exceptions in a service and send SOAP faults to a client application.
ProductsClient	This version of the client application for the *ProductsWorkflow Service* service that generates SOAP faults.
Completed\ProductsWorkflow WithIISDeployment	This solution shows how to deploy the *ProductsWorkflowService* service to IIS.
Completed\ProductsWorkflow WithCustomHost	This solution demonstrates how to create a custom host application for a workflow service.
Completed\ShoppingCartService	This solution contains a completed version of the *ShoppingCart Service* service implemented as a workflow service.
ShoppingCartGUIClient	This is a copy of the ShoppingCartGUIClient developed in Chapter 7. It is used to test the workflow version of the *ShoppingCartService* service.
Completed\ShoppingCartWith HostAndClient	This solution contains a complete version of the workflow version of the *ShoppingCartService* service, hosted in a custom host application and accessed from the ShoppingCartGUIClient application.
Completed\DurableShopping CartWithHostAndClient	This solution demonstrates how to implement a workflow service as a durable service.
Chapter 9	
ShoppingCartService	This solution contains a copy of the non-durable ShoppingCart Service, ShoppingCartServiceHost, and ShoppingCartClient projects from Chapter 7. It is used as the starting point for the exercises in this chapter.

Solution Folder	Description
Chapter 9 (continued)	
Completed\ShoppingCartService	This solution contains a version of the *ShoppingCartService* service that uses transactions to maintain database integrity. The client application initiates the transactions.
ProductsWorkflow	This solution shows how to implement transactions in a workflow service. It is based on the *ProductsWorkflowService* from Chapter 8. The client application is also based on a workflow.
Chapter 10	
ShoppingCartService	This solution contains a completed version of the Shopping CartService, ShoppingCartHost, and ShoppingCartClient applications from Chapter 9. It is used as the starting point for the exercises in this chapter.
Completed\ShoppingCartService	This solution shows how to configure the *ShoppingCartService* service and ShoppingCartClient application to implement reliable sessions. You run the client application and use the WCF Service Trace Viewer utility to examine the messages passing between the client application and service.
Completed\ShoppingCart ServiceWithReplayDetection	This solution implements a custom binding for the *Shopping CartService* service and ShoppingCartClient applications to support the secure conversation protocol and provide automatic message replay detection.
Chapter 11	
ShoppingCartService	This solution contains a copy of the completed ShoppingCart Service and ShoppingCartClient projects from Chapter 10. The binding and endpoint configuration has been removed from the ShoppingCartHost project. In the exercises in this chapter, you implement these items in code rather than by providing them in a configuration file.
Completed\ShoppingCartService	This solution contains an implementation of the ShoppingCartHost application that programmatically creates a custom binding rather than using one of the WCF predefined bindings.
Completed\ShoppingCart ServiceWithMessageInspector	This solution shows how to create a custom service behavior with which you can inspect request messages sent to the service and response messages that it sends back to client applications.
ProductsService	This solution contains a copy of the *ProductsService* service from Chapter 6. The code and configuration information in the client that connects to the service and sends request messages has been removed. You add code that performs these tasks programmatically in the exercises in this chapter.
Completed\ProductsService	This solution contains a completed version of the ProductsClient application. The client application connects to the service by creating a binding and a channel programmatically rather than using a generated proxy class.

Solution Folder	Description
Chapter 11 (continued)	
ProductsServiceWithManualProxy	This solution shows how to inherit from the *ClientBase* generic abstract class to implement a proxy class that enables a client application to authenticate itself to the service.
SimpleProductsService	This solution contains a stripped down version of the *Products Service* service and client application. You add code to the client application to connect to the service by creating a binding and a channel and then manually create and send a SOAP message to the service.
Completed\SimpleProducts Service	This solution contains a completed version of the client application that manually creates and sends a SOAP message to the service. It receives the response also as a SOAP message.
Chapter 12	
Completed\OneWay	This solution contains a new service called *AdventureWorksAdmin*. The *AdventureWorksAdmin* service exposes an operation that can take significant time to run. It demonstrates how to implement this operation as a one-way operation. You also use this solution to understand the circumstances under which a one-way operation call can block a client application and how to resolve this blocking.
Completed\Async	This solution contains a version of the *AdventureWorksAdmin* service that implements an operation that can execute asynchronously.
MSMQ	This solution contains a copy of the *AdventureWorksAdmin* service that acts as the starting point for the exercises that demonstrate how to use MSMQ as the transport for a WCF service.
Completed\MSMQ	This version of the solution contains a completed implementation of the *AdventureWorksAdmin* service that uses a message queue to receive messages from client applications. You run the client application and service at different times and verify that messages sent by the client application are queued and received when the service runs.
Chapter 13	
Throttling	This solution contains a simplified, non-transactional version of the *ShoppingCartService* service and an extended version of the client application that simulates multiple users connecting to the service. This solution provides the starting point for the exercises showing how to implement throttling.
Completed\Throttling	This solution contains the completed version of the *Shopping CartService* service. You use this service to test the way in which you can configure WCF to conserve resources during periods of heavy load.

Solution Folder	Description
Chapter 13 (continued)	
MTOM	This solution contains a service called *ShoppingCartPhotoService* that retrieves photographic images of products from the *Adventure Works* database. The solution also contains a basic WPF client application that displays images sent by the server. You use this solution to examine how a WCF service transmits messages containing large amounts of binary data.
Completed\MTOM	This version of the service encodes the binary data constituting the image by using the Message Transmission Optimization Mechanism (MTOM). You use this solution to generate message traces that you examine so you can see how the messages are encoded.
Streaming	This solution contains a version of the *ShoppingCartPhotoService* that uses streaming to send the image data to the client application rather than MTOM.
Chapter 14	
ProductsService	This solution contains a copy of the *ProductsService* service hosted by the ASP.NET Development Web Server, and client application that connects to this service. This solution is used as the starting point for the exercises that show how to implement service discovery.
Completed\ProductsServiceWith AdHocDiscovery	The *ProductsService* service in this solution implements ad hoc discovery. It is deployed to IIS. The client application is modified to broadcast a discovery request and retrieve the address of the *ProductsService* service.
Completed\ProductsServiceWith Announcements	In this version of the solution, the *ProductsService* service sends announcement messages when it starts up and shuts down. The client application listens for service announcements and caches the URLs of services as they come on-line. When the client application sends a request, it looks up the URL of the service in this cache.
Completed\ProductsServiceWith ManagedDiscovery	This solution shows how to implement a discovery proxy. The *ProductsService* service sends announcement messages, and the discovery proxy listens for these messages and caches the URLs of services as they come on-line. The client application is modified to retrieve the address of the *ProductsService* from the discovery proxy.
LoadBalancingRouter	This solution contains an amended copy of the durable Shopping CartService, ShoppingCartServiceHost, and ShoppingCartGUIClient from Chapter 7. It is used as the basis for the exercises that show how to implement routing inside a WCF service.
Completed\LoadBalancing Router	This solution contains a WCF service called *ShoppingCartService Router* that acts as a load-balancing router for two instances of the *ShoppingCartService* service. The client application connects to the router, which transparently redirects requests to one instance or the other of the *ShoppingCartService* service.

Solution Folder	Description
Chapter 14 (continued)	
ShoppingCartServiceWithRouter	This solution contains another copy of the durable Shopping CartService, ShoppingCartServiceHost, and ShoppingCartGUI Client from Chapter 7, except that the host application is preconfigured with two HTTP endpoints. This solution provides the starting point for the exercises that show how to implement a WCF routing service.
Completed\ShoppingCartService WithRouter	This solution contains a completed implementation of the WCF routing service
MessageInspector	This solution contains a version of the *MessageInspector* behavior created in Chapter 11. It is used to test the routing service in the ShoppingCartServiceWithRouter project by displaying the details of messages as they are received by the ShoppingCartHost project.
Chapter 15	
Completed\ProductsSales	This solution contains a REST Web service called *ProductsSales Service*, host, and client application that provides access to sales information. The client application tests the *ProductsSalesService* service by sending requests that query the details of orders and customers.
Completed\ProductsSales WithUpdates	This solution contains an updated version of the *ProductsSales Service* service that supports insert, update, and delete operations. The client application is extended to test this functionality.
Completed\SalesData	This solution contains a REST Web service called *SalesData* that also provides access to customer and order information. This service is implemented by using the WCF Data Services template. The SalesDataClient application in this solution uses the client library for the service to connect and send requests to the service.
Chapter 16	
ProductsServiceV3	This solution contains another version of the *ProductsService* service that provides an additional operation that updates the price of a product. The solution also contains a host application, and a client application for testing the service.
Completed\ProductsServiceV3	In this solution, the *ProductsService* service implements a callback contract. The operation that changes the price of a product is reconfigured as a one way operation, and the callback contract enables the service to asynchronously notify the client application of the result of the operation when it has completed.
Completed\ProductsServiceV3 WithEvents	This version of the *ProductsService* service implements an eventing mechanism. Instances of the client applications subscribe to an event, and the service uses a callback contract to notify each subscribing client when the event occurs.

Solution Folder	Description
Chapter 17	
ShoppingCartService	This solution contains a completed version of the *ShoppingCart Service* service, host, and client applications from Chapter 10. It is used as the starting point for the exercises in this chapter.
Completed\ShoppingCartService	The *ShoppingCartService* service in this solution implements claims-based security. The client application uses Windows Card-Space to manage user credentials and send claims information to the service. The service uses verified claims to authorize access to users.
Chapter 18	
ASPNETService	This solution contains a legacy ASP.NET Web site called ASPNET ProductsService. This Web site provides an ASP.NET Web service. The solution also contains a client application that connects to this Web service. Both applications were developed by using the .NET Framework 2.0. The service is used as the basis for exercises that show how to migrate an ASP.NET Web service to WCF and the .NET Framework 4.0.
ProductsServiceHost	This project contains the host application for the WCF service that implements the functionality migrated from the *ASPNETService* Web service.
Completed\ASPNETService	This solution is a version of the *ASPNETProductsService* service that has been migrated to WCF, together with the host and client applications. The code in the client application has not changed, and connects to WCF service in exactly the same way as it did to the original ASP.NET Web service.
Products	This solution contains a legacy COM+ application that you configure to appear to client applications as a WCF service.
ProductsClient	This solution contains an incomplete copy of the ProductsClient application for testing the Products COM+ application by connecting to it as though it was a WCF service. You finish this application during the exercises in this chapter.
Completed\ProductsClient	This solution contains the completed version of the ProductsClient application.

Uninstalling the Code Samples

To remove the code samples from your computer, delete the folder Microsoft Press\WCF Step By Step from your Documents folder by using Windows Explorer.

Your Companion eBook

The eBook edition of this book allows you to:

- Search the full text
- Print
- Copy and paste

To download your eBook, please see the instruction page at the back of this book.

Errata and Book Support

We've made every effort to ensure the accuracy of this book and its companion content. If you do find an error, please report it on our Microsoft Press site:

1. Go to *http://microsoftpressstore.com*.
2. In the Search box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, find the Errata and Updates tab.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We Want to Hear from You

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read *every one* of your comments and ideas. Thanks in advance for your input!

Stay in Touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

This page intentionally left blank

Chapter 3

Making Applications and Services Robust

After completing this chapter, you will be able to:

- Explain how the WCF runtime can convert common language runtime exceptions into SOAP fault messages to transmit exception information from a WCF service.
- Use the *FaultContract* attribute in a service to define strongly typed exceptions as SOAP faults.
- Catch and handle SOAP faults in a client application.
- Describe how to configure a WCF service to propagate information about unanticipated exceptions to client applications for debugging purposes.
- Describe how to detect the *Faulted* state in a WCF service host application and how to recover from this state.
- Explain how to detect and log unrecognized messages sent to a service.

Detecting and handling exceptions is an important part of any professional application. In a complex desktop application, many different situations can raise an exception, ranging from programming errors or events such as unexpected or malformed user input, to failure of one or more hardware components in the computer running the application. In a distributed environment, the scope for exceptions is far greater. This is due to the nature of networks and the fact that, in some cases, neither the application nor the development or administrative staff has control over how the network functions or its maintenance (who is responsible for making sure that the Internet works?). If you factor in the possibility that your application might also access services written by some third party, who may modify or replace the service with a newer version (possibly untested!), or remove the service altogether, then you might begin to wonder whether your distributed applications will ever be able to work reliably.

This chapter shows you how to handle exceptions in client applications and services developed using WCF. You will see how to specify the exceptions that a WCF service can raise and how to propagate information about exceptions from a WCF service to a WCF client. You will also explore the states that a service can be in, how to determine when a host application switches from one state to another, and how to recover a service that has failed. Finally, you will see how to detect unrecognized messages sent to a service by client applications.

CLR Exceptions and SOAP Faults

A WCF service is a managed application that runs by using the .NET Framework common language runtime, or CLR. One important feature of the CLR is the protection that it provides when an error occurs; the CLR can detect many system-level errors and raise an exception if necessary. A managed application can endeavor to catch these exceptions and either attempt some form of recovery or at least fail in a graceful manner, reporting the reason for the exception and providing information that can help a developer to understand the cause of the exception in order to take steps to rectify the situation in the future.

CLR exceptions are specific to the .NET Framework. WCF is intended to build client applications and services that are interoperable with other environments. For example, a Java client application would not understand the format of a CLR exception raised by a WCF service or how to handle it. Part of the SOAP specification describes how to format and send errors in SOAP messages by using SOAP faults. The SOAP specification includes a schema for formatting SOAP faults as XML text and encapsulating them in a SOAP message. A SOAP fault must specify an error code and a text description of the fault (called the "reason"), and it can include other optional pieces of information. Interoperable services built using the WCF should convert .NET Framework exceptions into SOAP faults and follow the SOAP specification for reporting these faults to client applications.

More Info For a detailed description of the format and contents of a SOAP fault, see the World Wide Web Consortium Web site *at http://www.w3.org/TR/soap12-part1/#soapfault*.

Throwing and Catching a SOAP Fault

The WCF library provides the *FaultException* class in the *System.ServiceModel* namespace. If a WCF service throws a *FaultException* object, the WCF runtime generates a SOAP fault message that is sent back to the client application.

In the first set of exercises in this chapter, you will add code to the WCF *ProductsService* service that detects selected problems when accessing the *AdventureWorks* database and uses the *FaultException* class to report these issues back to the client application.

Modify the WCF Service to Throw SOAP Faults

1. Using Visual Studio, open the ProductsServiceFault solution located in the Microsoft Press\WCF Step By Step\Chapter 3\ProductsServiceFault folder (within your Documents folder).

   This solution contains a copy of the ProductsServiceLibrary, ProductsServiceHost, and ProductsClient applications that you created in Chapter 2, "Hosting a WCF Service."

2. In the ProductsServiceLibrary project, open the ProductsService.cs file to display the code for the service in the Code And Text Editor window.

3. Locate the *ListProducts* method in the *ProductsServiceImpl* class.

   You should recall from Chapter 1, "Introducing Windows Communication Foundation," that this method uses the Entity Framework to connect to the *AdventureWorks* database and retrieve the product number of every product in the *Product* table. The product numbers are stored in a list which is returned to the client application. Notice that the exception handler for this method currently ignores all exceptions.

4. Modify the exception handler, as shown in bold in the following:

```
public List<string> ListProducts()
{
    ...
    try
    {
        ...
    }
    catch (Exception e)
    {
        // Edit the Initial Catalog in the connection string in app.config
        // to trigger this exception
        if (e.InnerException is System.Data.SqlClient.SqlException)
        {
            throw new FaultException(
                "Exception accessing database: " +
                e.InnerException.Message, new FaultCode("Connect to database"));
        }
        else
        {
            throw new FaultException(
                "Exception reading product numbers: " +
                e.Message, new FaultCode("Iterate through products"));
        }
    }

    // Return the list of product numbers
    return productsList;
}
```

If an exception occurs, this code examines the cause. If the *InnerException* property of the *Exception* object is a *SqlExecption*, then the exception was caused by the code that accesses the database in the Entity Framework. If the exception is some other type, then the problem must lie in the code that iterates through the list of products retrieved from the database. In both cases, this code creates a new *System.ServiceModel.FaultException* object with the details of the exception and throws it. The operation will stop running and will instead generate a SOAP fault containing a description of the fault and a fault code (which for the purposes in this example simply specifies a name for identification). This SOAP fault is sent back to the client.

Note If you don't create a *FaultCode* object, the WCF runtime will itself automatically generate a *FaultCode* object named "Client" and add it to the SOAP fault sent back to the client.

5. Build the solution.

Modify the WCF Client Application to Catch SOAP Faults

1. In the ProductsClient project, open the file Program.cs to display the code for the client application in the Code And Text Editor window.
2. In the *Main* method, add a *try/catch* block around the code that calls the operations in the WCF service, as shown in bold in the following:

```
static void Main(string[] args)
{
    ...
    // Test the operations in the service

    try
    {
        // Obtain a list of all products
        ...

        // Fetch the details for a specific product
        ...

        // Query the stock level of this product
        ...


        // Modify the stock level of this product
        ...

        // Disconnect from the service
        ...
    }
    catch (FaultException e)
    {
        Console.WriteLine("{0}: {1}", e.Code.Name, e.Reason);
    }

    Console.WriteLine("Press ENTER to finish");
    Console.ReadLine();
}
```

If any of the operations generate a SOAP fault, the WCF runtime on the client creates a *FaultException* object. The catch handler for the *FaultException* object displays the fault code and reason. The name of the fault code is the value specified by the *FaultCode* constructor in the service, and the *Reason* string is the text description of the fault provided by the service.

Test the FaultException Handler

Important Before you perform this exercise, make sure that you still have port 8000 reserved for your application, as described in the exercise "Reserve HTTP Port 8000" in Chapter 2. To reserve this port, open a command prompt as Administrator, and run the following command (replace *UserName* with your Windows user name):

```
netsh http add urlacl url=http://+:8000/ user=UserName
```

1. In the ProductsServiceHost project, edit the App.config file. The <*connectionStrings*> section of this file contains the information used by the Entity Framework to connect to the *AdventureWorks* database.
2. In the <*add*> element of the <*connectionStrings*> section, change the *Initial Catalog* part of the *connectionString* attribute to refer to the **Junk** database, as follows (do not change any other parts of the *connectString* attribute):

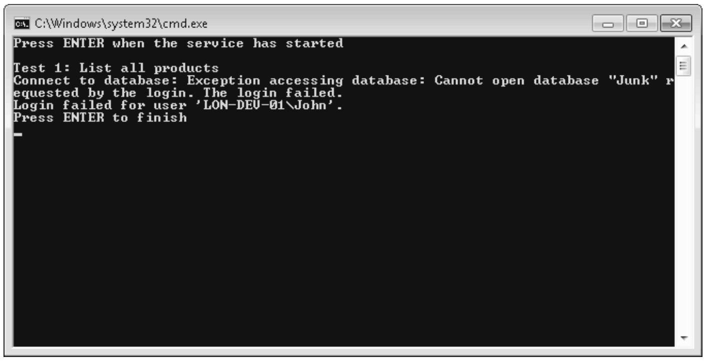
```
<connectionStrings>
  <add ... connectionString="...;Initial Catalog=Junk;..." />
</connectionStrings>
```

3. Build and run the solution without debugging.

   The Products Service Host window and the ProductsClient console window should both start.
4. In the Products Service Host window, click Start.

   If a Windows Security Alert message box appears, click Allow Access.
5. When the service status in the Products Service Host window displays the message "Service Running," press Enter in the ProductsClient console window.

   After a short delay, the ProductsClient application reports an exception similar to the following when performing Test 1 (your message might vary if you are attempting to connect to the database as a different user):



```
Press ENTER when the service has started

Test 1: List all products
Connect to database: Exception accessing database: Cannot open database "Junk" r
equested by the login. The login failed.
Login failed for user 'LON-DEV-01\John'.
Press ENTER to finish
```

The ProductsService service failed when attempting to connect to the database—the SOAP fault code is "Connect to database."

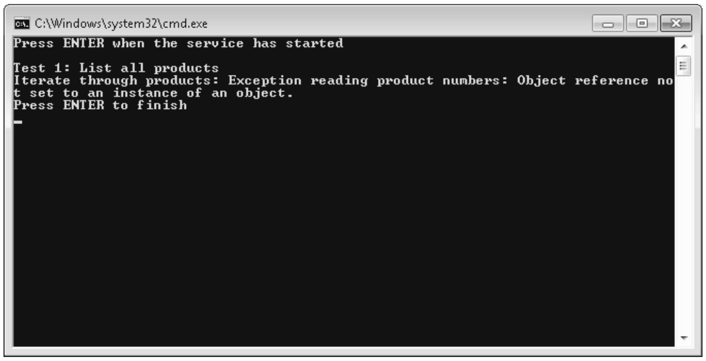
6. Press Enter to close the ProductsClient console.
7. Click Stop in the Products Service Host window, and then close the application.
8. In the App.config file for the ProductsServiceHost application, change the database back to **AdventureWorks** in the *<connectionString>* attribute.
9. In the *ListProducts* method in the ProductsService.cs file, comment out the code that instantiates the *productsList* object and replace it with code that sets this object to **null**. In the body of the *try* block, add a statement that clears the *productsList* object before assigning it the data retrieved from the database, as shown in bold in the following:

```
public List<string> ListProducts()
{
    // Create a list for holding product numbers
    List<string> productsList = null; // new List<string>();

    try
    {
        // Fetch the product number of every product in the database
        var products = from product in database.Products
                       select product.ProductNumber;

        productsList.Clear();
        productsList = products.ToList();
    }
    catch (Exception e)
    {
        ...
    }
    ...
}
```

Note The statement that calls *Clear* is actually redundant and is only used by this exercise to illustrate generating an exception that results in a SOAP fault.

10. Build and run the solution again, without debugging.
11. In the Products Service Host window, click Start.
12. When the service is running press Enter in the ProductsClient console window.

The ProductsClient application reports a different exception when performing Test 1:



```
Press ENTER when the service has started

Test 1: List all products
Iterate through products: Exception reading product numbers: Object reference no
t set to an instance of an object.
Press ENTER to finish
```

This time, the *ProductsService* service failed when clearing the list of products prior to iterating through the items retrieved from the database and adding them to the *productsList* collection (which is now set to *null*)—the SOAP fault code is "Iterate through products"; the reason explains that an object reference was not initialized correctly.

13. Press Enter to close the ProductsClient console window.

14. Click Stop in the Products Service Host form, and then close the application.

Note Do not change the code in the *Listproducts* method back to the correct version just yet.

Using Strongly Typed Faults

Throwing a *FaultException* is very simple but is actually not as useful as it first appears. A client application must examine the *FaultException* object that it catches to determine the cause of the error, so it is not easy to predict what possible exceptions could occur when invoking a WCF service. In such situations, all developers can do is write generalized catch handlers with very limited scope for recovering from specific exceptions. You can think of this as analogous to using the *System.Exception* type to throw and handle exceptions in regular .NET Framework applications. A better solution is to use strongly typed SOAP faults.

In Chapter 1, you saw that a service contract for a WCF service contains a series of operation contracts defining the methods, or operations, that the service implements. A service contract can additionally include information about any faults that might occur when executing an operation. If an operation in a WCF service detects an exception, it can generate a specific SOAP fault message that it can send back to the client application. The SOAP fault message should contain sufficient detail so the user or an administrator can understand the reason for the exception and, if possible, take any necessary corrective action. A client application can use the fault information in the service contract to anticipate faults and provide specific handlers that can catch and process each different fault. These are strongly typed faults.

You specify the possible faults that can occur by using the *FaultContract* attribute in a service contract. This is what you will do in the next set of exercises.

Note You can only apply the *FaultContract* attribute to operations that return a response. You cannot use them with one-way operations. You will learn more about one-way operations in Chapter 12, "Implementing One-Way and Asynchronous Operations."

Use the FaultContract Attribute to Specify the SOAP Faults an Operation Can Throw

1. In the ProductsServiceFault solution, in the ProductsServiceLibrary project, open the IProductsService.cs file.
2. In the IProductsService.cs file, add the following classes shown in bold to the *Products* namespace:

```
namespace Products
{
    // Classes for passing fault information back to client applications
    [DataContract]
    public class SystemFault
    {
        [DataMember]
        public string SystemOperation { get; set; }

        [DataMember]
        public string SystemReason { get; set; }

        [DataMember]
        public string SystemMessage { get; set; }
    }

    [DataContract]
    public class DatabaseFault
    {
        [DataMember]
        public string DbOperation { get; set; }

        [DataMember]
        public string DbReason { get; set; }

        [DataMember]
        public string DbMessage { get; set; }
    }

    // Data contract describing the details of a product
    ...

    // Service contract describing the operations provided by the WCF service
    ...
}
```

These classes define types that you will use for passing the details of SOAP faults as exceptions from a service back to a client. Note that although both classes have a similar shape, you can pass almost any type of information in a SOAP fault; the key point is that the type and its members must be serializable. These two classes use the *Data Contract* and *DataMember* attributes to specify how they should be serialized.

3. Locate the *IProductsService* interface at the end of the IProductsService.cs file.

   Remember that this interface defines the service contract for the *ProductsService*.

4. In the *IProductsService* interface, modify the definition of the *ListProducts* operation, as shown in bold in the following code:

```
// Service contract describing the operations provided by the WCF service
[ServiceContract]
public interface IProductsService
{
    // Get the product number of every product
    [FaultContract(typeof(SystemFault))]
    [FaultContract(typeof(DatabaseFault))]
    [OperationContract]
    List<string> ListProducts();

    // Get the details of a single product
    ...

    // Get the current stock level for a product
    ...

    // Change the stock level for a product
    ...
}
```

   The *FaultContract* attributes indicate that the *ListProducts* method can generate SOAP faults, which a client application should be prepared to handle. The parameter to the *FaultContract* attribute specifies the information that the SOAP fault will contain. In this case, the *ListProducts* operation can generate two types of SOAP faults: one based on the *SystemFault* type, and the other based on the *DatabaseFault* type.

Modify the WCF Service to Throw Strongly Typed Faults

1. In the ProductsService.cs file, locate the *ListProducts* method in the *ProductsServiceImpl* class.

2. Replace the code in the *catch* block that traps *SqlException* exceptions, as shown in bold in the following:

```
public List<string> ListProducts()
{
    ...
    try
    {
        ...
    }
    catch (Exception e)
    {
        // Edit the Initial Catalog in the connection string in app.config
        // to trigger this exception
        if (e.InnerException is System.Data.SqlClient.SqlException)
        {
            DatabaseFault dbf = new DatabaseFault
            {
                DbOperation = "Connect to database",
                DbReason = "Exception accessing database",
                DbMessage = e.InnerException.Message
            };

            throw new FaultException<DatabaseFault>(dbf);
        }
        else
        {
            ...
        }
    }
    ...
}
```

This block creates and populates a *DatabaseFault* object with the details of the exception. The *throw* statement creates a new *FaultException* object based on this *DatabaseFault* object. Note that in this case, the code makes use of the generic *FaultException* class; the type parameter specifies a serializable type with the type-specific details of the exception. At runtime, WCF uses the information in this object to create a SOAP fault message. The *FaultException* constructor is overloaded, and you can optionally specify a reason message and a fault code as well as the *DatabaseFault* object.

3. Replace the code in the *else* part of the *catch* block with that shown in bold, as follows:

```
public List<string> ListProducts()
{
    ...
    try
    {
        ...
    }
    catch (Exception e)
    {
        // Edit the Initial Catalog in the connection string in app.config
        // to trigger this exception
        if (e.InnerException is System.Data.SqlClient.SqlException)
```

```
            {
                ...
            }
            else
            {
                SystemFault sf = new SystemFault
                {
                    SystemOperation = "Iterate through products",
                    SystemReason = "Exception reading product numbers",
                    SystemMessage = e.Message
                };

                throw new FaultException<SystemFault>(sf);
            }
        }
        ...
    }
```

This block of code is similar to the previous *catch* code, except that it creates a *System Fault* object and throws a *FaultException* based on this object. The rationale behind using a different type for the exception is that the kinds of exceptions that could arise when accessing a database are fundamentally different from the exceptions that could occur when reading configuration information. Although not shown in this example, the information returned by a database access exception could be quite different from the information returned by a system exception.

4. Build the solution.

You can now modify the client application to handle the exceptions thrown by the service. However, first you must regenerate the proxy class that the client uses to communicate with the service. The service is not currently running, so you cannot use the *Update Service Reference* feature of Visual Studio. Instead, you will use the *svcutil* utility to generate the proxy class from the assembly containing the *ProductsService* service.

Regenerate the Proxy Class for the WCF Client Application

1. Open a Visual Studio command prompt window and move to the folder \Microsoft Press\WCF Step By Step\Chapter 3\ProductsServiceFault\ProductsServiceLibrary\bin\Debug folder.
2. Run the following command:

```
svcutil ProductsServiceLibrary.dll
```

This command runs the *svcutil* utility to extract the definition of the *ProductsService* service and the other types from the *ProductsServiceLibrary* assembly. It generates the following files:

- **Products.xsd** This is an XML schema file that describe the structure of the, *DatabaseFault*, *SystemFault*, and *ProductData* types. The *svcutil* utility uses the information specified in the data contracts for these types to generate this file. Part of this file, displaying the *DatabaseFault* type, is shown in the following:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="http://schemas.datacontract.org/2004/07/Products"
elementFormDefault="qualified"
targetNamespace="http://schemas.datacontract.org/2004/07/Products"
xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="DatabaseFault">
    <xs:sequence>
      <xs:element minOccurs="0" name="DbMessage" nillable="true"
        type="xs:string" />
      <xs:element minOccurs="0" name="DbOperation" nillable="true"
        type="xs:string" />
      <xs:element minOccurs="0" name="DbReason" nillable="true"
        type="xs:string" />
    </xs:sequence>
  </xs:complexType>
  <xs:element name="DatabaseFault" nillable="true" type="tns:DatabaseFault" />
  ...
</xs:schema>
```

- **Tempuri.org.xsd** This is another XML schema file. This schema describes the messages that a client can send to, or receive from, the *ProductsService* service. You will see later (in the WSDL file for the service) that each operation in the service is defined by a pair of messages; the first message in the pair specifies the message that the client must send to invoke the operation, and the second message specifies the response sent back by the service. This file references the data contract in the Products.xsd file to obtain the description of the *ProductData* type used by the response message of the *GetProduct* operation. The portion of this file that defines the messages for the *ListProducts* and *GetProduct* operations appears as follows:

```
<?xml version="1.0" encoding="utf-8"?>
  ...
  <xs:element name="ListProducts">
    <xs:complexType>
      <xs:sequence />
    </xs:complexType>
  </xs:element>
  <xs:element name="ListProductsResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="ListProductsResult" nillable="true"
         xmlns:q1="http://schemas.microsoft.com/2003/10/Serialization/Arrays"
         type="q1:ArrayOfstring" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="GetProduct">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="productNumber" nillable="true"
         type="xs:string" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="GetProductResponse">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" name="GetProductResult" nillable="true"
         xmlns:q2="http://schemas.datacontract.org/2004/07/Products"
         type="q2:ProductData" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  ...
</xs:schema>
```

Note The name of this file and the namespace of the types in this file are dictated by the *ServiceContract* attribute of the interface implemented by the service. The name *Tempuri.org* is the default namespace. You can change it by specifying the *Namespace* parameter in the *ServiceContract* attribute, like this:

```
[ServiceContract (Namespace="Adventure-Works.com")]
```

- **Schemas.microsoft.com.2003.10.Serialization.Arrays.xsd** This file is another XML schema that describes how to represent an array of strings in a SOAP message. The *ListProducts* operation references this information in the *ListProducts Response* message. The value returned by the *ListProducts* operation is a list of strings containing product numbers. As described in Chapter 1, the .NET Framework generic *List*<> type is serialized as an array when transmitted as part of a SOAP message.
- **Schemas.microsoft.com.2003.10.Serialization.xsd** This XML schema file describes how to represent the primitive types (such as float, int, decimal, and string) in a SOAP message, as well as some other built-in types frequently used when sending SOAP messages.
- **Tempuri.org.wsdl** This file contains the WSDL description of the service, describing how the messages and data contracts are used to implement the operations that a client application can invoke. It references the XML schema files to define the data and messages that implement operations. Notice that the definition of the *ListProducts* operation includes the two fault messages that you defined earlier:

```
...
<wsdl:operation name="ListProducts">
  <soap:operation soapAction="http://tempuri.org/IProductsService/ListProducts"
style="document" />
  <wsdl:input>
    <soap:body use="literal" />
  </wsdl:input>
  <wsdl:output>
    <soap:body use="literal" />
  </wsdl:output>
  <wsdl:fault name="DatabaseFaultFault">
    <soap:fault name="DatabaseFaultFault" use="literal" />
  </wsdl:fault>
  <wsdl:fault name="SystemFaultFault">
    <soap:fault name="SystemFaultFault" use="literal" />
  </wsdl:fault>
</wsdl:operation>
```

You can use the WSDL file and the XML schema files to generate the proxy class.

3. In the Visual Studio command prompt window, type the following command:

```
svcutil /namespace:*,ProductsClient.ProductsService tempuri.org.wsdl *.xsd
```

Note The character between the asterisk (*) and the string *ProductsClient.ProductsService* is a comma (,).

This command runs the *svcutil* utility again, but this time it uses the information in the WSDL file and all the schema files (*.xsd) to generate a C# source file containing a class that can act as a proxy object for the service. The namespace parameter specifies the C# namespace generated for the class (the namespace shown here has been selected to be the same as that generated by Visual Studio in the exercises in Chapter 1, to minimize the changes required to the code in the client application; however, you will need to modify the client configuration file to match this namespace). The *svcutil* utility creates two files:

- **Products.cs** This is the source code for the proxy class.
- **Output.config** This is an example application configuration file that the client application could use to configure the proxy to communicate with the service. By default, the configuration file generates an endpoint definition with the *basicHttpBinding* binding.

Note You can also use the *svcutil* utility to generate a proxy directly from a Web service endpoint rather than generating the metadata from an assembly. This is what Visual Studio does when you use the *Add Service Reference* feature. For more information about the *svcutil* utility, see the "ServiceModel Metadata Utility Tool" on the Microsoft Web site at *http://msdn.microsoft.com/en-us/library/aa347733.aspx*.

4. In Visual Studio, in the ProductsClient project, copy the app.config file and paste the copied file back into the ProductsClient project with the default name, Copy of app.config. This step is necessary because the next step will remove some important information from the app.config file that you will need later.

5. In the ProductsClient project, delete the *ProductsService* service from the Service References folder. As well as removing the service reference, this action also deletes the configuration information for accessing the service from the app.config file, which is why you made a copy of the original version of this file in the previous step.

6. Add the file Products.cs that you have just created to the ProductsClient project. This file is located in the Microsoft Press\WCF Step By Step\Chapter 3\ProductsServiceFault\ProductsServiceLibrary\bin\Debug folder.

7. Delete the app.config file from the ProductsClient project and rename the file Copy of app.config as **app.config**.

8. Open the app.config file in the Code And Text Editor window. Change the contract for both client endpoints to **ProductsClient.ProductsService.IProductsService**, as shown in bold in the following.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <system.serviceModel>
    <bindings>
      ...
    </bindings>
    <client>
      <endpoint address="http://localhost:8000/ProductsService/Service.svc"
          binding="basicHttpBinding"
          bindingConfiguration="BasicHttpBinding_IProductsService"
          contract="ProductsClient.ProductsService.IProductsService"
          name="BasicHttpBinding_IProductsService" />
      <endpoint address="net.tcp://localhost:8080/TcpService" binding="netTcpBinding"
          contract="ProductsClient.ProductsService.IProductsService"
          name="NetTcpBinding_IProductsService" />
    </client>
  </system.serviceModel>

</configuration>
```

This change is necessary as you generated the types for the proxy in the *ProductsClient.ProductsService* namespace when you ran the *svcutil* utility.

Note You could have copied the output.config file generated by the *svcutil* utility to the ProductsClient project and renamed it as app.config rather than preserving and editing the original app.config file. However, although the output.config file specifies the correct type name for the contract attribute of the endpoint, it does not include the address of the service, so you would have had to edit the file and add this information. Additionally, the output.config file only contains the definition of a single *BasicHttpBinding* endpoint, so you would also have needed to add the definition of the *NetTcpBinding* endpoint. It was simpler to modify the existing app.config file!

Modify the WCF Client Application to Catch Strongly Typed Faults

1. In the ProductsClient project, open the Program.cs file in the Code And Text Editor window.
2. Add the following *catch* handlers shown in bold after the *try* block in the *Main* method (leave the existing *FaultException* handler in place as well):

```
static void Main(string[] args)
{
    ...
    try
    {
        ...
    }
    catch (FaultException<SystemFault> sf)
    {
        Console.WriteLine("SystemFault {0}: {1}\n{2}",
            sf.Detail.SystemOperation, sf.Detail.SystemMessage,
            sf.Detail.SystemReason);
    }
    catch (FaultException<DatabaseFault> dbf)
    {
        Console.WriteLine("DatabaseFault {0}: {1}\n{2}",
            dbf.Detail.DbOperation, dbf.Detail.DbMessage,
            dbf.Detail.DbReason);
    }
    catch (FaultException e)
    {
        Console.WriteLine("{0}: {1}", e.Code.Name, e.Reason);
    }
    ...
}
```

These two handlers catch the *SystemFault* and *DatabaseFault* faults. Notice that the fields containing the exception information that are populated by the *ProductsService* (*SystemOperation*, *SystemMessage*, *SystemReason*, *DbOperation*, *DbMessage*, and *DbReason*) are located in the *Detail* field of the *FaultException* object.

Important You must place these two exception handlers before the non-generic *FaultException* handler. The non-generic handler would attempt to catch these exceptions if it occurred first, and the compiler would not let you build the solution.

3. Build and run the solution without debugging.
4. When the Products Service Host window appears, click Start to run the service.

5. When the service has started, in the client application console window, press Enter.

   The code in the *ListProducts* method in the *ProductsService* service still generates a null reference exception. The service throws a *FaultException*, containing a *SystemFault* object, which is serialized as a SOAP fault. The client application catches this fault and displays the details.

6. Press Enter to close the client application. Stop the service and close the Products Service Host window.

7. Edit the ProductsService.cs file in the *ProductsServiceLibrary* project. In the *ListProducts* method, restore the statement that initializes the *productsList* variable back to its original state and remove the code in the *try* block that calls the *Clear* method, as shown in the following:

```
public List<string> ListProducts()
{
    // Create a list for holding product numbers
    List<string> productsList = new List<string>();

    try
    {
        // Connect to the AdventureWorks database by using the Entity Framework
        using (AdventureWorksEntities database = new AdventureWorksEntities())
        {
            // Fetch the product number of every product in the database
            var products = from product in database.Products
                           select product.ProductNumber;

            productsList = products.ToList();
        }
    }
    catch (Exception e)
    {
        ...
    }
    ...
}
```

8. Edit the App.config file in the ProductsServiceHost project by using the Code And Text Editor window, and change the *Initial Catalog* part of the *connectionString* attribute to refer to the **Junk** database, as you did earlier:

```
<connectionStrings>
  <add ... connectionString="...;Initial Catalog=Junk;..." />
</connectionStrings>
```

9. Build and run the solution without debugging.

10. When the Products Service Host window appears, click Start to run the service.

11. When the service has started, in the client application console window, press Enter.

    The application configuration file for the service host application again refers to an invalid database. This "mistake" causes the service to generate a SOAP fault containing a *DatabaseFault* with details of the failure. The ProductsClient application catches this exception in the *FaultException<DatabaseFault>* handler.

12. Press Enter to close the client application. Stop the service and close the Products Service Host window.

13. Correct the *Initial Catalog* attribute in the app.config file for the ProductsServiceHost project and set it back to refer to the *AdventureWorks* database, as follows:

```
<connectionStrings>
  <add ... connectionString="...;Initial Catalog=AdventureWorks;..." />
</connectionStrings>
```

14. Build and run the solution without debugging.

15. In the Products Service Host window, start the service. Press Enter in the client application console window. Verify that the code now runs without any exceptions. Close the client console window, stop the service, and close the Products Service Host window when you have finished.

Reporting Unanticipated Exceptions

Specifying the possible exceptions that a service can throw when performing an operation is an important part of the contract for a service. If you use strongly-typed exceptions, you must specify every exception that an operation can throw in the service contract. If a service throws a strongly-typed exception that is not specified in the service contract, the details of the exception are not propagated to the client—the exception does not form part of the WSDL description of the operation used to generate the client proxy. There will inevitably be situations where it is difficult to anticipate the exceptions that an operation could throw. In these cases, you should catch the exception in the service, and if you need to send it to the client, raise an ordinary (non-generic) *FaultException* as you did in the first set of exercises in this chapter.

While you are developing a WCF service, it can be useful to send information about all exceptions that occur in the service—anticipated or not—to the client application for debugging purposes. You will see how you can achieve this in the next set of exercises.

Modify the WCF Service to Throw an Unanticipated Exception

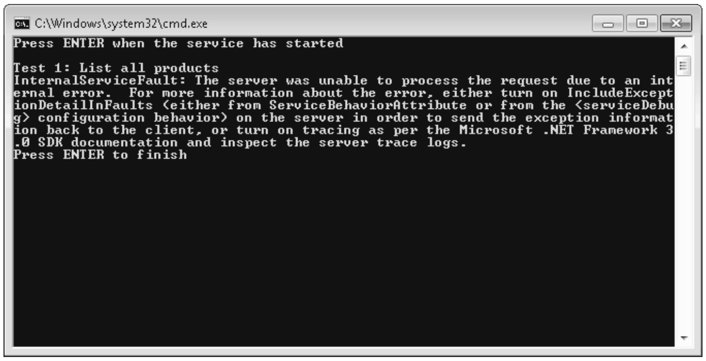
1. In the ProductsServiceFault solution, in the ProductsServiceLibrary project, edit the ProductsService.cs file.
2. Add the following statement (shown in bold) as the first line of code in the *ListProducts* method in the *IProductsImpl* class:

```
public List<string> ListProducts()
{
    int i = 0, j = 0, k = i / j;
    ...
}
```

   This statement will generate a *DivideByZeroException*. Note that the method does not trap this exception, and it is not mentioned in the service contract.

3. Build and run the solution without debugging.
4. In the Products Service Host window, click Start. In the client application console window, press Enter to connect to the service and invoke the *ListProducts* operation.

   The service throws the *DivideByZero* exception. However, the details of the exception are not forwarded to the client application. Instead, the WCF runtime generates a very nondescript SOAP fault that is caught by the *DefaultException* handler in the client:



```
Press ENTER when the service has started

Test 1: List all products
InternalServiceFault: The server was unable to process the request due to an int
ernal error.  For more information about the error, either turn on IncludeExcept
ionDetailInFaults (either from ServiceBehaviorAttribute or from the <serviceDebu
g> configuration behavior) on the server in order to send the exception informat
ion back to the client, or turn on tracing as per the Microsoft .NET Framework 3
.0 SDK documentation and inspect the server trace logs.
Press ENTER to finish
```

   This lack of detail is actually a security feature. If the service provided a complete description of the exception to the client, then, depending on the information provided, a malicious user could glean potentially useful information about the structure of the service and its internal workings.

5. Close the client console window. Stop the service and close the Products Service Host window.

In the next exercise you will configure the host server to provide detailed information about unanticipated exceptions.

Configure the WCF Service to Send Details of Exceptions

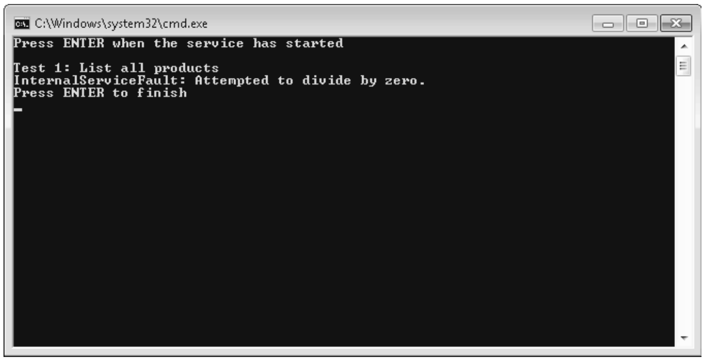
1. In the ProductsServiceHost project, edit the App.config file by using the Code And Text Editor window.
2. In the *<serviceBehaviors>* section, edit the *<serviceDebug>* element in the *<behavior>* section and set the *includeExceptionDetailInFaults* attribute to true:

```
<?xml version="1.0"?>
<configuration>
  ...
  <system.serviceModel>
    ...
    <behaviors>
      <serviceBehaviors>
        <behavior>
          <!-- To avoid disclosing metadata information, set the value below to
               false and remove the metadata endpoint above before deployment -->
          <serviceMetadata httpGetEnabled="false"/>
          <!-- To receive exception details in faults for debugging purposes, set
               the value below to true.  Set to false before deployment to avoid
               disclosing exception information -->
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
    ...
  </system.serviceModel>
</configuration>
```

   Setting the *includeExceptionDetailInFaults* attribute to *true* causes WCF to transmit the full details of exceptions when it generates SOAP faults for unanticipated errors.

3. Build and run the solution with debugging.
4. In the Products Service Host window, click Start. In the client application console window, press Enter.

   The service throws the *DivideByZero* exception. This time, the client is sent specific information about the exception and reports it:



```
Press ENTER when the service has started

Test 1: List all products
InternalServiceFault: Attempted to divide by zero.
Press ENTER to finish
_
```

5. Close the client console window. Stop the service and close the Products Service Host window.
6. In the ProductsServiceLibrary project, edit the ProductsService.cs file.
7. In the *ListProducts* method, comment out the line of code that causes the *DivideBy ZeroException* exception.
8. In the App.config file for the ProductsServiceHost project, set the *includeException DetailInFaults* attribute of the <*serviceDebug*> element to false.
9. Build and run the solution without debugging.
10. In the Products Service Host window, start the service. Press Enter in the client application console window. Verify that the code runs without any exceptions. Close the client console window and the Products Service Host window when you have finished.

The previous exercise used the application configuration file to specify the *serviceDebug* behavior for the service. You can perform the same task by using the *ServiceBehavior* attribute of the class that implements the service, like this:

```
[ServiceBehavior(IncludeExceptionDetailInFaults=true)]
public class ProductsServiceImpl : IProductsService
{
    ...
}
```

However, it is recommended that you enable this behavior only by using the application configuration file. There are a couple of good reasons for this:

- You can turn the behavior on and off in the configuration file without rebuilding the application. You should not deploy an application to a production environment with this behavior enabled, and it is very easy to forget that you have enabled this behavior if you use the *ServiceBehavior* attribute in code.
- If you enable this behavior in code, you *cannot* disable it by altering the application configuration file. Rather more confusingly, if you *disable* this behavior in code, you *can* enable it in the application configuration file. The general rule is that if the *Include ExceptionDetailInFaults* behavior is enabled either in code or in the application configuration file, it will work. It must be disabled in *both* places to turn it off. Keep life simple by only specifying this behavior in one place—the application configuration file.

Managing Exceptions in Service Host Applications

In Chapter 2, you saw how to create a host application for a WCF service and use this application to control the lifecycle of the service. A service host application uses a *ServiceHost* object to instantiate and manage a WCF service. The *ServiceHost* class implements a finite-state machine. A *ServiceHost* object can be in one of a small number of states, and there are well-defined rules that determine how the WCF runtime transitions a *ServiceHost* object from one state to another. Some of these transitions occur as the result of specific method calls, while others are caused by exceptions in the service, in the communications infrastructure, or in the objects implementing the channel stack. A service host application should be prepared to handle these transitions and attempt recovery to ensure that the service is available whenever possible.

ServiceHost States and Transitions

When you instantiate a *ServiceHost* object, it starts in the *Created* state. In this state, you can configure the object; for example, you can use the *AddServiceEndpoint* method to cause the *ServiceHost* object to listen for requests on a particular endpoint. A *ServiceHost* object in this state is not ready to accept requests from client applications.

You start a *ServiceHost* object listening for requests by using the *Open* method (or the *BeginOpen* method if you are using the asynchronous programming model). The *ServiceHost* object moves to the *Opening* state while it creates the channel stacks specified by the bindings for each endpoint and starts the service. If an exception occurs at this point, the object transitions to the *Faulted* state. If the *ServiceHost* object successfully opens the communication channels for the service, it moves to the *Opened* state. Only in this state can the object accept requests from client applications and direct them to the service.

You stop a *ServiceHost* object from listening for client requests by using the *Close* (or *BeginClose*) method. The *ServiceHost* object enters the *Closing* state. Currently running requests are allowed to complete, but clients can no longer send new requests to the service. When all outstanding requests have finished, the *ServiceHost* object moves to the *Closed* state. You can also stop a service by using the *Abort* method. This method closes the service immediately without waiting for the service to finish processing client requests. *Stopping* or aborting the service disposes the service object hosted by the *ServiceHost* object and reclaims any resources it was using. To start the service, you must recreate the *ServiceHost* object with a new instance of the service and then execute the *Open* method to reconstruct the channel stacks and start listening for requests again.

A *ServiceHost* object enters the *Faulted* state either when it fails to open correctly or if it detects an unrecoverable error in a channel used by the *ServiceHost* object to communicate with clients (for example, if some sort of protocol error occurs). When a *ServiceHost* object is in the *Faulted* state, you can examine the properties of the object to try and ascertain the cause of the failure, but you cannot send requests to the service. To recover the service, you should use the *Abort* method to close the service, recreate the *ServiceHost* object, and then execute the *Open* method again. Figure 3-1 summarizes the state transitions for a *ServiceHost* object along with the methods and conditions that cause the object to move between states.

Tip You can determine the current state of a *ServiceHost* object by examining the value of the *State* property.

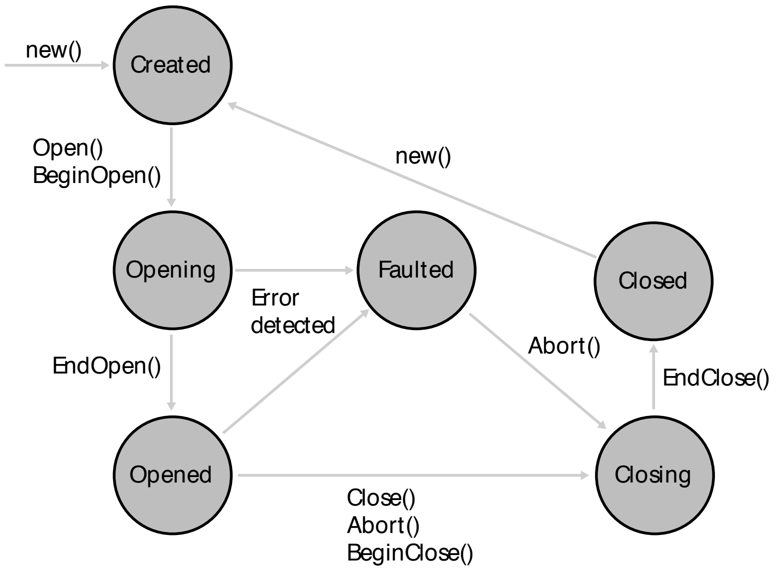


FIGURE 3-1 State transition diagram for the *ServiceHost* class.

Handling Faults in a Host Application

When a *ServiceHost* object moves from one state to another, it can trigger an event. These events were described in Table 2-2 in Chapter 2. From an error-handling perspective, the most important of these is the *Faulted* event, which occurs when a *ServiceHost* object enters the *Faulted* state. You should subscribe to this event, and provide a method that attempts to determine the cause, and then abort and restart the service, like this:

```
// ServiceHost object for hosting a WCF service
ServiceHost productsServiceHost;
productsServiceHost = new ServiceHost(...);
...
// Subscribe to the Faulted event of the productsServiceHost object
productsServiceHost.Faulted += (eventSender, eventArgs) =>
    {
        // FaultHandler method
        // Runs when productsServiceHost enters the Faulted state

        // Examine the properties of the productsServiceHost object
        // and log the reasons for the fault
        ...
        // Abort the service
        productsServiceHost.Abort();

        // Recreate the ServiceHost object
        productsServiceHost = new ServiceHost(...);

        // Start the service
        productsServiceHost.Open();
    };
...
```

Note You can use the *Close* method rather than *Abort* in the fault handler, but a service in the faulted state will not be able to continue processing current requests or receive new ones. Using the *Abort* method to close the service can reduce the time required in the *FaultHandler* method to restart the service.

Handling Unexpected Messages in a Host Application

One other exceptional circumstance that can arise in a host application is an unexpected message from a client. Client applications built by using the WCF library typically communicate with the service by using a proxy object, generated by using the *svcutil* utility. The proxy object provides a strongly-typed interface to the service that specifies the operations the client can request (and therefore the messages that the client sends). It is unlikely that a WCF client using a correctly generated proxy object will send an unexpected message. However, remember that a WCF service is simply a service that accepts SOAP messages, and developers building client applications can use whatever means they see fit for sending these messages. Developers building Java client applications will typically use Java-specific tools and libraries for constructing and sending SOAP messages. WCF also provides a low-level mechanism that allows developers to open a channel to a service, create SOAP messages, and then send them to the service, as shown in this fragment of code:

```
// Create a binding and endpoint to communicate with the ProductsService
BasicHttpBinding binding = new BasicHttpBinding();

EndpointAddress address = new EndpointAddress(
    "http://localhost:8000/ProductsService/Service.svc");

ChannelFactory<IRequestChannel> factory = new
    ChannelFactory<IRequestChannel>(binding, address);

// Connect to the ProductsService service
IRequestChannel channel = factory.CreateChannel();
channel.Open();

// Send a ListProducts request to the service
Message request = Message.CreateMessage(MessageVersion.Soap11,
    "http://tempuri.org/IProductsService/ListProducts");
Message reply = channel.Request(request);

// Process the reply
// (should be a SOAP message with a list of product numbers)
...
// Release resources and close the connection
reply.Close();
channel.Close();
factory.Close();
```

Don't worry too much about the details of this block of code—you will learn more about using Message and Channel objects in Chapter 11, "Programmatically Controlling the Configuration and Communications." The key statement is the line that creates the message sent to the ProductsService service:

```
Message request = Message.CreateMessage(MessageVersion.Soap11,
    "http://tempuri.org/IProductsService/ListProducts");
```

The second parameter to the *CreateMessage* method specifies the action that identifies the message sent to the service. If you recall the earlier discussion in this chapter describing the use of the *svcutil* utility to generate the client proxy, one of the files generated contained the WSDL description of the service. The WSDL description includes the definitions of each of the operations exposed by the service and the messages that an application sends to invoke these operations. Here is part of the WSDL describing the *ListProducts* operation:

```
<wsdl:operation name="ListProducts">
  <wsdl:input wsaw:Action="http://tempuri.org/IProductsService/ListProducts"
message="tns:IProductsService_ListProducts_InputMessage" />
  ...
</wsdl:operation>
```

When the service receives a message identified by the action *http://tempuri.org/IProductsService/ListProducts*, it performs the *ListProducts* operation. If a client application sends a message specifying an action that the service does not recognize, the service host application raises the *UnknownMessageReceived* event. The host application can catch this event and record the unrecognized message, like this:

```
// ServiceHost object for hosting a WCF service
ServiceHost productsServiceHost;
productsServiceHost = new ServiceHost(...);
...
// Subscribe to the UnknownMessageReceived event of the
// productsServiceHost object
productsServiceHost.UnknownMessageReceived += (eventSender, eventArgs) =>
    {
        // UnknownMessageReceived event handler

        // Log the unknown message
        ...
        // Display a message to the administrator
        MessageBox.Show(string.Format(
            "A client attempted to send the message: {0} ",
            eventArgs.Message.Headers.Action));
    };
...
```

There could be a perfectly innocent explanation for a client sending a message such as this, or it could be part of a more concerted attack by a malicious user trying to probe a service and gather information about the operations it supports.

Important The default value for the *httpGetEnabled* property of the *serviceMetadata* behavior is *false*, so unless you explicitly set it to *true*, WCF services do not publish their metadata. It is also worth noting that if you create a WCF service by using Visual Studio, the WCF Service template sets *httpGetEnabled* to *true*. Unless you explicitly need client applications to be able to access the metadata of a service, you should make sure that you reset this property to **false** when you deploy the service to a production environment.

One other possibility is that a WCF client application is using an out-of-date proxy object for sending messages to the service. If a developer modifies the service contract for a WCF service, she might change the messages that the service sends and receives. If any client applications that use the service are not updated, they might send messages that the service no longer understands. Therefore, if you update a service, you should ensure that you retain backward compatibility with existing clients. The same issues can arise with data contracts. You will learn more about how to update data contracts for a WCF service safely in Chapter 6, "Maintaining Service Contracts and Data Contracts."

Summary

In this chapter, you have seen how to use the *FaultException* class to send information about exceptions back to client applications as SOAP faults. You have seen how to use the *Fault Contract* attribute to specify the faults that a service can send and how to catch these faults in a client application. You have also seen how to propagate information about unanticipated exceptions from a service to a client for debugging purposes. You should understand how to make a service host application robust by tracking the states of a service, recovering from faults, and handling unexpected messages sent by client applications.

This page intentionally left blank

Index

About the Author

John Sharp is a Principal Technologist at Content Master Ltd, a technical authoring company based in the United Kingdom. There he researches and develops technical content for technical training courses, seminars, and white papers. Throughout his development career, John has been active in training, developing, and delivering courses. He has conducted training on subjects ranging from UNIX Systems Programming, to SQL Server Administration, to Enterprise Java Development.

John is deeply involved with .NET development, writing courses, building tutorials, and delivering conference presentations covering Visual C#, WCF, SQL Server, Visual J#, ASP.NET, and Windows Server AppFabric. Apart from *Windows Communication Foundation Step By Step*, John has also authored five editions of *Microsoft Visual C# Step By Step*, and *Microsoft Visual J# .NET*, all published by Microsoft Press.