

**1 YEAR UPGRADE**  
BUYER PROTECTION PLAN



DEVELOPING

# .NET Web Services with XML

## Complete Coverage of Using XML in Web Services

- Complete Coverage of Simple Object Access Protocol, Web Services Description Language, and XML Schema Definition
- Hundreds of Configuring & Implementing, Developing & Deploying, Debugging Sidebars, and Web Service FAQs
- Includes Ready-to-Run Applications

David Jorgensen

## s o l u t i o n s @ s y n g r e s s . c o m

With more than 1,500,000 copies of our MCSE, MCSA, CompTIA, and Cisco study guides in print, we continue to look for ways we can better serve the information needs of our readers. One way we do that is by listening.

Readers like yourself have been telling us they want an Internet-based service that would extend and enhance the value of our books. Based on reader feedback and our own strategic plan, we have created a Web site that we hope will exceed your expectations.

**Solutions@syngress.com** is an interactive treasure trove of useful information focusing on our book topics and related technologies. The site offers the following features:

- One-year warranty against content obsolescence due to vendor product upgrades. You can access online updates for any affected chapters.
- “Ask the Author” customer query forms that enable you to post questions to our authors and editors.
- Exclusive monthly mailings in which our experts provide answers to reader queries and clear explanations of complex material.
- Regularly updated links to sites specially selected by our editors for readers desiring additional reliable information on key topics.

Best of all, the book you’re now holding is your key to this amazing site. Just go to [www.syngress.com/solutions](http://www.syngress.com/solutions), and keep this book handy when you register to verify your purchase.

Thank you for giving us the opportunity to serve your needs. And be sure to let us know if there’s anything else we can do to help you get the maximum value from your investment. We’re listening.

[www.syngress.com/solutions](http://www.syngress.com/solutions)



**1 YEAR UPGRADE**  
BUYER PROTECTION PLAN



DEVELOPING

# **.NET** Web Services with XML

David Jorgensen  
DotThatCom.com

Syngress Publishing, Inc., the author(s), and any person or firm involved in the writing, editing, or production (collectively “Makers”) of this book (“the Work”) do not guarantee or warrant the results to be obtained from the Work.

There is no guarantee of any kind, expressed or implied, regarding the Work or its contents. The Work is sold AS IS and WITHOUT WARRANTY. You may have other legal rights, which vary from state to state.

In no event will Makers be liable to you for damages, including any loss of profits, lost savings, or other incidental or consequential damages arising out from the Work or its contents. Because some states do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation may not apply to you.

You should always use reasonable care, including backup and other appropriate precautions, when working with computers, networks, data, and files.

Syngress Media®, Syngress®, “Career Advancement Through Skill Enhancement®,” and “Ask the Author UPDATE®,” are registered trademarks of Syngress Publishing, Inc. “Mission Critical™,” “Hack Proofing®,” and “The Only Way to Stop a Hacker is to Think Like One™” are trademarks of Syngress Publishing, Inc. Brands and product names mentioned in this book are trademarks or service marks of their respective companies.

KEY	SERIAL NUMBER
001	GYV43PK9H7
002	T6CVFQ2UN7
003	5TX3A8J9HF
004	NH89YZ2B76
005	5R33SU8MPT
006	C4E56X6B7N
007	PQ2AKG8D4E
008	J6RD79BKMU
009	7V6FHSW4KP
010	UM39Z5BVF7

PUBLISHED BY  
Syngress Publishing, Inc.  
800 Hingham Street  
Rockland, MA 02370

### **Developing .NET Web Services with XML**

Copyright © 2002 by Syngress Publishing, Inc. All rights reserved. Printed in the United States of America. Except as permitted under the Copyright Act of 1976, no part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of the publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication.

Printed in the United States of America

1 2 3 4 5 6 7 8 9 0

ISBN: 1-928994-81-4

Technical Editor: David Jorgensen  
Acquisitions Editor: Catherine B. Nolan  
Copy Editor: Adrienne Rebello

Cover Designer: Michael Kavish  
Page Layout and Art by: Shannon Tozier  
Indexer: J. Edmund Rush

Distributed by Publishers Group West in the United States and Jaguar Book Group in Canada.



# Acknowledgments

We would like to acknowledge the following people for their kindness and support in making this book possible.

Ralph Troupe, Rhonda St. John, Emlyn Rhodes, and the team at Callisma for their invaluable insight into the challenges of designing, deploying and supporting world-class enterprise networks.

Karen Cross, Lance Tilford, Meaghan Cunningham, Kim Wylie, Harry Kirchner, Kevin Votel, Kent Anderson, Frida Yara, Jon Mayes, John Mesjak, Peg O'Donnell, Sandra Patterson, Betty Redmond, Roy Remer, Ron Shapiro, Patricia Kelly, Andrea Tetrick, Jennifer Pascal, Doug Reil, David Dahl, Janis Carpenter, and Susan Fryer of Publishers Group West for sharing their incredible marketing experience and expertise.

Jacquie Shanahan, AnnHelen Lindeholm, David Burton, Febea Marinetti, and Rosie Moss of Elsevier Science for making certain that our vision remains worldwide in scope.

David Buckland, Wendi Wong, Marie Chieng, Lucy Chong, Leslie Lim, Audrey Gan, and Joseph Chan of Transquest Publishers for the enthusiasm with which they receive our books.

Kwon Sung June at Acorn Publishing for his support.

Jackie Gross, Gayle Voycey, Alexia Penny, Anik Robitaille, Craig Siddall, Darlene Morrow, Iolanda Miller, Jane Mackay, and Marie Skelly at Jackie Gross & Associates for all their help and enthusiasm representing our product in Canada.

Lois Fraser, Connie McMenemy, Shannon Russell, and the rest of the great folks at Jaguar Book Group for their help with distribution of Syngress books in Canada.

A special welcome to the folks at Woodslane in Australia! Thank you to David Scott and everyone there as we start selling Syngress titles through Woodslane in Australia, New Zealand, Papua New Guinea, Fiji Tonga, Solomon Islands, and the Cook Islands.



# Contributors

**Mesbah Ahmed** (PhD and MS, Industrial Engineering) is a Professor of Information Systems at the University of Toledo. In addition to teaching and research, he provides technical consulting and training for IT and manufacturing industries in Ohio and Michigan. His consulting experience includes systems design and implementation projects with Ford Motors, Dana Corporation, Riverside Hospital, Sears, and others. Currently, he provides IT training in the areas of Java Server, XML, and .NET technologies. He teaches graduate level courses in Database Systems, Manufacturing Systems, and Application Development in Distributed and Web Environment. Recently, he received the University of Toledo Outstanding Teaching award, and the College of Business Graduate Teaching Excellence award. His current research interests are in the areas of data warehousing and data mining. He has published many research articles in academic journals such as *Decision Sciences*, *Information & Management*, *Naval Research Logistic Quarterly*, *Journal of Operations Management*, *IIE Transaction*, and *International Journal of Production Research*. He has also presented numerous papers and seminars in many national and international conferences. Mesbah is a contributor to Syngress Publishing's *ASP .NET Developer's Guide* (ISBN: 1-928994-51-2).

**Patrick Coelho** (MCP) is an Instructor at The University of Washington Extension, North Seattle Community College, Puget Sound Center, and Seattle Vocational Institute, where he teaches courses in Web Development (DHTML, ASP, XML, XSLT, C#, and ASP.NET). Patrick is a Co-Founder of DotThatCom.com, a company that provides consulting, online development resources, and internships for students. He is currently working on a .NET solution with contributing author David Jorgensen and nLogix. Patrick holds a bachelor's of Science degree from the University of Washington, Bothell. Patrick lives in Puyallup, WA with his wife, Angela. Patrick is a contributor to Syngress Publishing's

*ASP.NET Developer's Guide* (ISBN: 1-928994-51-2), *C# .NET Web Developer's Guide* (ISBN: 1-928994-50-4), and *.NET Mobile Web Developer's Guide* (ISBN: 1-928994-56-3).

**Adrian Turttschi** (MCSD, MCSE) was formerly employed by KPMG International/CERING as an Integration Architect. He was responsible for integration of components, services and third-party applications of KPMG's next generation global knowledge management and collaboration solution (KnewPro). KnewPro is an application supporting collaboration between geographically and organizationally distributed teams, integrating knowledge sharing and content management, team collaboration, enterprise search, workflow, and legacy data connectivity. Adrian also co-wrote the KnewPro architecture document. Prior to joining KPMG, he worked for EBSCO Publishing as a Software Engineer. Adrian is experienced with Java, C#, Visual Basic, Pascal, and the .NET Framework as a member of the Early Adopter program. Adrian is a contributor to Syngress Publishing's *C# .NET Web Developer's Guide* (ISBN: 1-928994-50-4). He is fluent in English, French, German, and Italian. He has done presentations and has published articles with *XML Journal*, *Nature*, and *Exchange & Outlook Magazine*. Adrian graduated with a master's of Science in Computer Science and Mathematics from the University of Bern, School of Science, Bern, Switzerland and a master's of Arts in Mathematics from Brandeis University, Graduate School of Arts and Sciences, Waltham, MA. He resides in Germany.





# Technical Editor and Contributor

**David Jorgensen** (MCP) David works for Alliance Enterprises, Inc. in Olympia WA, which develops Web-based case management software for social service organizations such as state vocational rehabilitation agencies. His latest project; convert a state agencies data, involved complex SQL Server Data Transformation Packages. David holds a bachelor's degree in Computer Science from St. Martin's College and resides in Puyallup, WA with his wife, Lisa and their two sons, Scott and Jacob. David is a contributor to Syngress Publishing's *C# .NET Web Developer's Guide* (ISBN: 1-928994-50-4), and the *.NET Mobile Web Developer's Guide* (ISBN: 1-928994-56-3).

# Contents

## Configuring & Implementing...

### Setting the Start Page

When testing a Web Service in a project that contains other .aspx or .asmx files, be sure to set the file you are debugging or testing as the Start page, before running. To do this, right-click the filename in the Solution Explorer and select **Set as start page**.

<b>Foreword</b>	<b>xvii</b>
<b>Chapter 1 What Are Web Services?</b>	<b>1</b>
Introduction	2
Understanding Web Services	3
Communication between Servers	8
.asmx Files	10
WSDL	15
Using XML in Web Services	16
An Overview of the System.Web.Services	
Namespace	17
The <i>System.Web.Services.Description</i>	
Namespace	17
The <i>System.Web.Services.Discovery</i>	
Namespace	17
The <i>System.Web.Services.Protocols</i>	
Namespace	18
Type Marshalling	19
Using DataSets	21
Summary	24
Solutions Fast Track	24
Frequently Asked Questions	26
<b>Chapter 2 Introduction to the Microsoft .NET Framework</b>	<b>29</b>
Introduction	30
Obtaining the .NET Framework SDK	31
System Requirements	31
Hardware	32
Operating System	33

Additional Installation Information	33
Locations for Downloading	34
Installing the .NET Framework	34
Common Language Runtime	36
Major Responsibilities of the CLR	36
Safety and Security Checks	37
Class Loading	37
Object Lifetime Management	37
Just In Time (JIT) Compilation	38
Cross-Language Interoperability	38
Structured Exception Handling	39
Assemblies	39
Metadata	40
Enhanced Deployment and Versioning	
Support	41
Managed versus Unmanaged Code	41
Interoperability with Unmanaged Code	42
Namespaces	42
Developing Applications with the .NET Framework	43
Development Platforms for .NET	43
Language Choice	45
Using the Compilers	45
Tools	46
Base Class Libraries	49
Components in the .NET Framework	55
ASP.NET	55
ADO.NET	56
VB.NET	57
C#	59
Windows Forms	60
Web Services	61
Summary	62
Solutions Fast Track	63
Frequently Asked Questions	65

**Answers to Your Frequently Asked Questions**

- Q:** I want to install the .NET Framework SDK on Windows NT 4.0 server, can I do that?
- A:** To install on Windows NT 4.0 server you must have service pack 6a applied.
- Q:** Where can I find the install for ASP.NET?
- A:** ASP.NET ships as part of the .NET Framework SDK.

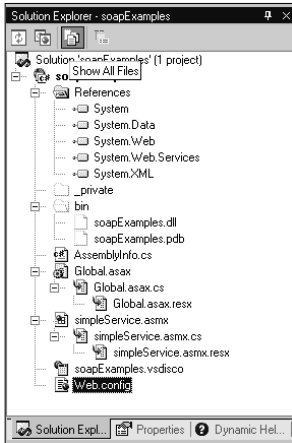
## Components of an XML Document

- **Declaration** Each XML document may have the optional entry `<?xml version="1.0"?>`.
- **Comment** An XML document may contain html-style comments like `<!--Catalog data -->`.
- **Schema or Document Type Definition (DTD)** In certain situations, a schema or DTD may precede the XML document.
- **Elements** An XML document is mostly composed of elements.
- **Root Element** In an XML document, one single main element must contain all other elements inside it. This specific element is often called the root element.
- **Attributes** An attribute is just an additional way to attach a piece of data to an element.

<b>Chapter 3 XML Fundamentals</b>	<b>67</b>
Introduction	68
An Overview of XML	68
What Does an XML Document Look Like?	69
Creating an XML Document	70
Creating an XML Document in VS.NET XML Designer	71
Components of an XML Document	72
Well-Formed XML Documents	75
Schema and Valid XML Documents	76
Structure of an XML Document	80
Processing XML Documents Using .NET	81
Reading and Writing XML Documents	82
Storing and Processing XML Documents	83
Reading and Parsing Using the <i>XmlTextReader</i> Class	84
Parsing an XML Document	85
Navigating through an XML Document to Retrieve Data	87
Writing an XML Document	
Using the <i>XmlTextWriter</i> Class	90
Generating an XML Document Using <i>XmlTextWriter</i>	90
Exploring the XML Document Object Model	93
Navigating through an <i>XmlDocument</i> Object	94
Parsing an XML Document	
Using the <i>XmlDocument</i> Object	95
Using the <i>XmlDataDocument</i> Class	98
Loading an <i>XmlDocument</i> and Retrieving the Values of Certain Nodes	99
Using the Relational View of an <i>XmlDataDocument</i> Object	100
Viewing Multiple Tables of a <i>XmlDataDocument</i> Object	103
Querying XML Data Using <i>XPathDocument</i> and <i>XPathNavigator</i>	107

Using <i>XPathDocument</i> and <i>XPathNavigator</i> Objects	110
Using <i>XPathDocument</i> and <i>XPathNavigator</i> Objects for Document Navigation	112
Transforming an XML Document Using XSLT	115
Transforming an XML Document to an HTML Document	116
Transforming an XML Document into Another XML Document	119
Working with XML and Databases	124
Creating an XML Document from a Database Query	125
Reading an XML Document into a DataSet	127
Summary	129
Solutions Fast Track	129
Frequently Asked Questions	133

**Showing All Files  
through the Solution  
Explorer**



**Chapter 4 Information Exchange Using  
the Simple Object Access Protocol (SOAP) 135**

Introduction	136
The Case for Web Services	136
The Role of SOAP	137
Why SOAP?	138
Why Web Services?	139
Wiring Up Distributed Objects— The SOAP Protocol	139
Creating Your Very First Web Service	139
Running Your Very First Web Service	146
Working with Web Services	159
Passing Complex Data Types	159
Error Handling	162
Malformed SOAP Request	163
Wrong Argument Types	165
Exceptions in Server Code	165
Writing a SOAP Client Application	167
Passing Objects	174
Passing Relational Data (DataSets)	179

## Web Services

Web Services are different from previous technologies used to create distributed systems, such as COM/DCOM, in that:

- They use open standards.
- They were designed from the ground up to work on the Internet, including working well with corporate firewalls.
- They use a “simple” protocol not requiring multiple round trips to the server.
- They purposefully don’t address advanced features such as security or transaction support as part of the protocol specification.

Passing XML Documents	182
SOAP Headers	186
Advanced Web Services	187
Maintaining State	187
State Information in the URL (URL Mangling)	189
State Information in the Http Header (Cookies)	191
State Information in the Http Body (SOAP Header)	194
Security	202
Summary	204
Solutions Fast Track	205
Frequently Asked Questions	207
<b>Chapter 5 WSDL and UDDI</b>	<b>209</b>
Introduction	210
Web Service Standards	211
Describing Web Services—WSDL	211
Discovering Web Services—DISCO	217
Publishing Web Services—UDDI	219
Working with UDDI	220
Summary	228
Solutions Fast Track	229
Frequently Asked Questions	231
<b>Chapter 6 Building an ASP.NET/ADO.NET Shopping Cart with Web Services</b>	<b>233</b>
Introduction	234
Setting Up the Database	234
Setting Up the Table <i>Books</i>	237
Setting Up the Table <i>Categories</i>	237
Setting Up the Table <i>Customer</i>	237
Setting Up the Table <i>Orders</i>	238
Setting Up the Table <i>BookOrders</i>	238
Creating an Access <i>Database</i>	238
SQL Server Database	242
Creating the Stored Procedures	244

**Answers to Your  
Frequently Asked  
Questions**

**Q:** My project has a few different pages in it. Unfortunately, the last page I created is the one that is loaded when I run the project. How do I set the first page to open when I run the project?

**A:** In your **Project Explorer**, right-click the file you want and set it as the **Start Page**.

**Q:** I am working with the *XmlDocument* object in my code-behind page, and I am not getting any IntelliSense. What am I doing wrong?

**A:** Make sure you have included "Using *System.Xml*" in the top section of the page.

Creating the Web Services	250
Overview of the Book Shop Web Services	250
Creating the Data Connection	252
Creating a Web Service	253
Testing a Web Service in ASP.NET	259
Using WSDL Web References	263
Building the Site	264
Site Administration	266
Creating the Administration Login (adminLogin.aspx)	266
Creating the Administrator Page (adminPage.aspx)	268
Retrieving the Data: Creating the <i>getBooks.AllBooks</i> Web Method	268
Displaying the Data: Binding a <i>DataGrid</i> to the <i>DataSet</i>	272
Adding New Books to the Database: Creating the <i>allBooks.addItem</i> Web Method	272
Deleting Books: Deleting from the <i>DataGrid</i> and the Database	272
Updating Book Details: Updating the <i>DataGrid</i> and the Database	273
Creating the addBook Page (addBook.aspx)	274
Customer Administration	275
Creating the Customer Admin Section	275
Creating the <i>loginCustomer</i> Page	275
Creating the <i>updateCustomerInfo</i> Page	276
Creating an ADOCatalog	278
Creating the <i>BookCatalog</i> Class	279
Creating the <i>CreateSummaryTable</i> Method	280
Creating the <i>InitCatalog</i> Method	281
Creating the <i>Catalog</i> Method	281
Creating the <i>catalogItemDetails</i> , <i>catalogRange</i> , and <i>catalogByCategory</i> Methods	281

**Configuring & Implementing...**



**SQL Template Queries**

Previously accessing SQL Templates server-side from within an ASP.NET application would fail to load the XML because the security context of the user would be lost when hopping from IIS to SQL. SQLXML 3.0 solves this problem by allowing server-side access to Template queries by setting *SqlXmlCommand.CommandType = SqlXmlCommandType.TemplateFile*.

Creating the <i>catalogRangeByCategory</i> Method	282
Building an XMLCart	284
Creating the User Interface	287
Creating the start.aspx Page	288
Rendering the Catalog	289
Rendering the Cart	290
Creating the Code	290
Summary	293
Solutions Fast Track	293
Frequently Asked Questions	297
<b>Chapter 7 Building a SQLXML Web Service Application</b>	<b>299</b>
Introduction	300
SQLXML Web Services	301
Developing the TimeTrack Application	301
Creating the Database	302
Creating the Stored Procedures	303
Creating a SQL Server Virtual Directory	305
Enabling Stored Procedures for Soap	310
Creating a Client Application in ASP.NET	313
Consuming the Web Services	317
Summary	333
Solutions Fast Track	334
Frequently Asked Questions	335
<b>Chapter 8 Building a Jokes Web Service</b>	<b>337</b>
Introduction	338
Motivation and Requirements for the Jokes Web Service	338
Functional Application Design	340
Defining Public Methods	340
Defining the Database Schema	341
Defining the Web Service Architecture	342
Security Considerations	344
State Management	345



## Error Handling for the Public Web Methods

The *throwFault* method throws a SOAP fault and ends execution of the Web Service method. But it does a whole lot more:

- The (internal) error code is replaced by a user friendly error message.
- A log entry is written to the *Application* event log.
- The standard SOAP fault XML document is appended with a custom element, called *failReason*, where client applications can find the error message to display to users.

Error Handling	345
Implementing the Jokes Data Repository	345
Installing the Database	346
Creating the Stored Procedures	348
Implementing the Jokes Middle Tier	361
Setting Up the Visual Studio Project	361
Developing the Error Handler	366
Developing the Database Access Component	369
Developing the User Administration Service	371
Adding New Users	371
Checking Existing User Information	376
Adding Moderators	379
Creating the Public Web Methods—Users	381
Error Handling for the Public Web Methods	384
Creating The Public Web Methods—Administrators	386
Testing the Public Web Methods	389
Developing the Jokes Service	390
Best Practices for Returning Highly Structured Data	390
Setting Up Internal Methods to Wrap the Stored Procedure Calls	393
Setting Up Internal Methods to Manage Jokes and Ratings	399
Setting Up Internal Methods to Return Jokes	407
Creating the Public Web Methods	413
Creating a Client Application	423
Some Ideas to Improve the Jokes Web Service	439
Summary	440
Solutions Fast Track	441
Frequently Asked Questions	443

## Index 445

# Foreword

Since its inception in February of 1998, XML has been moving forward through the continued efforts of the World Wide Web Consortium (W3C). At first many developers scoffed at XML, thinking it was just a new way to script. However, those developers, who regularly worked with database management and development soon realized that XML could be a way to provide data between parties without needing to rely on proprietary solutions.

At first, this handful of developers began to incorporate snippets of XML into their desktop applications to store configuration data or as an export file. As time passed, developers began to apply XML to the Internet. Databases began to communicate to each other via XML, and companies were discovering that they had an easier time coping with external database data thanks to XML.

Developers, however, were not the only ones that noticed the possibilities of XML. Microsoft realized the potential of XML, and made it one of the cornerstones of the .NET Framework. .NET aims to bridge the gap between desktop applications and online applications, and facilitate the communication of objects between the two. At the same time, the concept of *Web Services* was being developed.

Broadly speaking, a Web Service is the exposure of a business process over a network. The connotation is generally that XML-based traffic is being moved on a public network (the Internet) via the HTTP protocol. However, Web Services can also be internally useful to an organization, as a mechanism for encapsulating and exposing the business logic inherent in legacy systems. New applications can then utilize this Web Service interface to leverage the complex business logic that has been refined, sometimes for decades, in these legacy systems. This allows for the reuse of systems at the logical level, without regard to physical configuration.

Web Services are a fundamental part of the new .NET Framework. You can group Web Services into two categories: *producers* and *consumers*. A producer Web Service is one that will retrieve a result set, for instance orders from an e-commerce

database, or Jokes; as in Chapter 8 “Building a Jokes Web Service”. A consumer Web Service is one that will use that result set of data and do something with it, as in the shopping cart example in Chapter 6 “Building an ASP.NET/ADO.NET Shopping Cart with Web Services.” In this book we will examine both and provide examples of each. This book focuses on all aspects of Web Services including: XML, SOAP, WSDL, UDDI, and the .NET Framework.

However, Web Services are not limited to just the .NET framework and Microsoft. IBM, SUN, and Oracle will all be players in this fast changing environment. The W3C is still working on revisions for SOAP and XML, which means that this subject matter is continuously evolving.

Taking this a step further, databases will be leveraging their objects as Web Services, as shown in Chapter 7 “Building a SQLXML Web Service.” This is a growing segment of Web Services and XML. Using the universal versatility of XML, an application can transport data across multiple platforms and achieve the same results. Encapsulating this logic in Web Services adds functionality to both the application and the database.

This book assumes that you have prior experience with XML. The code contained in the examples will be in both C# and VB.NET. We will also take a look at SQLXML Web Services along with ADO.NET. You do not have to be a guru to buy this book, but you should possess object oriented programming knowledge to get the most benefit from the code examples. If you have any experience in programming at all you should be able to pick up the content easily. If you need a more fundamental start, I suggest picking one of these two books, *VB.NET Developer's Guide* (ISBN: 1-928994-48-2) and the *C# Web Developer's Guide* (ISBN: 1-928994-50-4). Both of these books offer the proper foundation to properly leverage the knowledge and information in *Developing .NET Web Services with XML*.

—David Jorgensen, MCP

# About the Web Site

The Syngress Solutions Web site ([www.syngress.com/solutions](http://www.syngress.com/solutions)) contains the code files, applications, sample databases, and Web Services that are used throughout *Developing .NET Web Services with XML*.

The code files are located in a chXX directory. For example, the files for Chapter 3 are located in folder ch03. Any further directory structure depends upon the Web Services and applications that are presented within the chapter. Some of the notable pieces of code include those found in Chapters 6 through 8.

In Chapter 6, “Building an ASP.NET/ADO.NET Shopping Cart with Web Services,” readers will find all of the code needed to create a fully functional application for an online bookseller, that is capable of authenticating users and querying a database of both customers and products.

Chapter 7, “Building a SQLXML Web Service Application,” includes all of the code used to create a Web Service for the purpose of project management called “TimeTracker.”

Finally, Chapter 8, “Building a Jokes Web Service,” includes all of the code needed to create a Web Service for interaction between a database of clients, and content. Code for the GUI is included as well.



**Look for this icon to locate the code files that will be included on our Web site.**



## What Are Web Services?

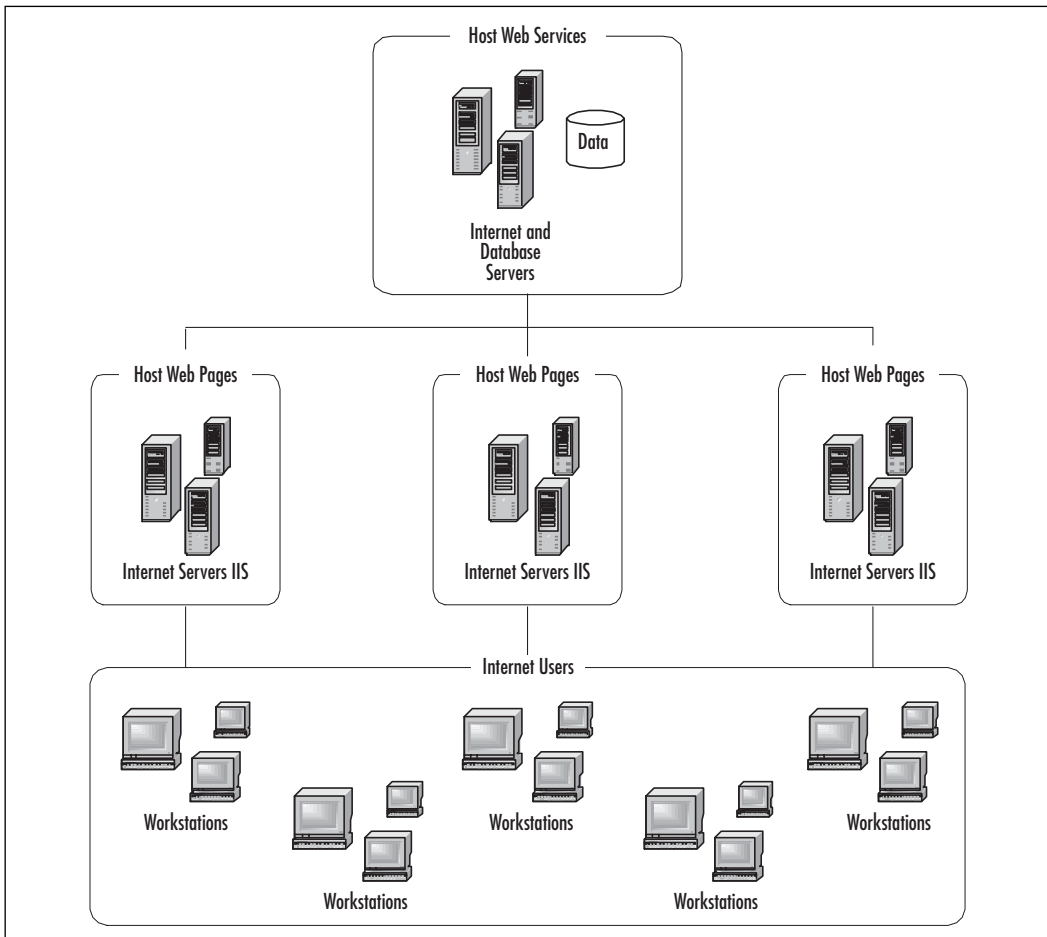
### Solutions in this chapter:

- Understanding Web Services
- Using XML in Web Services
- An Overview of the System.Web.Services Namespace
- Type Marshalling
- Using DataSets
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

# Introduction

Web Services provide a new level of interaction to all kinds of applications. The ability to access and use a remote Web service to perform a function within an application enables programmers to quickly deliver a more sophisticated applications in less time. The programmer no longer has to create and maintain all functions of the application. Reusability is also greatly enhanced by creating multiple Web services that perform functions in multiple applications, thus freeing up time and resources to work on other aspects of specific projects. See Figure 1.1, which shows a graphical representation of this process.

**Figure 1.1** Where Do Web Services Fit In?



In this chapter we will be looking at a simple Hello World Web Service delivered via ASP.NET. This Web Service example can be accessed by any application that can handle Simple Object Access Protocol (SOAP).

Web Services function primarily through XML in order to pass information back and forth through the Hypertext Transfer Protocol (HTTP). Web Services are a vital part of what the .NET Framework offers to programmers. XML-based data transfer is realized, enabling primitive types, enumerations, and even classes to be passed through Web Services to the application performing the request. This brings a whole new level of reusability to an application. XML is the backbone from which the whole Framework is built. The user interface (UI) can be created by applying Extensible Stylesheet Language Transformations (XSLTs) or by loading the data into *DataSets* and binding to Web Controls. Having XML as the intermediary enables new avenues of client design.

## Understanding Web Services

Web Services are objects and methods that can be invoked from any client over HTTP. Web Services are built on the Simple Object Access Protocol (SOAP).

Unlike the Distributed Component Object Model (DCOM) and Common Object Request Broker Architecture (CORBA), SOAP enables messaging over HTTP on port 80 (for most Web servers) and uses a standard means of describing data. SOAP makes it possible to send data and structure easily over the Web. Web Services capitalizes on this protocol to implement object and method messaging. Web Services are easy to create in VS.NET. Here is an ASP.NET *Hello World* class in C#:

```
public class hello
{
    public string HelloWorld()
    {
        return "Hello World";
    }
}
```

This class describes a *hello* object that has one method, *HelloWorld*. When called, this method will return data of type *string*. To convert this to a Web Service method, we simply have to add one line of code:

```
public class hello
{
```



```

        [WebMethod]
        public string HelloWorld()
        {
            return "Hello World";
        }
    }
}

```

A little bit more code is involved to make this a method of a Web Service. This is the code that VS.NET auto-generates when we create a new .asmx page, along with our *Hello World* method:

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
namespace Hello
{
    public class Hello : System.Web.Services.WebService
    {
        public Hello()
        {
            InitializeComponent();
        }
        private void InitializeComponent()
        {
        }
        protected override void Dispose( bool disposing )
        {
        }
        [WebMethod]
        public string HelloWorld()
        {
            return "Hello World";
        }
    }
}

```

```
}  
}
```

You can quickly create this class in VS.NET by creating and opening a C# Web Application project or Web Service project and adding a new WebService page.

If you prefer, similar code could be written to create a VB.NET Service:

```
Imports System.Web.Services  
  
Public Class Service1  
    Inherits System.Web.Services.WebService  
    <WebMethod(>> Public Function HelloWorld() As String  
        HelloWorld = "Hello World"  
    End Function  
  
End Class
```

## Debugging...

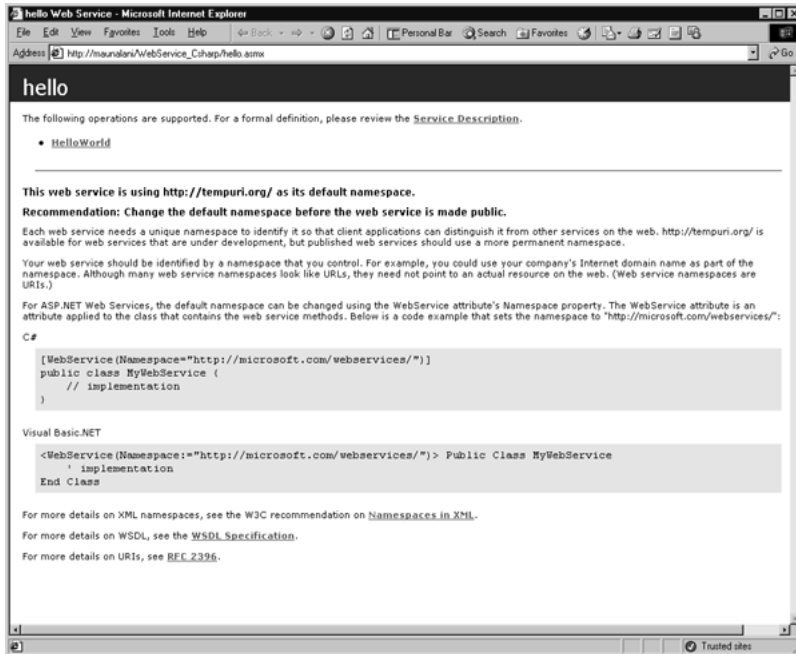
### Setting the Start Page

When testing a Web service in a project that contains other .aspx or .asmx files, be sure to set the file you are debugging/testing to be the Start page, before running. To do this, right-click the filename in the **Solution Explorer** and select **Set as start page**.

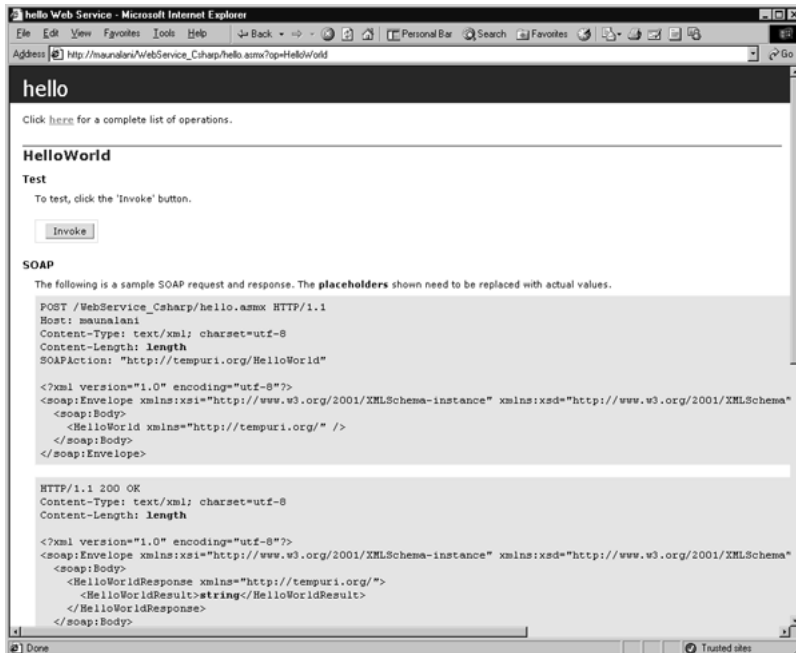
To run this sample in VS.NET, simply press **F5**. It will take a few moments to build and compile. When that phase is complete, you should see the Hello service screen shown in Figure 1.2.

The top line on the screen states that the operations listed below it are supported. This is followed by a bulleted list of links to each of the Web methods that belong to the Web service. In our case, we created only one Web method, *HelloWorld*. If you click the link **HelloWorld**, you will be taken to that service's description page (see Figure 1.3).

## Figure 1.2 Hello Service

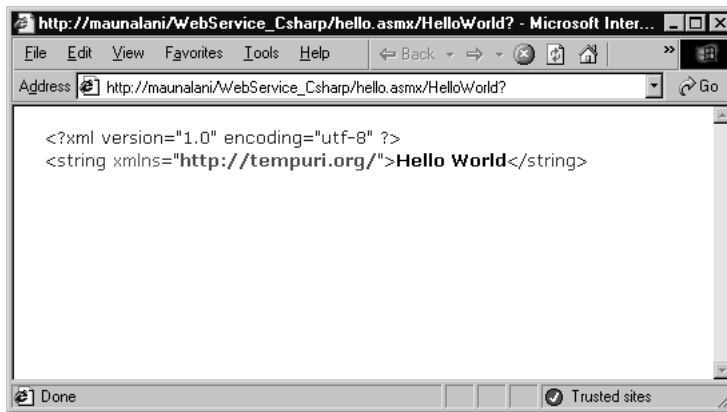


## Figure 1.3 HelloWorld Service Description Page



To test our Hello Web Services *HelloWorld* Web method, simply click the **Invoke** button and our method will be called. Recalling our method returns the string “hello world”; the result is returned in an XML wrapper (see Figure 1.4).

**Figure 1.4** Results from Invoking the *HelloWorld* Web Method



Note that the XML node reflects the datatype of the method’s return value, *string*. This XML message is received and converted to the string “Hello World”. This means that any variable (of type *string*) in our code can be assigned to the result of our Web method.

## Developing & Deploying...

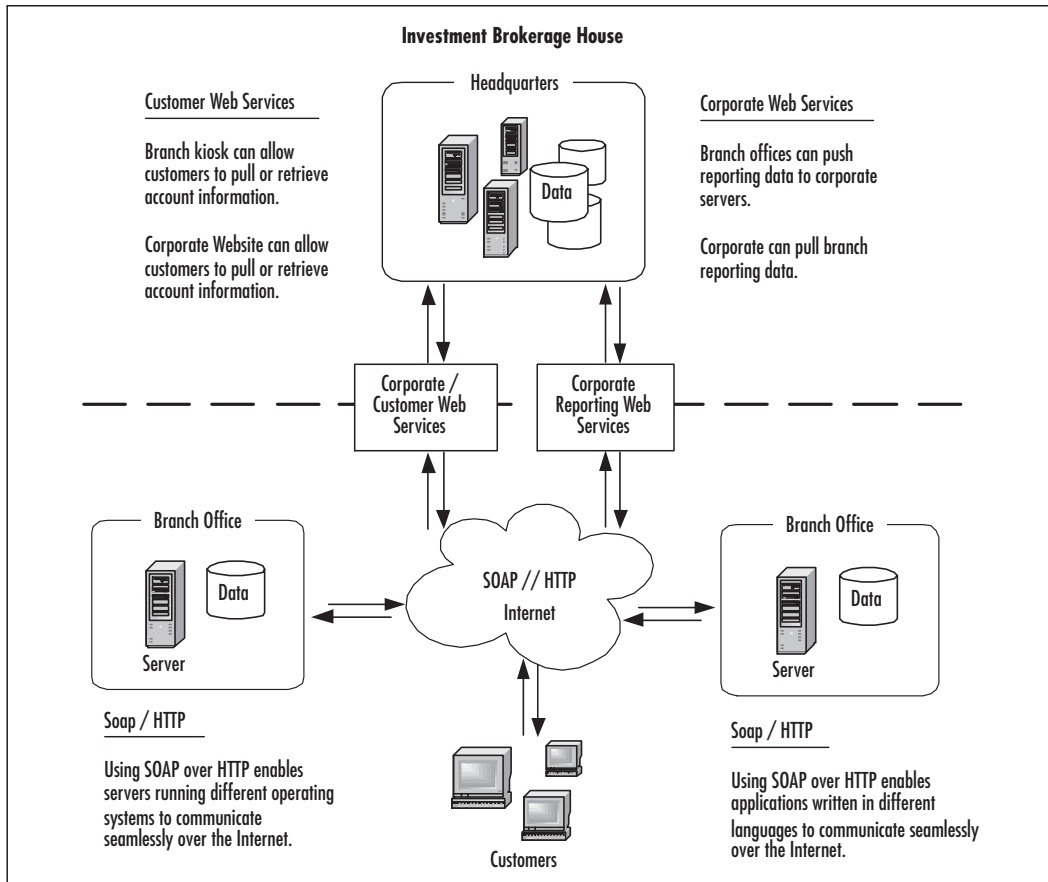
### Building and Compiling

If you have experience programming in C/C++ or Java, you will be familiar with the building and compiling steps. If you are a Web Developer who hasn’t really played with a compiled language before, these steps will be new to you. Think of it as the phase in which the compiler gets all your code together and checks it for unassigned variables, variable type mismatches, and other syntactic errors. In this phase, it also converts your code into the Common Language Runtimes (CLR) Intermediate Language (IL), and then into machine language. This will allow the code to run faster and more efficiently than uncompiled script. After this phase completes, the code is run in the Browser. So, testing Web page output may seem to take longer in the .NET environment.

## Communication between Servers

The concept of sending messages between servers or remotely calling functions is not new. Technologies such as DCOM and CORBA are well-known proprietary protocols that have been in use for years. What is new is the use of a standard protocol to transfer messages over HTTP, that protocol is SOAP. SOAP makes it possible for applications written in different languages running on different platforms to make remote procedure calls (RPC) effectively, even through firewalls. DCOM doesn't use port 80, which is reserved for HTTP traffic; this causes DCOM calls to be blocked by firewalls. SOAP calls use port 80, which makes it possible to call procedures that exist behind firewalls. Figure 1.5 shows a high level overview of how Web Services can be used, both for customer interactions with a company from multiple client types as well as for internal company data gathering and reporting between all company servers, including legacy systems.

**Figure 1.5** Overview: Where Does Web Services Fit in?



In ASP.NET, Web Services and their methods are defined in pages with the .asmx extension. When we create Web Services, the .NET Framework generates a Web Services Description Language (WSDL) file on the server hosting the Service; this WSDL file describes the Web Service interface. On the Web server that hosts our .aspx pages, VS.NET generates a WSDL proxy when we click **Add Web reference** in the **Solutions Explorer** and select the server and Service (see Figure 1.6).

**Figure 1.6** Overview Where WSDL and WSDL Proxies Fit into the Internet User Page Request Process

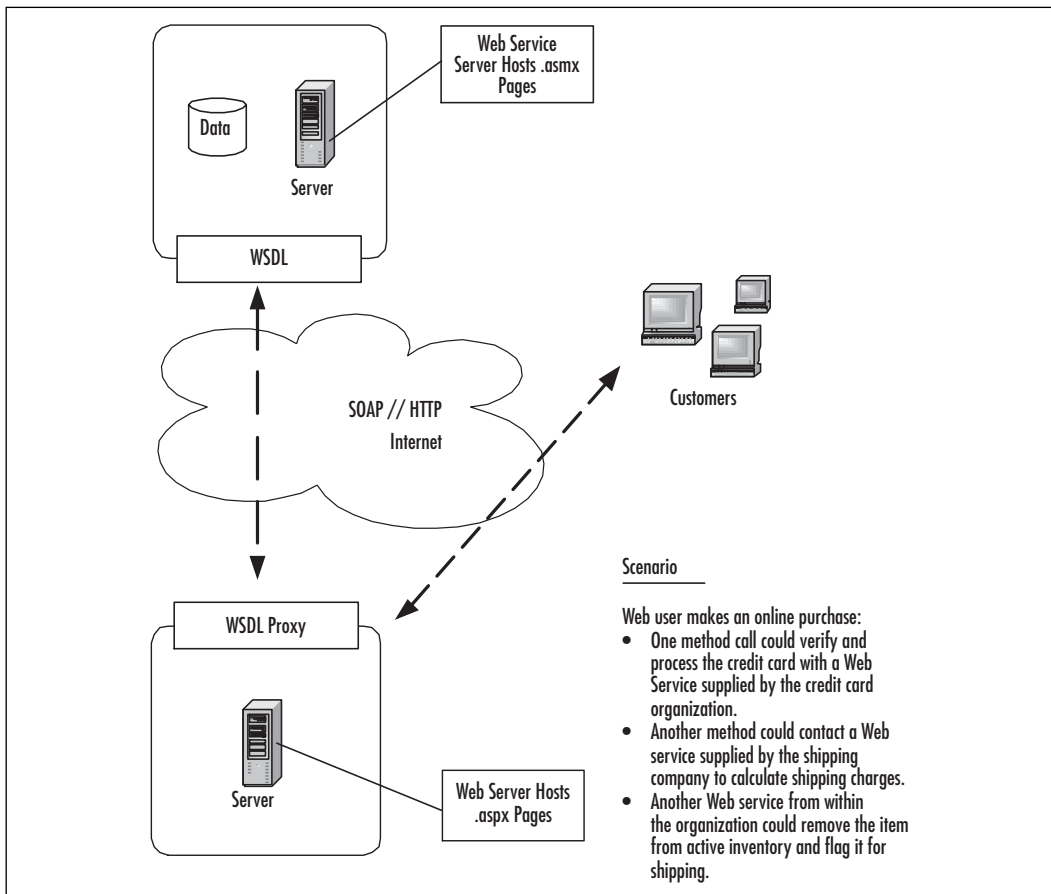
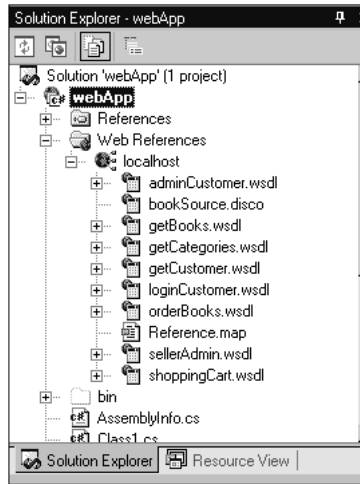


Figure 1.7 shows a Web reference for “localhost” and the WSDL proxies for each Web Service that exists on that server.

Figure 1.7 Web References in VS.NET's Solution Explorer Window

**NOTE**

A single application hosted on the Web server may access several Web Services residing on different servers. Likewise, many Web servers may access one Web Service.

**.asmx Files**

ASP.NET uses the .asmx file extension for defining ASP.NET Web Services. The code-behind pages are .asmx.cs and .asmx.vb for C# and VB.NET, respectively.

**Migrating...****What Is the Difference between .asmx and .aspx?**

In ASP, we have the .asp extension to denote an Active Server Page. When IIS sees this extension, it knows it has some extra processing to do. This is the same with ASP.NET, except that we have two new extensions, .aspx and .asmx.

Continued

Lets do a quick comparison:

- Both file types have a template, which includes references to the primary namespaces.
- .aspx pages have references to System.Drawing since their purpose is to generate a user interface.
- .asmx pages have references to System.Web.Services since their purpose is to generate an interface for external programs.
- You can add UI components and DataConnections to an .aspx page.
- You can add Server and DataConnections to an .asmx page.
- .aspx pages usually begin with an @Page directive to designate: this is a WebForm.
- .asmx pages usually begin with an @WebService directive to designate: this is a Web Service.
- Using the wrong @ directive with the wrong type of file extension will generate an error.

While the client for an .aspx page is the Web browser, the client for an .asmx file is the Web server. Since they are used as programming interfaces and not directly utilized by the Web user, .asmx files should not contain any UI. To get a better understanding of how this all works, lets create an .aspx page that calls our “Hello” service.

1. In the **Solutions Explorer**, right-click the project name.
2. Select **Add | New Item**.
3. Select **Web Form**. Name the file **helloPage.aspx**.
4. While in design view, open the toolbox and drag onto the page a label and a button control from the selection of Web Forms (see Figure 1.8).  
While still in design mode double-click the new button. This will generate event code in the code behind page (see Figure 1.9).
5. Right-click **References** in the **Solution Explorer** and select **Add Web Reference**. This is basically a graphical user interface (GUI) for the WSDL.exe command line utility.



Figure 1.8 Adding a Web Form Control to an .aspx Page

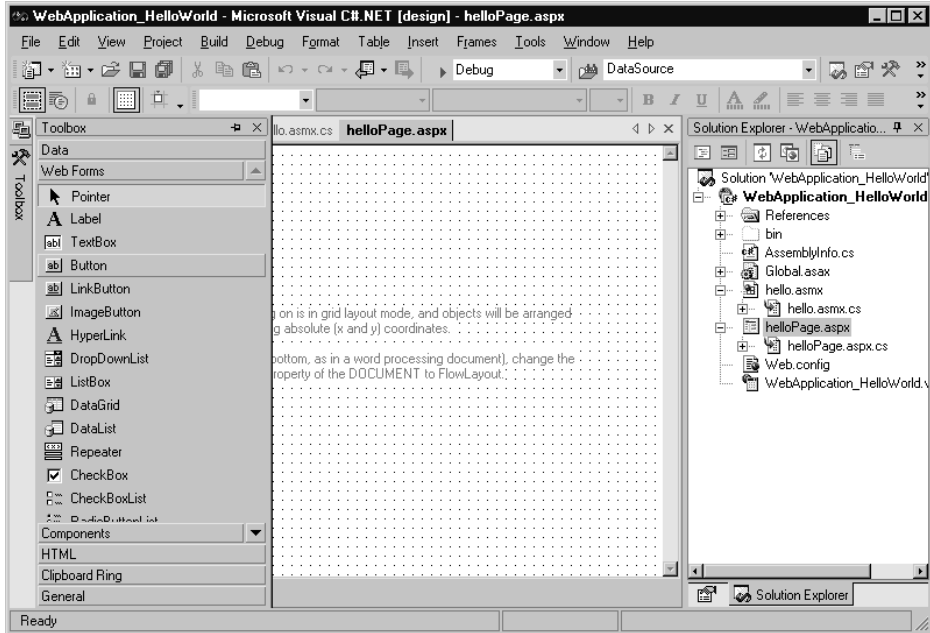
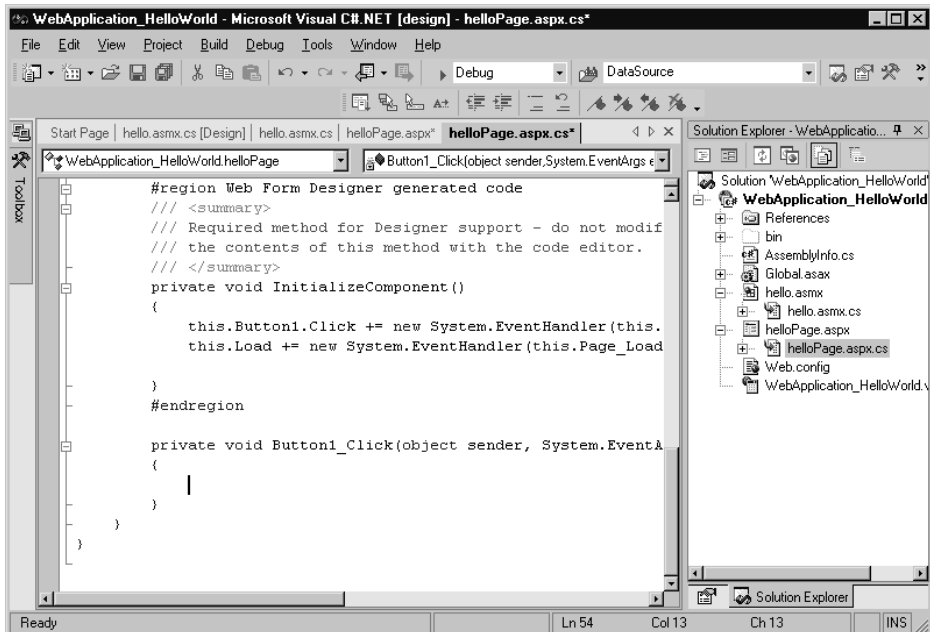


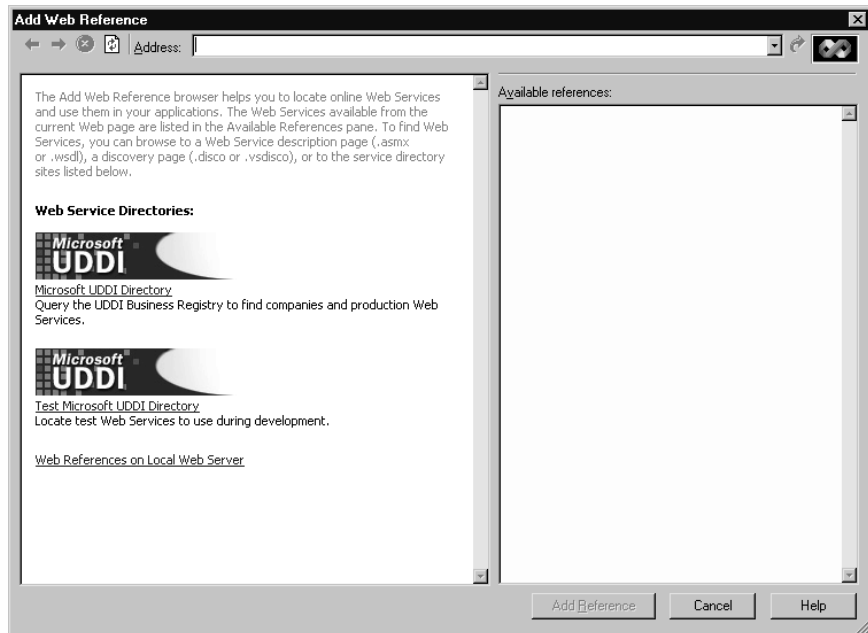
Figure 1.9 Auto-generated Button Event Code



- When the **Add Web Reference** dialog opens (see Figure 1.10) click the link **Web References on local server**.

The dialog will pause while it searches your local machine for a list of services available.

**Figure 1.10** Add Web Reference Dialog Box



- When the list appears, click the name of the service that matches the name of your project, **WebApplication\_HelloWorld**.
- When the service loads, click the **Add Reference** button. This will create several new entries in your Solutions Explorer.
- Now take a look at helloPage.aspx in HTML view. You should see code similar to the following:

```
<body MS_POSITIONING="GridLayout" >
  <form id="helloPage" method="post" runat="server" >
    <asp:Button id=Button1 Text="Button" runat="server" >
      </asp:Button>
    <asp:Label id=Label1 runat="server">Label</asp:Label>
  </form>
</body>
```

- Note the name of the label control is Label1. Now open **helloPage.aspx.cs** and add the following code below the label and button code.

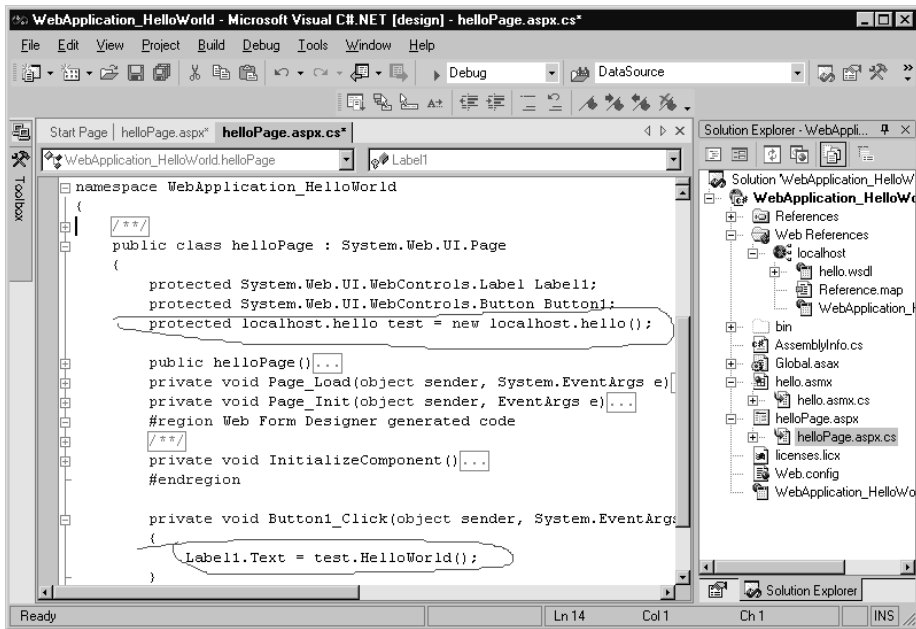
```
localhost.hello test = new localhost.hello();
```

- In the **Button Click handler**, add the following:

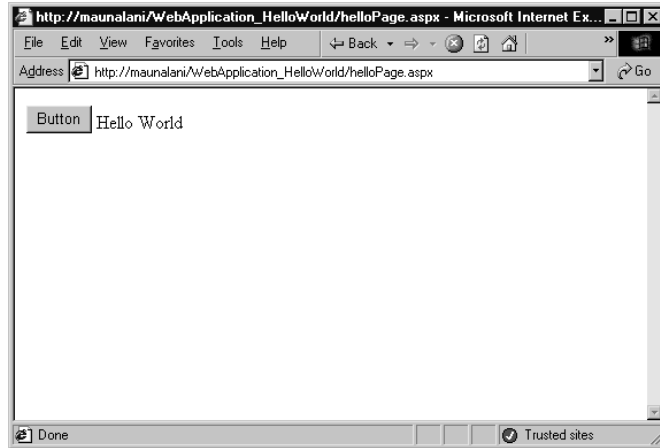
```
Label1.Text = test.HelloWorld();
```

- Your code should now look like Figure 1.11.

**Figure 1.11** helloPage.aspx.cs



- Right-click **helloPage.aspx** and click **Set as start page**.
- Press **F5** to run the application.
- When the browser loads, click the button, this will invoke our *helloWorld* method and assign its value to the label text. After clicking the button, your page should look like Figure 1.12.

**Figure 1.12** HelloPage.aspx in the Browser after Clicking the Button

## WSDL

WSDL is an XML-based language that describes Web Services. It is the composite of work done by Ariba, IBM, and Microsoft. Currently, it only supports SOAP as a messaging protocol.

The thought behind WSDL is that in future applications it will be a collection of networked-Web Services. WSDL describes what a service can do, where it lives, and how to invoke it. WSDL describes the Web Service method interfaces thoroughly enough for it to be used to create proxy methods that enable other classes to invoke its members as if they were local methods. IBM and Microsoft both have WSDL command line utilities available that do just that. IBM does it for Java, and Microsoft does it for Visual Studio. VS.NET has this ability built into the GUI. In VS.NET, we simply right-click **add Web Reference** and select the service we want to generate a proxy class for. Here is an example of a WSDL file for a Web Service: `getCategories.wsdl`. This file is auto-generated by the .NET Framework.

While the auto-generated file will cover the basic functionality, it may do more or less than you intended. The auto-generated code can be simplified by removing support for asynchronous operations if you do not need to support this type of operation. Also, you could add custom SOAP headers and customize other parts of the SOAP envelope by creating your own class. WSDL and UDDI are covered in later chapters.

## Developing & Deploying...

### **When Moving a VS.NET Web Service to Another Server**

When transferring a project to another server, make sure the page namespaces match the project name and be sure to update Web references.

## Using XML in Web Services

Web Services use SOAP as a messaging protocol. SOAP is a relatively simple XML language that describes the data to be transmitted. Why use XML? XML is a standard language designed to be understandable by humans, and structured so it can be interpreted programmatically. XML does not only describe data, it can also describe structure, as we will see when we take a closer look at the ADO.NET *DataSet*.

Consider the case of replicating a database into cache. We might want to do this to reduce the load on the database server, to speed client processing, or to provide an offline data handling scenario. We could transport an XML document that contains the new W3C XML Schema Definition Standard (XSD) schema describing the database tables, relations, and constraints, along with the actual data (see the section “Using DataSets” later in this chapter). Because XSD can describe relational data and can be embedded within an XML document, any database can be converted to a ubiquitous data source. That is, a data source that can be accessed on any platform by any application. This is possible because the transfer protocol, SOAP, uses XML over HTTP and because XML, XSD, SOAP, and HTTP are all nonproprietary industry standards.

It is the use of non proprietary industry standards that makes Web Services so powerful. By using XML to describe structure and content, Web Services can provide an interface to data on legacy systems, or between incompatible platforms from acquisitions or between vendors over intranets, extranets, or the Internet.

# An Overview of the `System.Web.Services` Namespace

`System.Web.Services` is the namespace from which all Web service classes are derived. It consists of all the classes needed to create Web Services in the .NET Framework. When using VS.NET most of the `System.Web.Services` classes and subclasses are transparent to the developer, so we won't go into much depth here. The three primary child classes of `System.Web.Services` are: `Description`, `Discovery`, and `Protocols`.

## The `System.Web.Services.Description` Namespace

The `System.Web.Services.Description` namespace contains the classes needed to describe a Web Service using the Microsoft SDL (Service Definition Language), a Microsoft implementation of the WSDL standard. VS.NET uses these classes to create the `.disco` or `.vsdisco` file. Many of the subclasses of this class are related to binding: `MessageBinding`, `OperationBinding`, `OutputBinding`, and so on. One of the more interesting subclasses is the `ServiceDescription` class. It takes as a parameter an XML file and enables the creation of a valid WSDL file.

```
ServiceDescription MyDescription = new ServiceDescription();
ServiceDescription MyDescription =
    ServiceDescription.Read("MyTestFile.xml");
```

## The `System.Web.Services.Discovery` Namespace

The `System.Web.Services.Discovery` namespace consists of the classes that enable Web Service consumers to locate available Web Services. In VS.NET when we create a Web Reference, these classes find the `.vsdisco` files that describe Web Services.

Disco file from our Hello World example:

```
<?xml version="1.0" encoding="utf-8"?>
<discovery xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           xmlns="http://schemas.xmlsoap.org/disco/">
  <contractRef
    ref="http://localhost/WebApplication_HelloWorld/hello.asmx?wsdl"
    docRef="http://localhost/WebApplication_HelloWorld/hello.asmx"
```

```
xmlns="http://schemas.xmlsoap.org/disco/scl/" />
</discovery>
```

## The *System.Web.Services.Protocols* Namespace

The *System.Web.Services.Protocols* namespace consists of the classes used to define the protocols that enable message transmission over HTTP between ASP.NET Web Services and ASP.NET Web Service clients. These classes are used in our WSDL proxy classes. They are mostly involved with the formatting, bindings, and settings of the SOAP message.

WSDL proxy from our Hello World example:

```
namespace WebApplication_HelloWorld.localhost {
    using System.Diagnostics;
    using System.Xml.Serialization;
    using System;
    using System.Web.Services.Protocols;
    using System.Web.Services;

    [System.Web.Services.WebServiceBindingAttribute(Name="helloSoap",
        Namespace="http://tempuri.org/")]
    public class hello :
        System.Web.Services.Protocols.SoapHttpClientProtocol {

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        public hello() {
            this.Url =
                "http://localhost/WebApplication_HelloWorld/hello.asmx";
        }

        [System.Diagnostics.DebuggerStepThroughAttribute()]
        [System.Web.Services.Protocols.SoapDocumentMethodAttribute(
            "http://tempuri.org/HelloWorld",
            Use=System.Web.Services.Description.SoapBindingUse.Literal,
            ParameterStyle=
                System.Web.Services.Protocols.SoapParameterStyle.Wrapped)]
        public string HelloWorld() {
            object[] results = this.Invoke("HelloWorld", new object[0]);
```

```

        return ((string)(results[0]));
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    public System.IAsyncResult BeginHelloWorld(
        System.AsyncCallback callback, object asyncState)
    {
        return this.BeginInvoke(
            "HelloWorld", new object[0], callback, asyncState);
    }

    [System.Diagnostics.DebuggerStepThroughAttribute()]
    public string EndHelloWorld(System.IAsyncResult asyncResult) {
        object[] results = this.EndInvoke(asyncResult);
        return ((string)(results[0]));
    }
}
}
}

```

## Type Marshalling

*Type marshalling* refers to the translation of datatypes from an application or database as it is mapped to a SOAP datatype. When any datatype, object, method, or string (xml, or a simple string) is passed as a SOAP request or response, it is automatically converted into an XML representation of itself. Since any programming language can use SOAP, SOAP has defined its own set of datatypes. When data is passed in a SOAP envelope its datatypes are translated or converted to a SOAP equivalent. This enables different languages with different names for similar datatypes to communicate effectively. The datatypes supported when using Web Services include:

- **Standard primitive types** *String, char, Boolean, byte, single, double, DateTime, int16, int32, int 64, UInt16*, and so on.

string "hello World" is represented as:  
 <string>hello World</string>



- **Enum Types** Enumerations like `enum weekday {sun=0, mon=1, tue=2, wed=3, thu=4, fri=5, sat =6}`

- **Arrays of Primitives or Enums**

`MyArray[ 5,7 ]` is represented as:

```
<ArrayOfInt>
  <int>5</int>
  <int>7</int>
</ArrayOfInt>
```

- **Classes and Structs**

`struct Order( OrderID, Price )` is represented as:

```
<Order>
  <OrderID>12345</OrderID>
  <Price>49.99</Price>
</Order>
```

- **Arrays of Classes (Structs)**

`MyArray Orders( order1, order2 )` may be represented as:

```
<ArrayOfOrder>
  <Order>
    <OrderID>int</OrderID>
    <Price>double</Price>
  </Order>
  <Order>
    <OrderID>int</OrderID>
    <Price>double</Price>
  </Order>
</ArrayOfOrder>
```

- **DataSets** The representation of a DataSet is rather lengthy; it includes an inline XSD schema defining the structure followed by the XML data. For an example of a DataSet, see the next section, “Using *DataSets*.”
- **Arrays of DataSets**
- **XmlNodeNodes**

```
<book id=1><title>book1</title><price>25.00</price></book>
```

## ■ Arrays of XmlNode

```
<ArrayOfBook>
  <book id="1">
    <title>book1</title>
    <price>25.00</price>
  </book>
  <book id="2">
    <title>book2</title>
    <price>49.99</price>
  </book>
</ArrayOfBook>
```

It is important to note that when we create and use Web Services in VS.NET, the marshalling of data is transparent to the developer. This is also true when using the WSDL.exe command line utility. While it is important to have some understanding of how data is transported between the Web Service and the Service proxy or client, this layer is and should be transparent to the developer, just as packet structures for transmitting data over HTTP is transparent to the Web developer.

## Using DataSets

A *DataSet* can be used to cache an entire database within an ASP application variable. This would reduce the Database Server Load and speed data access over the life of the application object. The following is a code snippet that calls a Web Service that returns a *DataSet*. The *DataSet* in turn stores the data in an application object.

```
myServer.getBooks DataSource = new myServer.getBooks();
Application["AllBooks"] = DataSource.AllBooks();
```

This makes the *DataSet* available to all instances of the Web application, which is very efficient. Operations can be performed on the *DataSet* and, on *Application\_End* the Database can be updated.

*DataSets* store database structure information and contain *DataTable*, *DataColumn*, *DataRow*, and *DataView* children. *DataSet RowFilter* operations are very much like SQL Queries. The *DataSet* can easily be databinded to ASP.NET UI controls. It also has an XML output format that makes it easily translated to XML for XML processing.

Here is an example of the Books *DataSet* returned by the *getBooks.allBook* service:

```
<?xml version="1.0" encoding="utf-8"?>
<DataSet xmlns="http://tempuri.org/">
  <xsd:schema id="NewDataSet" targetNamespace="" xmlns=""
    xmlns:xsd=http://www.w3.org/2001/XMLSchema
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xsd:element name="NewDataSet" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="Books">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="isbn" type="xsd:string"
                  minOccurs="0" />
                <xsd:element name="name" type="xsd:string"
                  minOccurs="0" />
                <xsd:element name="id" type="xsd:int" minOccurs="0" />
                <xsd:element name="imgSrc" type="xsd:string"
                  minOccurs="0" />
                <xsd:element name="author" type="xsd:string"
                  minOccurs="0" />
                <xsd:element name="price" type="xsd:decimal"
                  minOccurs="0" />
                <xsd:element name="title" type="xsd:string"
                  minOccurs="0" />
                <xsd:element name="description" type="xsd:string"
                  minOccurs="0" />
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
```

```
xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
<NewDataSet xmlns="">
  <Books diffgr:id="Books1" msdata:rowOrder="0">
    <isbn>0072121599    </isbn>
    <name>cisco</name>
    <id>2</id>
    <imgSrc>ccda.gif</imgSrc>
    <author>Syngress Media Inc</author>
    <price>49.99</price>
    <title>Ccda Cisco Certified Design Associate Study Guide</title>
    <description>
      Written for professionals intending on taking the CCDA test,
      this special guide covers all the basics of the test and
      includes hundreds of test questions on the enclosed CD.
    </description>
  </Books>
  <Books diffgr:id="Books2" msdata:rowOrder="1">
    <isbn>0072126671    </isbn>
    <name>cisco</name>
    <id>2</id>
    <imgSrc>ccna.gif</imgSrc>
    <author>Cisco Certified Internetwork Expert Prog</author>
    <price>49.99</price>
    <title>CCNA Cisco Certified Network Associate Study Guide</title>
    <description>
      Cisco certification courses are among the fastest-growing
      courses in the training industry, and our guides are designed
      to help readers thoroughly prepare for the exams.
    </description>
  </Books>
</NewDataSet>
</diffgr:diffgram>
</DataSet>
```

## Summary

In this chapter, we discussed Web Services, along with their related technologies, protocols, and standards, such as Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL), Extensible Markup Language (XML), and the XML Schema Definition (XSD) standard. We examined the role of Web Services and how messages are passed between servers and data sources.

We created simple Web Services (producers) as well as Web Services (consumers) using the .NET Framework and VS.NET to show how the Web Service messaging infrastructure works and how it can be used transparently to the developer.

The power of Web Services is due to its foundation in nonproprietary protocols and standards. Web Services would not be as useful if it were not built on XML for defining data and structure, XSD for defining structure, SOAP for defining a messaging transport mechanism over the well-established HTTP, WSDL for defining method interfaces in XML, Universal Description, Discovery, and Integration (UDDI, a Web Service discovery mechanism), and DISCO, the Web Service discovery description document.

## Solutions Fast Track

### Web Services

- ☑ Web Services provide an XML interface that can be accessed by any SOAP-enabled client, which means a Web Service developed with .NET can be accessed by a Java application, a Web page, or any SOAP-enabled desktop application.
- ☑ Web Services can be accessed over HTTP through port 80, which means remote procedure calls can be made to objects behind firewalls.

### Using XML in Web Services

- ☑ XML is the enabling standard upon which SOAP and Web Services are built.

- ☑ The SOAP envelope is an XML document. The SOAP message, meanwhile, describes the data being passed as an XML representation of the original datatype or object.

## An Overview of the *System.Web.Services* Namespace

- ☑ *System.Web.Services* is .NET Framework's namespace of classes that enable .NET Web Services. The three primary subclasses or subnamespaces are:
  1. ***System.Web.Services.Description*** Classes that support WSDL, used to define the methods, parameters, and datatypes of Web Services.
  2. ***System.Web.Services.Discovery*** Classes that support UDDI and the generation of WSDL proxies for Web Service clients.
  3. ***System.Web.Services.Protocols*** Classes that support the generation and customization of Web service protocols, and can be used for things such as creating custom SOAP headers.

## Type Marshalling

- ☑ Type marshalling is the mapping of types from Web Service method calls to SOAP datatypes.
- ☑ When remote calls are made using Web Services and the SOAP protocol; datatypes and objects that are passed are represented as XML descriptions of themselves. (Datatypes are marshalled as one of many SOAP standard datatypes.)

## Using *DataSets*

- ☑ *DataSets* are ADO.NET objects that provide database type operations.
- ☑ *DataSets* enable the transfer of database structure and content to and from WebServices.

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** Why replace COM objects with Web Services?

**A:** Web Services have a platform neutral interface. This enables Web Services to be easily utilized by multiple clients on different platforms developed with different programming languages. Note that existing COM components can be wrapped by Web Services.

**Q:** How do I know I need Web Services?

**A:** If you have data that is needed by various customers (different departments, different levels of management, vendors, industry partners, consumers and so on) and getting to that data is hindered or prevented by issues involving platform, programming language, legacy hardware or other types of incompatibility, developing Web Services can help.

**Q:** What area of development are Web Services best for?

**A:** I believe that Web Services development like COM development will remain in the hands of the middle tier programmer.

**Q:** Is it possible to try out Web Services using only my local computer?

**A:** Yes, it is. Using the WSDL.exe command line tool, you can point to any Web server. This is even easier with the VS.NET UI. Simply right-click **Web references**, then select any Web service from the UDDI directory or your local machine, or simply type the URL of a disco file on any server. You can easily generate a WSDL proxy and use it as long as you are connected to the Internet.

**Q:** I’m currently in the process of migrating. What considerations should I take with my existing COM components?

**A:** Here are a few things to consider:

- *Who is the customer?* If the customer is only within the intranet and there are no external customers in the foreseeable future, an existing DCOM infrastructure needn't be changed.
- *What type of clients do I have?* If the client is a desktop application, switching to Web Services would require updating the client, which may include updating the OS, and possibly the hardware so that the client has enough memory to host the .NET Framework.
- *Will I need to support Mobile devices in the near future?* Using the .NET Mobile Framework to access Web Services is as simple as it is with .NET. Updating the existing clients to .NET will make adding future clients simple and cost-effective.





## Introduction to the Microsoft .NET Framework

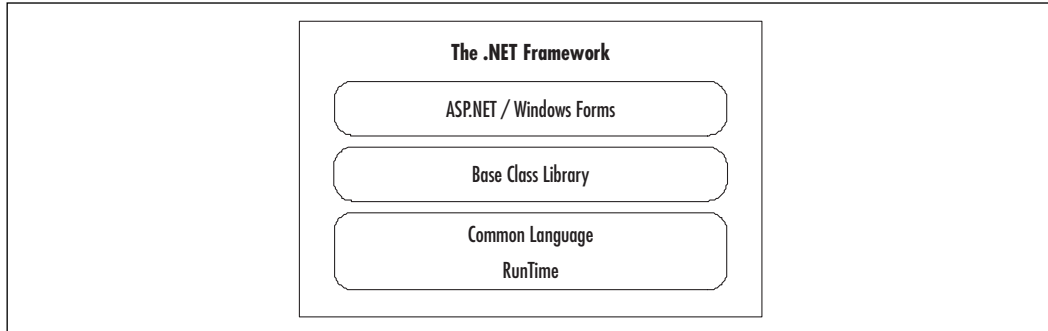
### Solutions in this chapter:

- Obtaining the .NET Framework
- Installing the .NET Framework
- Common Language Runtime
- Developing Applications with the .NET Framework
- Components in the .NET Platform
  
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

# Introduction

The .NET Framework is just part of Microsoft's overall .NET platform strategy. The framework is made up of the Common Language Runtime environment, Base Class Library, and higher-level frameworks such as ASP.NET and Windows Forms as shown in Figure 2.1.

**Figure 2.1** The .NET Framework Architecture



The complete documentation on the .NET Framework fills entire books. What we do here is cover the basics so that you have a firm enough understanding of the .NET Framework to enable you to get started developing XML Web Services.

We'll start with how to obtain the .NET Framework SDK. The minimum system requirements can be confusing so we'll cover those in some detail. The common language runtime (CLR) is the foundation that sits on top of the Windows operating system. Since this is the most important part of the .NET Framework we'll spend most of our time going through the CLR. The base class library is a set of hundreds of classes that are provided as part of the framework to help us build applications that will execute in the CLR. We'll get an overview of these so that we can get an idea of the breadth of support provided by the SDK "out of the box". To make building applications even easier Microsoft provides some higher level frameworks like ASP.NET and Windows Forms that utilize and extend the functionality provided by the Base Class Library. ASP.NET greatly simplifies the building of Internet applications by using Web Forms and Web Services. Windows Forms provides the ability to develop for the rich environment that the Windows platform provides. We'll take a look at these frameworks to give you an overview of the purpose of each.

# Obtaining the .NET Framework SDK

The .NET Framework SDK contains a Framework Runtime for redistribution, compilers, tools, documentation and samples. This is everything you need to develop, test, and deploy applications that target the .NET Framework. You can obtain the .NET Framework from multiple places. Before detailing those locations let's take a look at the system requirements for installing the .NET Framework. Please review them carefully and you will save yourself some major headaches.

If you can, I would recommend that you install the .NET Framework on a machine or partition that has a new operating system installation and is totally separate from your important applications and data. It's much easier if you encounter any unforeseen problems to just reformat and start over. At a minimum, you should uninstall any previous versions of the .NET Framework SDK prior to installing a newer version, and don't install the SDK on a machine that has production data on it.

## NOTE

The .NET Framework SDK and Visual Studio .Net are not the same thing. Visual Studio .NET is an application development tool that enables you to create applications targeted for the .Net Framework CLR. You can create .NET applications with any text editor and compile with the SDK provided compilers. Visual Studio .NET just makes it easier.

## System Requirements

Prior to installing the .NET Framework SDK you must make sure that your system meets or exceeds the minimum system requirements. Remember that they are the *minimum* requirements. As always, before you install any subsequent versions be sure to note the requirements in the documentation provided with the install. Requirements can change from version to version as enhancements are made to the SDK. The following sections outline the .NET Framework SDK minimum system requirements:

## Hardware

The hardware requirements are quite reasonable by today's standards for desktops and servers. You can see by the minimum and recommended notes that the more RAM you have, the better.

### *Desktop .NET Framework SDK Install*

The requirements for developing desktop application using the .NET Framework SDK are:

- **Processor** Intel Pentium class 90 MHz or higher
- **RAM** 32 MB (96 MB or higher recommended)
- **Video** 800x600, 256 colors

### *Server .NET Framework SDK Install*

The following requirements below are to install and develop with the .NET Framework SDK, not just the runtime. We'll cover the requirements for the runtime in the next section.

- **Processor:** Intel Pentium class 133 MHz or higher
- **RAM:** 128 MB (256 MB or higher recommended)
- **Hard disk space required to install:** 360 MB
- **Hard disk space required:** 210 MB
- **Additional space required to install and compile all samples:** 300 MB
- **Video:** 800x600, 256 colors

#### NOTE

---

ASP.NET is not supported on Windows 95/98 operating systems such as Windows Me, Windows 98 Second Edition, Windows 98, and Windows 95.

---

## Operating System

Currently the only operating systems that can be used with the .NET Framework SDK are Windows operating systems. Since the CLR is so much like the Java virtual machine, don't be surprised if some enterprising third-party creates a CLR for another operating system. For now here are the operating system requirements for the .NET Framework development and runtime.

### *.NET Framework SDK*

The following are the operating system requirements for developing with the .NET Framework SDK:

- Windows XP, Windows 2000, Windows NT 4.0
- Internet Explorer 5.01 or later

### *.NET Framework Redistributable Package (Runtime)*

Please note that the requirements below are for the runtime only. They do not include development. You would install the runtime on a machine that contains and runs deployed components. Just as you would expect, the requirements for only the runtime are less restrictive than those for development with the SDK. The operating system requirements for the runtime are as follows:

- Windows XP, Windows 2000, Windows NT 4.0
- Windows 98, Windows Me
- Internet Explorer 5.01 or later

## Additional Installation Information

Frequently included with the install is documentation listing any late-breaking issues that have come up since the version was first issued. This is information in addition to the normal readme.htm file. The following list includes a few items that are in the .NET Framework SDK install documentation and late-breaking issues that you may overlook if you're not careful:

- Internet Explorer 5.5 is distributed along with the .NET Framework installation.
- For Server setups Microsoft Data Access Components (MDAC) 2.6 is required, MDAC 2.7 is recommended. You can download this from [www.microsoft.com/downloads](http://www.microsoft.com/downloads).

- For Windows NT 4.0 setups Service Pack 6a must be installed.

## NOTE

Visual Studio .NET includes the .NET Framework SDK

## Locations for Downloading

There are multiple locations available for downloading the .NET Framework SDK. Some of these include:

- [www.microsoft.com/net](http://www.microsoft.com/net)
- [www.gotdotnet.com](http://www.gotdotnet.com)
- [msdn.microsoft.com/net](http://msdn.microsoft.com/net)

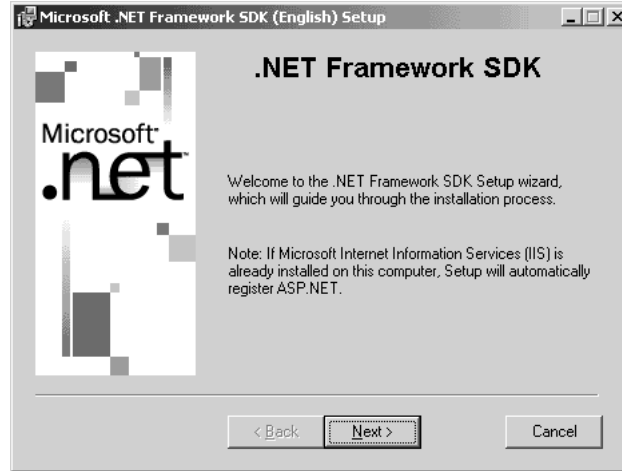
You can also obtain the SDK on a CD-ROM by going to the appropriate Microsoft Developer Store, the locations of which are listed on the Web sites noted in the preceding bullets. If you are an Microsoft Developer Network (MSDN) subscriber you can download the SDK from the MSDN subscriber download site ([msdn.microsoft.com](http://msdn.microsoft.com)).

## Installing the .NET Framework

Because the download file is over 120 MB, you have a choice of downloading one large file or many smaller ones. If you decide to go the multifile download route you need to take the following steps to prepare the install:

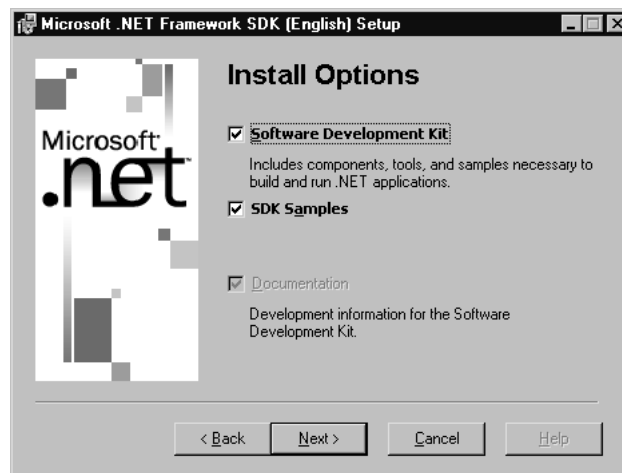
1. Place each downloaded file into the same directory.
2. Run the **setup.bat** file. This will create a master setup.exe file. To find out if your COPY command supports the /Y switch go to a DOS prompt and type in **COPY /?**. If the /Y switch is listed then your system supports it. If it isn't then you need to edit the setup.bat file to remove the /Y switch from the COPY command.
3. Run the **setup.exe** to install the .NET Framework SDK.

The installation program provides a wizard that takes you through the process as noted in Figure 2.2.

**Figure 2.2** Initial Setup Screen

The installation program searches your machine to see if you have any components already installed, and will direct your options accordingly. If you don't have the updated Windows Installer components you will be asked whether or not you want to update them. Because these components are required for the install you must answer **Yes**.

Figure 2.3 shows an example of an installation that already has the documentation installed. So if this is the first .NET Framework SDK installation on your target machine, you have the option of installing the SDK, the SDK samples, and/or the documentation.

**Figure 2.3** .NET Framework SDK Installation Options

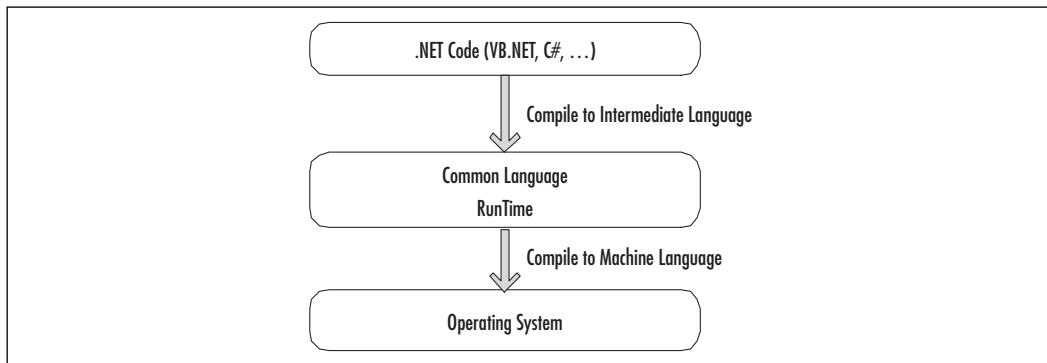


The SDK install includes the Framework as well as ASP.NET. Once you have the install completed you can verify proper operation by going to the ... \Microsoft.NET\FrameworkSDK\Samples\StartSamples.htm page and running some samples. This page has tutorials, technology samples, and a few full applications that will allow you to make sure that you are set up and ready to code. You can get a jump start on your learning as well.

## Common Language Runtime

The Common Language Runtime (CLR) is arguably the most important part of the .NET Framework. It is responsible for executing and managing the compiled output of code written in the .NET-compatible languages. The CLR is the engine at the core of managed code execution. Architecturally it sits above the operating system and below the .NET language code as shown in Figure 2.4.

**Figure 2.4** CLR Placement in .NET Architecture



## Major Responsibilities of the CLR

The runtime supplies the managed code with many services, such as cross-language integration, code access security, object lifetime management, and debugging support. Let's take a look at the more important responsibilities in more detail.

### NOTE

The phrase "managed code" keeps coming up when discussing the CLR. Basically code compiled with a language compiler that targets the CLR is called managed code.

## Safety and Security Checks

Before the CLR executes any code, that code is checked to make sure that it is type-safe in regards to memory locations. Each type has its own area of memory, and can't access the private data from other types. This means that code allowed to run in the CLR can only access memory locations that it is authorized to access. This prevents some of the crashes and tampering that can occur in other languages.

Code can also have specific security requirements that tell the CLR to verify that any client requesting the code in question meets those requirements. The security information resides inside the code executable files (DLLs or EXEs). Depending on the current security policy, certain operations may not be allowed as well.

## Class Loading

When you start a .NET application, the Windows operating system recognizes that the executable was compiled for the CLR and passes control over to the CLR. The .NET Framework SDK install takes care of updating the operating system to be able to accomplish this. The CLR must find the entry point (*Main()* *method*) of the application and locate the class that contains that entry point.

The CLR is responsible for finding the proper version of the executable file and if it passes the security checks, loads the class in question, and activates the desired object when required. To find the proper version, the CLR reads the information saved with the executable and searches a defined path to locate the desired file. Once found the class loader can load the class in memory and cache the classes' type information so that it can just pull the information from the cache the next time the class is called. This is why the first time you run an application might seem a little bit slower than subsequent runs.

## Object Lifetime Management

One of the historic hassles with development in the lower-level object-oriented languages is making sure that your objects are cleaned up when you no longer use them. Because the CLR handles that for you now, it's no longer a concern. The chance of memory leaks occurring and taking down a server is greatly reduced, and a big troubleshooting headache has been removed.

We discussed above how the CLR takes care of loading and activating objects. The CLR is also responsible for monitoring or tracing the operation of the object and removing the object when it is no longer referenced. Once an

object is no longer referenced in any way from any other objects or variables, garbage collection (GC) will occur and return that memory for use.

## NOTE

---

Garbage collection occurs based on many different calculations and can't be forced to occur. Therefore use of code intended to run when an object is destroyed (destructors) is not recommended. There is a *Dispose()* method that disables the object but leaves it in memory. This is where you would put in your cleanup code if it is still necessary.

---

## Just In Time (JIT) Compilation

When you compile your code using Visual Studio .NET or one of the command line compilers, that code isn't converted directly into machine language. It's converted to code called the Microsoft Intermediate Language (MSIL) or Common Intermediate Language (CIL). This is how one .NET language can utilize objects created with other .NET languages. Both are compiled to the MSIL. To speed up the process of compiling the MSIL to native code for the associated CPU, the .NET Framework includes JIT compilers—one for each CPU architecture that .NET supports.

Each method for which MSIL has been generated is JIT-compiled when it is called for the first time, cached, then executed. The next time the method is executed, the existing JIT-compiled native code from the cache is executed. This process of JIT compiling and then executing the code is repeated until the application execution is finished. As mentioned above the type safety is also checked at this time. As part of the compiling process the code goes through the security checks noted earlier in the “Safety and Security Checks” section.

## Cross-Language Interoperability

All languages targeted for the CLR must follow the Common Type System (CTS) rules for how their types are created and used. In addition, type information storage is the same across languages. So when the CLR looks up the type information to provide the proper services and execute the code it doesn't matter what language it was written in, it's MSIL at this time. The MSIL is JIT-compiled and executed in the same way regardless of language. The result of this interoperability is that we get the following by using managed code:

- Inheriting from types created in other languages.
- Passing objects created in one language into a method of a class created by a different language
- Common debugger across languages. You can step through code from one language to another as you debug.
- Error (exception) handling is the same across languages. An exception “thrown” in one language can be “caught” in another, and handled.

## NOTE

One caveat with cross-language interoperability: In certain cases one language supports functionality that another may not support. The example usually presented to describe this is if one language has support for unsigned integers (*type UInt32*) and you try to add a parameter of *UInt32* to your code in a language that doesn't support *UInt32*, your code wouldn't run. Keep that in mind when you have different languages involved.

## Structured Exception Handling

Structured exception handling is a very significant part of the .NET Framework. The “structure” part is a control structure identical to that found in Java. It is composed of the *Try-Catch-Finally* block. This is similar to how C++ handles exceptions. We take another look at exception handling in the section “Structured Exception Handling Revisited” a little later on in the chapter.

## Assemblies

Assemblies are the executables of the .NET Framework (DLL or EXE). Your source code project compiles into an assembly. You can choose to have more than one compiled file in your assembly. Assemblies are known as the smallest deployable unit. All managed types and resources are marked either as accessible only within their implementation unit or as exported for use by code outside that unit. In the runtime, the assembly establishes the name scope for resolving requests and the visibility boundaries are enforced. The runtime can find the proper assembly and the proper type when it needs to load that type to prepare for execution. All types are contained in assemblies.

## Metadata

When your code is compiled for the CLR, more is going on than just compiling code. During compilation, the .NET targeted compilers also produce the metadata that describes all of the types contained in the executables (EXEs or DLLs). This metadata includes type information that you would typically find in a COM-type library, and version dependency information. This version information is what the CLR uses to allow multiple executables with the same name residing on the same machine (in different directories). In addition, information describing all of the resources that your components use is also contained there. So the components are seen as “self-describing”. When you start an application the CLR looks to the assembly before loading to make sure that everything you need to execute that code is available and proper. The tracking down dependency/versioning problems for days like we had to do with COM is now a thing of the past.

The information found in the metadata describes every element managed by the runtime, and any other information that is required by the runtime. Some examples are: type, debugging, garbage collection, and versioning information. If you want to add more information than is provided by the compilers, you can create custom metadata that allows you to add attribute information to your assemblies that can be read and acted upon at runtime. The information found in the metadata includes the following:

- Assembly description:
  - Identity (name, version, culture, public key)
  - Other assemblies that this assembly depends on
  - Any security permissions needed to run your code
- Description of types (classes):
  - Name, visibility, base class, and interfaces implemented
  - Members (methods, fields, properties, delegates, events, inner types)
- Attributes:
  - Additional descriptive elements that modify types and members
  - Any custom attributes that you create yourself

## Enhanced Deployment and Versioning Support

Deployment is one big area where the .NET Framework makes your life easier. You have the ability now to create *deployment projects*. These projects created within Visual Studio .NET use the Windows Installer technology. Deployment is totally different from previous versions of Visual Studio, so here we cover some of the highlights. You have various options to bundle your files for deployment:

- Use your assemblies as-is.
- Place files in a compressed CAB file. A good choice for large projects that would take a long time to download if installed over the Web.
- Create a Windows Installer package to take advantage of that technology. This gives you the most options as you are creating a Visual Studio .NET deployment project.
- Using a third party installer.

You also have various options to distribute your bundled files.

- XCOPY or FTP. If you package your assemblies as-is you can just copy the files over to an appropriate directory (using the proper namespace scope) and the application will run right from there.
- Have users download the code from a Web site and run your code from there.
- If you use Windows Installer to create your package, the installer can move the assemblies to wherever you need them if the security policy is met. You should understand these security rules prior to creating your install package.

## Managed versus Unmanaged Code

Managed code runs in collaboration with the CLR during runtime. Managed code carries with it the metadata necessary for the runtime to provide services such as memory management, cross-language interoperability, code access security, and control of the object from instantiation through disposal. All code based on MSIL executes as managed code.

Basically unmanaged code is code created outside of the .NET Framework. This code also runs outside of the .NET Framework in the Microsoft Windows environment. The CLR has the ability to run un-managed and managed code

together, but there is a performance hit. Some common examples of unmanaged code are C++ classes compiled with a non-CLR compiler, COM components, and the classes that make up the Win32 API.

## Interoperability with Unmanaged Code

The CLR takes care of interoperability between managed and unmanaged code, so it is quite seamless. This makes the decision of how and when to migrate your legacy code to .NET an easier one. You don't have to rewrite everything right away. The .NET Framework enables interoperability with COM+ services, the Windows operating systems, and COM components.

## Namespaces

Namespaces provide a logical naming scheme for grouping related types, similar to Java packages. For example, the .NET Framework uses a hierarchical naming scheme for grouping types into logical categories of related functionality, such as Data Access, GUI, Reporting, and so on. Design tools can use namespaces to make it easier for developers to browse and reference types in their code. In the .NET Framework, a namespace is a logical design-time naming convenience, whereas an assembly establishes the name scope for types at run time. Namespaces also aid the CLR in selecting the correct assembly to load when multiple assemblies have the same name.

For example when you want to access a class found in the .NET Framework we would use the fully qualified name, which includes the namespace. In the reference *System.Data.Dataset*, *System.Data* is the namespace and *Dataset* is the type name. Microsoft states that all namespaces shipped by Microsoft will start with either *System* or *Microsoft*.

Since namespaces provide the scope for your types the standard is to group types with related functionality in separate namespaces. *System.Data*, *System.Xml*, *System.Collections*, *System.Web*, *System.Net*, *System.Threading*, and *System.Security* are good examples of creating namespaces with logical groupings. These are found in the Base Class Library that Microsoft provides as part of the .NET Framework. We look into the Base Class Library and its namespaces in the next section.

# Developing Applications with the .NET Framework

To bridge the gap between the .NET Framework and developing XML Web Services, we will cover some programming related information. In this section we will look at the following:

- Development platforms
- .NET language choices
- Compilers
- Tools to make your .NET programming life easier
- Base Class Libraries

## Development Platforms for .NET

You can create all of your code using a text editor such as Notepad if you like. The SDK provides all of the tools necessary to accomplish development at that level. You would need to learn all of the specific compiler options and debugger tools and probably take quite a while before you could be productive.

Third-party and open source tools are cropping up that can color-code keywords and provide some level of syntax checking. Some tools you can also link to a compiler so that you can code and compile in the same integrated development environment.

By far the best tool for .NET development we've used is Visual Studio .NET. With VS.NET you can code, compile, and build deployment projects all within the same development environment. Figure 2.5 shows a typical project open in Visual Studio .NET Enterprise Architect Edition.

Figure 2.6 depicts another C# editor available on the Web. More and more editors of various levels of cost and functionality are showing up on the Web every day.

Once you have your development platform selected there are other issues to consider. We'll look at a few of those next.



Figure 2.5 Visual Studio .Net Integrated Development Environment

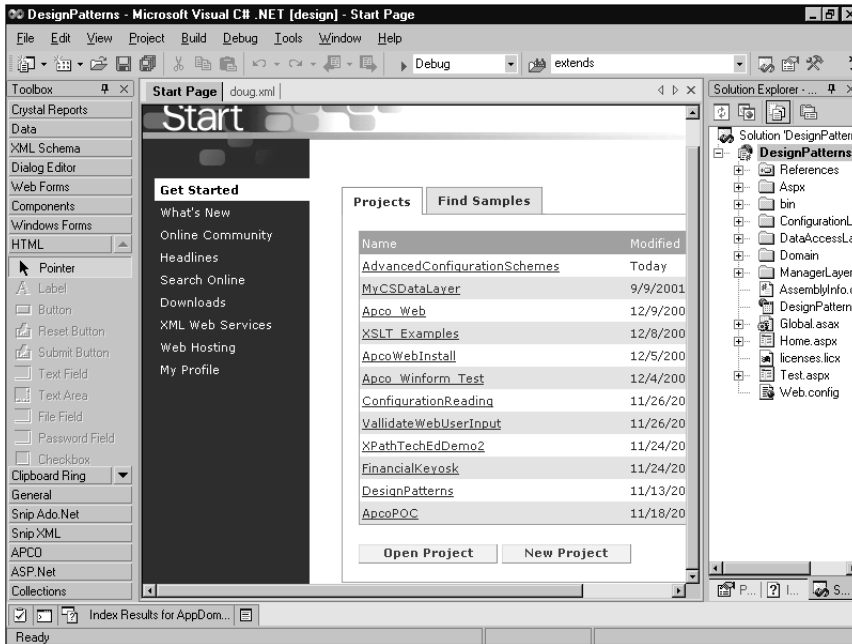
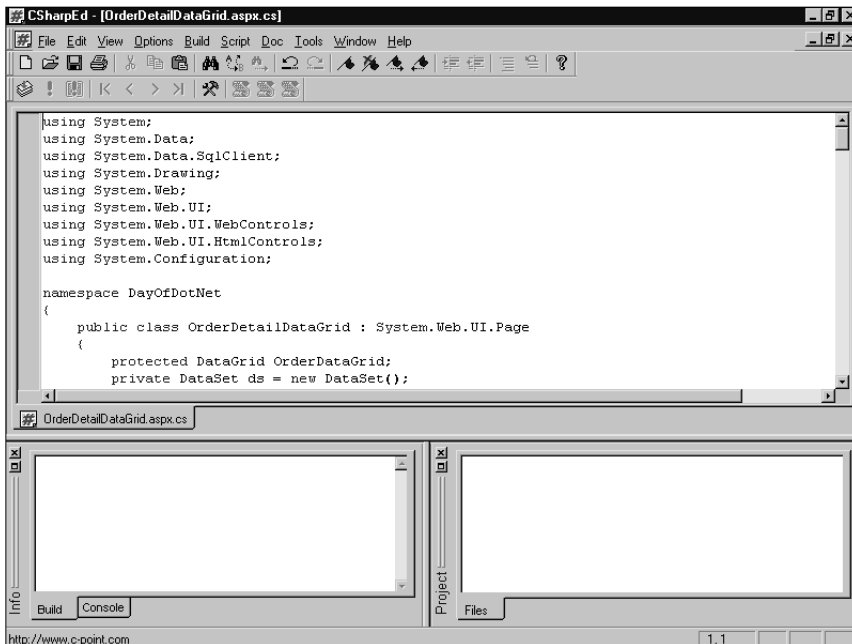


Figure 2.6 Antechinus C# Editor



## Language Choice

Currently there are many languages being targeted for the .NET Framework. Visual Studio .NET includes VB.NET, C#, managed C++, Jscript, and J# (allows using Java syntax for building .NET applications).

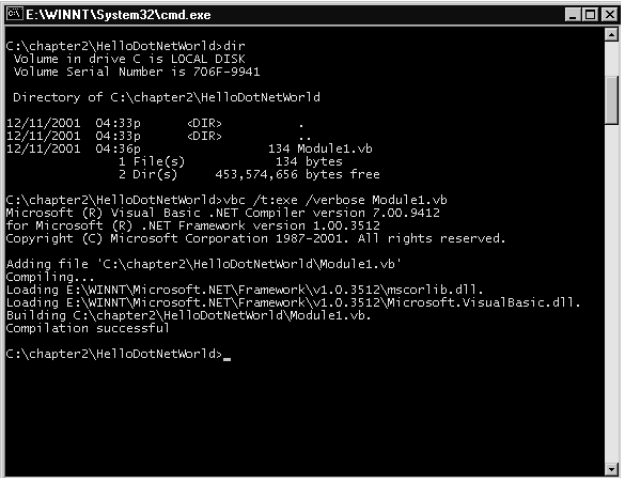
Since all .NET language compilers create MSIL code, there is theoretically no difference within the .NET Framework as far as performance goes. So the compelling performance or ease of coding reasons for developing in a specific language has disappeared. You can basically choose a language that you are comfortable with and not lose those major benefits from one language to another. Some say that choice of language is a “lifestyle” choice. The target we’ve chosen for this book is C#.

## Using the Compilers

As mentioned earlier, you can build .NET applications using Notepad and the command-line compilers if you want to. The compilers that ship with the SDK are `csc.exe` (C#), `vbc.exe` (VB.NET), and `jsc.exe` (Jscript .NET). Figure 2.7 depicts a subdirectory with one file called `Module1.vb`. The command `vbc /t:exe /verbose Module1.vb` translates to:

- `vbc` = Run the `vbc.exe` compiler with the following options
- `/t:exe` = The “target” for the compilation is an exe file
- `/verbose` = display all of the output information
- `Module1.vb` = the source file that we want to compile

**Figure 2.7** Example of Command Line Compilation Process



```
E:\WINNT\System32\cmd.exe
C:\chapter2\HelloDotNetWorld>dir
Volume in drive C is LOCAL DISK
Volume Serial Number is 706F-9941

Directory of C:\chapter2\HelloDotNetWorld

12/11/2001 04:33p <DIR>          .
12/11/2001 04:33p <DIR>          ..
12/11/2001 04:36p                134 Module1.vb
                   1 File(s)          134 bytes
                   2 Dir(s)          453,574,656 bytes free

C:\chapter2\HelloDotNetWorld>vbc /t:exe /verbose Module1.vb
Microsoft (R) Visual Basic .NET Compiler version 7.00.9412
For Microsoft (R) .NET Framework version 1.00.3512
Copyright (C) Microsoft Corporation 1987-2001. All rights reserved.

Adding file 'C:\chapter2\HelloDotNetWorld\Module1.vb'
Compiling...
Loading E:\WINNT\Microsoft.NET\Framework\v1.0.3512\mscorlib.dll.
Loading E:\WINNT\Microsoft.NET\Framework\v1.0.3512\Microsoft.VisualBasic.dll.
Building C:\chapter2\HelloDotNetWorld\Module1.vb.
Compilation successful

C:\chapter2\HelloDotNetWorld>
```

For C++ managed code you would compile your C++ code with the `/clr` compiler option. Please note that there are many restrictions to using the `/clr` option, so review the documentation and understand what you are doing before trying to use managed extensions.

Running the compilers from the command line gives you the same effect as setting project properties or compiler options in an integrated development environment like Visual Studio .NET. The example in Figure 2.7 is about as easy as it gets. Using Visual Studio .NET, or other integrated development environment for .NET development, sure manages that process more easily.

## Tools

There are a variety of tools provided with the .NET Framework SDK that aid developers in building better .NET applications. We cover a few of the more important ones here.

- ILDasm
- NGEN
- Debugging tools

### *ILDASM.exe*

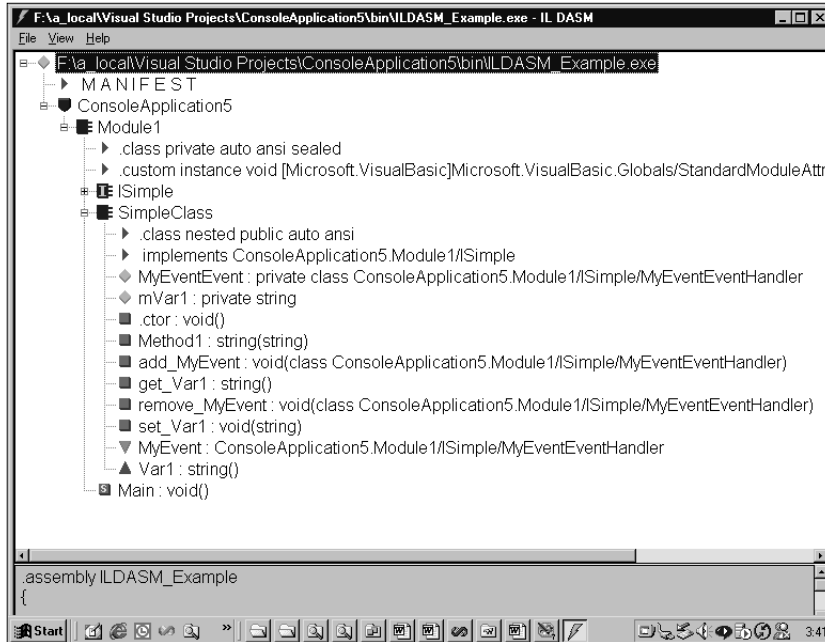
The Intermediate Language Disassembler (ILDasm) is a tool that .NET developers need to have at their fingertips. Similar to Java class path problems, a great deal of .NET problems with the compiler not being able to find classes is due to namespace issues. ILDasm allows you to look at an assembly and see the metadata for every element in the assembly. Double-clicking on an element actually brings up the MSIL so you can see exactly what the compiler did to your code. So if you receive an exception concerning types that can't be found by the compiler, looking at the metadata for the assembly quickly allows you to locate the type in question and where it is located. Figure 2.8 shows an example of what ILDasm can show. Double-clicking on an element shows the intermediate language code for that element so you can see the code created by the compilers.

### *NGEN*

The place where the machine specific assembly code is located is called the Native Image Cache. You can see this in the `\winnt\assembly` directory. Each assembly has a directory for the assembly, and a corresponding native image directory. When the CLR needs to run code from an assembly, it first looks in

this cache to see if it can avoid having to recompile the assembly to machine code and then run.

**Figure 2.8** ILDasm.exe View of a .NET Executable (DLL or EXE)



What NGEN.exe provides is the compilation to native image code up front. So you would probably want to do this on your production machine when a new build is ready for deployment. Obviously, any change and redeployment of the assembly would require a different version to be deployed, or run another NGEN compilation so that the existing outdated native image code won't be run. This is because the CLR always looks in the native directory first. Figure 2.9 shows the various command line switches available to you for NGEN.

**Figure 2.9** NGEN.exe Options

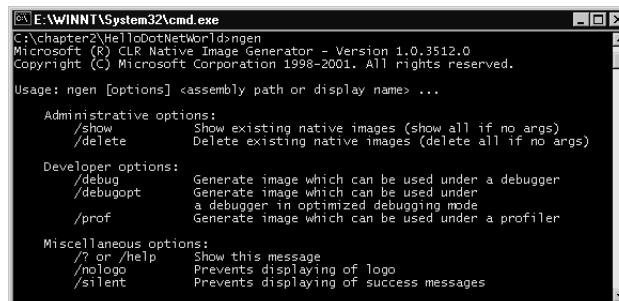
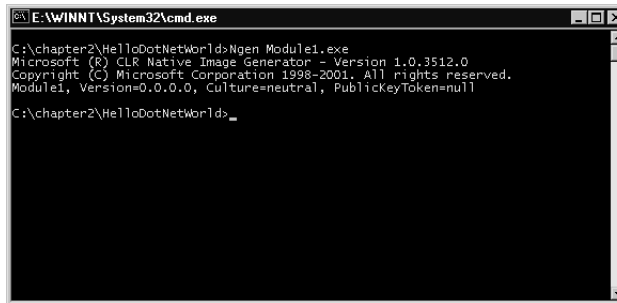


Figure 2.10 shows the output of a successful compilation. Note that the output displays all of the versioning information for the assembly that you ran NGEN on. Assuming that you would use NGEN for code that you're putting into production, displaying the full version information for assembly gives you a chance to double check that the proper assembly is being compiled into machine code.

**Figure 2.10** Successful NGEN.exe Compilation



```

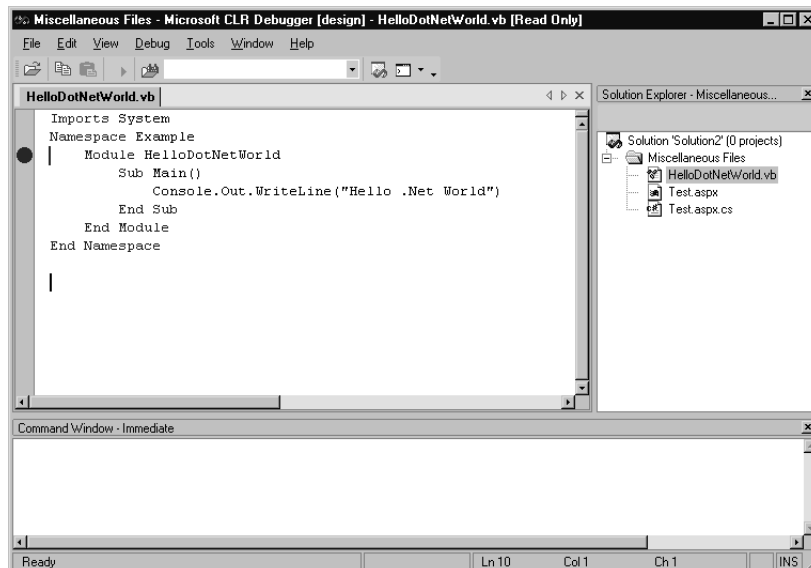
E:\WINNT\System32\cmd.exe
C:\chapter2\HelloDotNetWorld>Ngen Module1.exe
Microsoft (R) CLR Native Image Generator - Version 1.0.3512.0
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.
Module1, Version=0.0.0.0, Culture=neutral, PublicKeyToken=null
C:\chapter2\HelloDotNetWorld>_

```

## Debugging Tools

The two debuggers that are included with the .NET Framework SDK download are *CorDbg* and *DbgClr*. *CorDbg* is a command-line debugger. *DbgClr* is a Windows GUI debugger. Figure 2.11 is an example showing *DbgClr*. It looks very similar to the Visual Studio .NET integrated development environment.

**Figure 2.11** *DbgClr.exe* GUI



In both tools you can perform cross-language debugging. You can also attach and debug running processes. For both tools, you must have already compiled the code in question with the */debug* switch. These are tools that you need to take out and play with to get the most out of them.

## Base Class Libraries

In order to use the services that the .NET Framework provides, the Framework exposes a number of base classes that are easily referenced and used. Some of the more important ones are shown in Table 2.1 to give you an idea of how the base classes are packaged.

The .NET Framework is “strongly typed”. This means that everything is an object. And the objects follow a hierarchy starting from *System.Object*. If you don’t explicitly code your class to inherit from another class, it is implied that your class inherits from *System.Object*. The *System* namespace is the “root” namespace for the Base Class Library. Everything is an object in the .NET Framework, and everything ultimately inherits from the *System.Object* type. Because it’s a given if no explicit inheritance is defined in your class, your class inherits from *System.Object*. All of the base data types are found in this namespace as well (for example *String*, *Byte*, *Array*, and so on). Some of the most basic services that the .NET Framework provides are handled by classes in the *System* namespace, such as exception handling, garbage collection, and the system environment. Take a look at Table 2.1 to give you an idea of the breadth of functionality provided by the Base Class Library classes.

**Table 2.1** Some Important Base Class Library Namespaces

Namespace	Purpose
<i>System.Data</i>	Namespace that is made up of the ADO.NET classes. Provides data access services for any type of application built for the .NET Framework.
<i>System.Collections</i>	Provides different types of classes and interfaces to create collections of objects. Some examples are <i>Arraylist</i> , <i>Hashtable</i> , <i>Dictionary</i> , <i>Stack</i> , <i>SortedList</i> , and <i>Queue</i> . From the base collection types, you can easily create type specific collections.

Continued

**Table 2.1** Continued

<b>Namespace</b>	<b>Purpose</b>
<i>System.Configuration</i>	Allows access to the information in the configuration files that you store with your applications. If the default “handlers” (configuration file readers) don’t meet your needs you can create your own. All of the configuration files in the .NET Framework are in XML format.
<i>System.Diagnostics</i>	Used for debugging your code and allows for code tracing. Also included are classes that allow starting operating system processes, monitoring performance of the system, and writing information from the application to the operating system’s event log.
<i>System.IO</i>	File and directory classes provide information and operations. Reading/writing files and creating/using data streams.
<i>System.Net</i>	Provides access to different network protocols. Most notable are the <i>WebRequest</i> and <i>WebResponse</i> classes. The classes in this namespace wrap functionality for network and Internet communications.
<i>System.Reflection</i>	A very powerful group of classes and interfaces that allow runtime access to the metadata for your assemblies. With this information you could create objects and run methods on those objects at runtime which gives you an enormous amount of flexibility in your applications. You can also use reflection to view any custom information that you created and stored within your assemblies.
<i>System.Runtime.InteropServices</i>	Classes provide functionality for accessing COM objects and the Windows API components from the CLR. These classes can wrap COM components with .NET classes and add functionality into .NET classes so that

Continued

**Table 2.1** Continued

Namespace	Purpose
	calls can be made to COM components. Either way it's a performance hit to do this.
<i>System.Runtime.Remoting</i>	The <i>System.Runtime.Remoting</i> namespace provides for the creation and configuration of distributed applications. This gives you the ability to reference objects on remote machines, similar in some ways to what DCOM provides.
<i>System.Text</i>	<i>System.Text</i> contains classes for string manipulation and formatting, and character set encoding/decoding. You can improve performance if you manage your strings wisely.
<i>System.Web, System.Windows.Forms</i>	These namespaces provide for the creation of ASP.Net pages and win forms. Including classes and events associated with both.
<i>System.Threading</i>	Provides classes and interfaces that enable multithreaded programming. Thread pooling is included with classes for managing your threads. This is a change for VB6 developers used to the single-threaded apartment (STA) model.
<i>System.XML</i>	Provides the functionality to create and read standard XML documents. Support for XML schemas, SOAP, and XSL/T transformations are included. An XML query language called XPath is also found in this namespace.

This is just a fraction of the Base Class Library. You would be well served by going through the documentation provided with the SDK, and walk through the samples provided with the SDK. It's a great starting point for learning what functionality you have been provided with right out of the box. To use a Base Class Library, you can use the full namespace path or the short way. Figure 2.12 is a simple example of using a Base Class Library namespace to take advantage of its functionality using VB.NET.



**Figure 2.12** Using a Namespace from the Base Class Library

---

```
Imports System
Imports System.Collections

Module MyListModule1

    Sub Main()
        Dim myList As New ArrayList()

        myList.Add("ItemOne")
        myList.Add("ItemTwo")

        Console.Out.WriteLine("The first item is " & _
            myList.Item(0))

    End Sub

End Module
```

---

In this example the *Imports* statement is where you tell the compiler that you are referencing the *System* namespace. That allows us to use short notation to reference the associated namespace. If you didn't want to use the *Imports* statement, then to use the *Console* class you would be required to include the full namespace notation (*System.Console.Out.WriteLine()*).

### *Structured Exception Handling Revisited*

Structured exception handling gives you the ability to anticipate what errors could occur, and enclose the code that may cause those particular errors. If an error is "caught" in your monitored code you can invoke a predefined handler than can respond in a manner that will keep you up and running.

The .NET Framework provides its own exceptions for exceptions such as *FileNotFoundException*, *IndexOutOfRangeException*, and *SecurityException*. You have the ability to create very complex error-handling schemes by inheriting from the *System.Exception* and associated exception classes to create user-defined errors and a common way of handling all exceptions in your application.

You can also grab the stack information at the time of the exception and pass that back up through the layers to where you want to evaluate and/or save that

information for troubleshooting purposes. Stack trace information contains most of the information located on the stack at the time of the exception, so you can see what methods were called and exactly where the error occurred. Figure 2.13 below shows an example of using the *Try–Catch–Finally* block. The descriptions for the lines of code in Figure 2.13 are listed in Table 2.2

### Figure 2.13 Structured Exception Handling

---

```
Imports System
10 Imports System.IO
20 Public Class FileExceptionExample
30 Public Sub DoFileStuff(ByVal strFileName as String)
40 Try
50     Dim fs As New FileStream("c:\myfile.txt",
        FileMode.Open)
60     'File operations go here ....
70     'When file operations complete then close the stream
80     fs.Close()
90
100 Catch fnfException As FileNotFoundException
110     Throw New FileNotFoundException("Hey file isn't
        there", fnfException)
120 Catch eosException As EndOfStreamException
130     Throw New EndOfStreamException("put your message
        in here")
140 Catch e As Exception
150     Throw New Exception("put your message in here")
160
170 Finally 'optional
180
190 End Try
200
210 End Sub
```

---

**Table 2.2** Line Number Descriptions for Figure 2.13

Line Numbers	Description
40-100	Performing a file input/output operation. From experience, you may know that several different exceptions could occur with accessing the file. With that in mind we place the <i>Try</i> statement above the code that could cause exceptions.
100	A likely problem could be that the passed in string for the file name ( <i>strFileName</i> ) doesn't correspond to a file at the location specified. This is a problem that we would want to catch in our code so we place a <i>Catch</i> statement for the <i>FileNotFoundException</i> .
110	The <i>FileNotFoundException</i> is handled by passing to the method that called the <i>DoFileStuff</i> method, a message and the exception object variable (e) so that it can handle the exception appropriately. The <i>throw</i> statement does that for us. You have a few different options for the <i>throw</i> method as far as what you return to the calling method. We could handle the exception here, on line 110 instead of throwing it, and continued on with operation. If you don't throw the exception, then the code in the <i>Finally</i> block (line 170) will execute, and operation will continue with the code after the <i>Finally</i> block.
120	In this line you're catching an <i>EndOfStreamException</i> . You can catch multiple exceptions for a given <i>Try</i> block. When an exception occurs the CLR looks at the <i>Catch</i> statements from top to bottom to find the proper handler for the exception that occurred.
140	This <i>Catch</i> statement says, "If an exception occurs that I haven't anticipated and coded a <i>Catch</i> block for, I'll take care of it here".
170	The <i>Finally</i> statement is optional. This is where you would write "cleanup" code to release resources that may have been obtained before the exception. For example: When performing data operations you open a data connection, then get an <i>SQLException</i> when you try to access the data. In this case you would want to release that data connection.
190	The <i>End Try</i> is placed here to completely wrap our exception handling in the <i>Try</i> block. You can have multiple <i>Try</i> blocks in a method, and you can nest <i>Try</i> blocks.

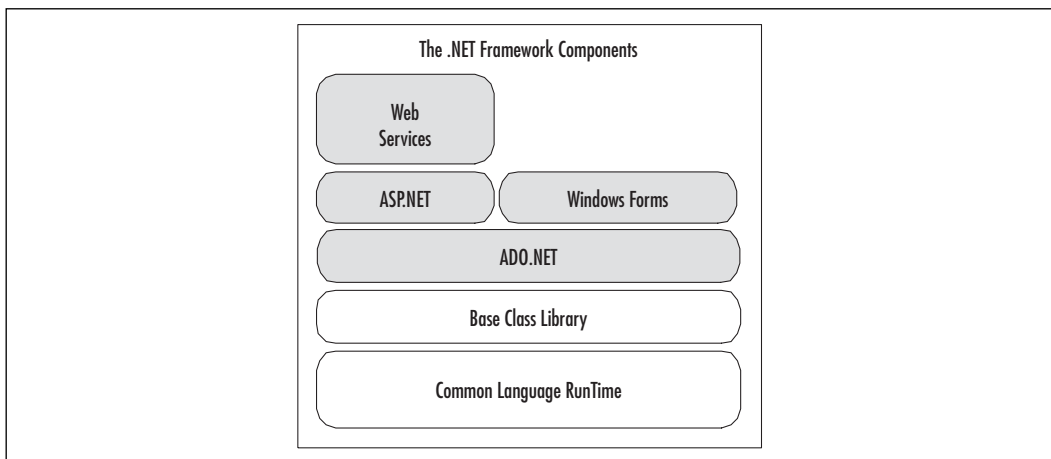
Generally, in our opinion, you want to try to handle the exception as close as possible to where it occurs. Most large enterprise applications have a "global" error handler that does things like provide notifications to the user of the

exception, logs the exception, and sends notifications via e-mail or pager to support personnel, all depending on the severity of the error.

## Components in the .NET Framework

The .NET Framework’s “layered” approach provides a firm foundation for building .NET applications. The CLR is at higher level of abstraction than the Windows operating system. The Base Class Library provides a higher level of abstraction than the CLR. Now in Figure 2.14, you can see that the major components for building .NET application like ASP.NET are at a higher level of abstraction still. Each layer makes development more productive, while still allowing the ability to drill down into the lower levels to get at specific functionality. The different frameworks also provide excellent models for developing your own frameworks. And you can always subclass the existing functionality to create your own.

**Figure 2.14** The .NET Framework with Major Components



## ASP.NET

As shown in Figure 2.14 ASP.NET is a framework that sits architecturally on top of the .NET Framework. So it takes the Base Class Libraries for Web development and abstracts them to a higher level to make creating Web applications even easier.

Web Form pages are the central display mechanism for ASP.NET applications. They are made up of .aspx files compiled with an associated .NET language class file that contains the code. This is known as *code-behind* and is the default when you create a new Web Form in Visual Studio .NET. It is possible to

put the code directly on the same page as the HTML but we don't recommend it. The whole idea is to separate the display from the logic. The Web Form pages are compiled prior to execution and cached so performance overall should prove to be better than the interpreted Active Server Pages (ASP).

Performance is also improved by the new caching classes that enable *smart caching*. That gives you the ability to store information that doesn't change much, but is expensive to obtain (like through database operations) in memory. It's called the cache, and when that information is needed, it's grabbed from memory, instead of having to perform the data operations. You can set up time durations for the cache and dependencies such as if file *x* changes then empty the cache of *y* information. There is even a caching namespace, called *System.Web.Caching*, to give you very low-level control over what you can cache. Careful use of this feature offers significant performance enhancement.

ASP.NET gives you server side controls which expose you to events and server side processing that didn't exist in traditional ASP. The page is created on the server, and pure HTML is rendered back to the client so they can be run on any browser. Visual Studio .NET is an awesome environment for creating ASP.NET Web applications. The VB-like ability to drag and drop controls and position them on the forms, double-clicking on the forms controls to call up the code and create events for that control, and the debugging make building Web applications fun. It removes the drudgery that was experienced with ASP development.

## ADO.NET

As shown in Figure 2.14 ASP.NET has full access to the *System.Data* namespace of ADO.NET. ADO.NET provides data management services to all .NET Framework components. ADO.NET is similar to ADO, therefore the learning curve is not as steep to get up to speed with ADO.NET. The additional object-oriented nature of ADO.NET is the major difference.

In the .NET Framework SDK only two data “managed providers” are included: The OLEDB managed provider, and SQLServer 7 and above managed provider. Data operations using the SQLServer provider enable better performance, as there is no middle layer of translation that you will find with the OleDB managed provider. The “middle layer” that I'm referring to is the OLEDB/ODBC layer that the OLEDB Data Provider requires for operations. The SQLServer provider communicates via the tabular data stream (TDS) protocol which is the native protocol for SQLServer. This optimization is what provides the major performance difference between the SQLServer and OLEDB data providers.

*DataReader* classes are provided in ADO.NET that process a data stream in a read-only, forward-only manner. Currently there is an *SqlDataReader* for SQLServer, and *OleDbDataReader* for the other data providers. This provides the most efficient way to get read-only data for filling simple lists and tables on your Windows Forms and Web Forms.

When you need update ability the *DataSet* object will do the trick. This is really an in memory database that allows you to get your data and work with it just like you would data in a normal database. This includes have multiple tables and relationships involved and maintained in *DataSet*. You can tell the *DataSet* to update the main datastore and refresh the *DataSet* as often as you need to. *DataSets* can be created and populated manually and used simple as a temporary database. The data in a *DataSet* is persisted as XML which can be very useful when you need to transport that data over the Internet, or to applications running on different platforms like mainframe or UNIX.

ADO.NET focuses on obtaining data from a data source, then disconnecting from that data source and managing the how the disconnected data is used. Using disconnected data makes the most sense in a Web environment. One of the things you look for when developing applications for the Web is to make sure that if you are performing operations that carry a high resource overhead, like setting and maintaining data connections, you try to get rid of those resources as soon as you're done with them. Having a data connection open just long enough to get a disconnected set of data allows the server to reclaim those resources sooner, so the server will be able to handle more users. If possible, having the data set on the client to be worked with and updated in batch, minimizes round trips to the server, which improves scalability as well.

## VB.NET

VB.NET takes the impression of non-VB programmers that VB is a “toy” language, and dispels that once and for all. You still have the productivity that VB has always been famous for. But now you also have the fully object-oriented language features and the ability to actually see what is going on behind the scenes. Nearly every feature that is available to the other .NET languages is open to VB.NET. All of this new power will require VB developers that don't have object-oriented programming experience to increase their knowledge in that area. Building scalable enterprise applications requires careful design. The good news is, if developers have been building object-oriented applications and components in VB using the Windows DNA architecture, then the learning curve is not as steep. And there are tools you

can use and steps you can take to move your current applications closer to .NET so your migration will be as painless as possible.

Since VB.NET is targeted at the .NET Framework, it carries with it all of the benefits and available services provided by the Framework. The .NET Framework also enables interoperability between objects you create with any .NET programming language. Some big language differences between VB.NET and previous versions are listed here:

- Structured exception handling.
- VB.NET is now a fully object-oriented language including inheritance.
- VB.NET is fully integrated into the .NET Framework with access to the same services provided to C#, managed C++, and the growing number of .NET Framework-targeted languages.
- Delegates have been included. Delegates are type-safe, object-oriented function pointers. They do more than simple function pointers in C++, and are a big difference between .NET languages and Java. Delegates can handle more than one method. You can also use delegates to identify event handlers and are also used with multithreaded applications.
- The threading model has been changed from single-threaded apartment to free threading.

## NOTE

Visual Basic 6.0 used the single-threaded apartment threading model. In the STA model, a component's methods are tied to a single thread (apartment). Therefore, any other methods for that component that could run have to wait for the running method to finish. The .NET Framework uses a multithreaded apartment (MTA) threading model, which allows multiple threads to be available to an apartment. This adds to the responsiveness that the user experiences.

An upgrade wizard is provided with Visual Studio .NET that will help you upgrade existing VB6 applications to VB.NET. The wizard also creates an upgrade report that documents the upgrade process and any errors that occurred during the process. Figure 2.15 shows a simple example of writing "Hello .NET World" to the console window.

**Figure 2.15** Hello .NET world in VB.NET

---

```
Imports System
Namespace Example
    Module HelloDotNetWorld
        Sub Main()
            Console.Out.WriteLine("Hello .NET World")
        End Sub
    End Module
End Namespace
```

---

## C#

C# is a language created from the ground up. It has taken the best from existing object-oriented languages, like Java and C++, and improved on perceived shortcomings from those languages. The intent was to create a language that would give the productivity of VB and the power of C++. C# is centered on building components. There have been many comparisons between Java and C#. In our opinion, C# has many similarities to Java and offers more.

Any Java or C++ programmer will have few problems reading and understanding the C# syntax. Like Java, it has taken the best of C++ and left out the areas that can cause the most headaches, such as function pointers or the ability to step on unauthorized memory locations that caused blue screen of death.

Figure 2.16 shows the same simple program as seen in Figure 2.15. Note the minor syntax changes between the C# example and the VB.NET examples.

**Figure 2.16** Hello .NET world in C#

---

```
using System;
namespace HelloDotNetWorld
{
    public class HelloDotNetWorld
    {
        public static void Main()
        {
            Console.Out.WriteLine("Hello .NET World!");
        }
    }
}
```

---



In most cases the difference between C# and VB.NET is in the syntax. So again, choice of language is really up to developer preference. The expectation is that C++ and Java developers will gravitate to C#, and VB developers will gravitate towards VB.NET. Time will tell.

## Windows Forms

The .NET Framework has two primary user interfaces, ASP.NET Web Forms, and Windows Forms. Windows Forms is a framework that provides the new platform for Microsoft Windows application development, based on the .NET Framework. This framework provides a clear, object-oriented, extensible set of classes that enable you to develop what is known as rich Windows applications. They are applications that allow you to take advantage of what the Windows operating system provides for creating user interfaces.

The *System.Windows.Forms* namespace provides the classes that enable you to create rich client applications. Because everything in the .NET Framework is a type, you can reuse and extend your forms classes by inheriting from them and then making whatever additions or changes you may need to.

The *Form* class is simply a window-like container for the controls that you place on the form. You have the ability to derive from the *System.Windows.Forms.Form* class to create new forms or use an existing form as a template and derive from the pre-existing form. Creating form templates could save a lot of work for larger applications. Any changes to the *base Form* class are automatically available to the other *Form* classes that inherit from that base class.

As with the *Form* class, you can use an existing control or inherit from an existing control to create your own. Windows Forms controls are called rich controls because they carry with them the rich Windows user interface and formatting. In comparison controls in ASP.NET Web Forms are limited to what HTML 3.2 Web browsers can provide as far as user interface.

Control layout has been enhanced with the Windows Forms Framework. If your forms can be resized at run time, and you want the controls to resize with the form, you would use the *Dock* property. This sets the controls position within the form and maintains that during resizing.

If you require that the control maintain a certain distance between the control and its container upon a resizing of the form you would use the *Anchor* property. You can anchor a control to one or more sides of its container so that “anchor” position is maintained with respect to the chosen sides of the container.

*Anchor* and *Dock* properties are also available at runtime. You have an unlimited number of different combinations of containers/controls/control properties so that you can set up your forms in any way you like.

With Visual Studio .NET you drag and drop the rich set of Windows controls on to the Form design interface just like in previous versions of Visual Basic

## Web Services

Web services are the latest and greatest “new thing” to be hyped in the industry. Microsoft, IBM, Sun and other companies are getting on the Web services bandwagon. Web services are basically components that can be referenced and their methods run via standard Web protocols, like SOAP (Simple Object Access Protocol) and XML. The industry is moving towards standard ways of describing the functionality that a Web service provides, and how a Web service is used. Since standard Web protocols are used it shouldn't matter what language the Web service is written in or what platform the Web service is running on. As long as you communicate over HTTP with XML and are using SOAP you should be able to send information to and receive information from a Web service. Because the communication is over HTTP it can make it through firewalls without any problem but the ability to encrypt the communication and/or use Secure Sockets Layer is also available to provide the same level of security as ASP.NET Web applications.

As long as a client application has Internet access, it can utilize a Web service. This includes interaction with other Web services, ASP.NET Web applications, and rich Windows Forms applications. The vision of the next incarnation of the Internet is made up of applications utilizing functionality from Web services all over the Internet. And users pay for the service every time they access the Web service. This means that a company that provides a particular set of functionality can now open that functionality to the world on the Internet. A totally new revenue stream is the vision.

For developers using Visual Studio .NET accessing a Web service for its functionality is as easy as adding a reference to the service and then coding to that reference just like it was a component on your machine. Creating a Web service is almost as simple. Create a Web service project and every method that is to be exposed publicly from your Web service you put the `<WebMethod>` attribute above the method's signature.

Web Services are integrated with the ASP.NET framework so they have the same set of services available to them as ASP.NET applications have. Some examples are data caching, ADO.NET, session state, and security.

## Summary

We've covered a lot in this chapter. We started with obtaining and installing the .NET Framework SDK. There can be some confusion on the system requirements. Just be careful that you meet them so that you can save yourself a lot of headaches. The .NET SDK install is very easy. The thing that does catch some people is the requirement for ASP.NET that IIS be installed first.

We went over the Common Language Runtime, the “sandbox” for .NET program execution. The CLR manages the code you write from verifying access, finding the proper assembly, through code execution. The code you write and compile for the CLR is converted to Microsoft Intermediate Language (MSIL) and is called managed code. When the CLR needs the code executed it Just-In-Time compiles it then runs it. When the objects are no longer referenced garbage collection will take care of them as appropriate.

Because all .NET languages compile to MSIL you have language interoperability between .NET languages for inheritance, referencing objects, error handling, and debugging. Interoperability with “unmanaged” code is also possible with the CLR turning over control of that code to the Windows operating system.

Deployment is much less of a hassle with versioning. Versioning defines an executable in different ways (name, version, culture, public key). This allows multiple versions of the same file to be deployed on the same machine, with no conflicts. There are different ways to bundle your files for deployment from XCOPY to Windows Installer packages and CAB files.

Metadata makes the executables self-describing which opens up many possibilities for dynamic loading of objects and method calls at runtime. The metadata contains information for every element managed by the runtime along with the MSIL. This makes an assembly a complete deployable unit. Therefore we have no need for the Windows registry.

Namespaces allow you to create scope for your common groups of functionality, and provide another way to disambiguate one assembly from another with the same name. The base class library provides excellent examples of how to intelligently use namespaces.

In the “programming” section we discussed some more important concepts that should help you get a jump start on your development. Once you become familiar with the base class library you will be well on your way to “becoming one” with the .NET Framework. ILDasm.exe and the debugging tools will make your development time more efficient and more pleasurable. NGen.exe compiles your code

to the native machine codes so the CLR will bypass the Just-In-Time compilation and run the machine code directly significantly improving performance.

The last section was on the .NET Framework's major components (ASP.NET, ADO.NET, Windows Forms, and Web Services). Most of these are actually frameworks in themselves put there to make our lives easier. You can use the frameworks provided, or create your own by using the base class library directly. But so far we've been able to get by with very little inheriting from the ASP.NET or the Windows Forms frameworks. Microsoft has really spent a lot of time and money researching developer productivity issues and the Visual Studio .NET and the .NET Framework show the results.

Two of the components, ASP.NET and ADO.NET have their own chapters so we'll revisit them again. We're sure with practice, and the help of the other .NET related books in the Syngress library, you'll find programming for the .NET Framework as fun as we have.

## Solutions Fast Track

### Obtaining the .NET Framework

- ☑ Ensure that your computer meets the minimum requirements for running the .NET Framework. This will save you much time and trouble.
- ☑ [www.microsoft.com/net](http://www.microsoft.com/net) is the main site for downloading.
- ☑ Follow the instructions carefully if you download the multiple small files. If you choose to download the one large file give yourself plenty of time. It's about 120Meg in size.

### Installing the .NET Framework

- ☑ Remember that even though the .NET Framework is very stable and solid, any Beta software should be installed on non-critical machines.
- ☑ You can save a lot of time by making sure that your machine meets the minimum system requirements.
- ☑ The Visual Studio .NET install includes the .NET Framework SDK.

## Common Language Runtime

- ☑ The CLR is the primary engine of the .NET Framework.
- ☑ Code that you write in any .NET language is compiled to the Microsoft Intermediate Language (MSIL).
- ☑ When first requested to be executed the MSIL is then Just-In-Time compiled to the applicable machine code for the CPU.

## Developing Applications with the .NET Framework

- ☑ It is possible to write code with the .NET SDK using a simple text editor, and compile the code using the command line.
- ☑ ILDasm.exe allows you to see all of the types in your .NET executable. Double-Clicking on any of the types will bring up a window displaying the MSIL code for that type.
- ☑ Since all .NET languages compile to MSIL, your choice for of language for .NET development can be based on personal preference without taking a major performance hit.
- ☑ Using NGEN.exe on an assembly when you place it into production eliminates the need for the CLR to compile the MSIL when executing code from that assembly.
- ☑ Become one with the base class libraries. It's a lot of functionality for free.
- ☑ Take a hard look at structured exception handling.

## Components in the .NET Framework

- ☑ The major components in the .NET platform utilize the services provided by the base class library, and in most cases abstract that functionality to a higher level to make our lives easier. An example would be the ASP.NET framework allowing us to program for the Web at a much higher level then would be necessary if we just used the base class library.
- ☑ Windows Forms and ASP.NET Web Forms are the user interface frameworks for .NET development.

- ☑ ADO.NET provides the services for data access in the .NET Framework. It is optimized for performance (disconnected datasets) and productivity (object oriented data access classes).
- ☑ Web Services are components residing on a Web server accessible from anywhere using standard Web protocols.

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** I installed Windows 2000 (professional or server), then installed the .NET Framework SDK. Why can't I create a Web project?

**A:** IIS 5.0 is a Windows Component that isn't part of a normal Windows 2000 install unless you specifically set that as an install option. IIS 5.0 must be installed prior to the .NET Framework SDK install or ASP.NET won't register correctly. To recover you must use `regsvr32.exe` to register the `Aspnet_isapi.dll`.

**Q:** I want to install the .NET Framework SDK on Windows NT 4.0 server, can I do that?

**A:** To install on Windows NT 4.0 server you must have service pack 6a applied.

**Q:** Where can I find the install for ASP.NET?

**A:** ASP.NET ships as part of the .NET Framework SDK?

**Q:** What is ASP.NET premium?

**A:** ASP.NET premium is a separate download and install giving you everything you need to develop ASP.NET applications. It includes the .NET Framework Redistributable, core ASP.NET support, and adds additional support for maintaining ASP.NET session state in a Web farm. In addition there is support for output caching and secure hosting. It basically gives you more functionality than the ASP.NET support in the .NET Framework SDK. Possibly by the time the .NET Framework SDK is out of beta and in production this functionality will be integrated with the SDK.



## XML Fundamentals

### Solutions in this chapter:

- An Overview of XML
  - Processing XML Documents Using .NET
  - Reading and Parsing Using the XmlTextReader Class
  - Writing an XML Document Using the XmlTextWriter Class
  - Exploring the XML Document Object Model
  - Querying XML Data Using XPathDocument and XPathNavigator
  - Transforming an XML Document Using XSLT
  - Working with XML and Databases
- 
- ☑ Summary
  - ☑ Solutions Fast Track
  - ☑ Frequently Asked Questions



# Introduction

The Extensible Markup Language (XML) is the latest offering in the world of data access. Microsoft has been actively supporting this language since its conception. XML provides a universal way for exchanging information between organizations. Its structure makes it perfect for online applications and working with data residing on the local or remote data sources.

Like Hypertext Markup Language (HTML), XML is a tag-based markup language. Many other technologies, such as browsers, JavaScript, VBScript, Dynamic HTML (DHTML), and Cascading Style Sheets (CSS), were developed to support the HTML documents. Similarly, XML cannot be singled out as a stand-alone technology. It is actually a family of a growing set of technologies and frameworks. The major members of this family are XML parsers, Extensible Stylesheet Language Transformations (XSLT), XPath, XLink, Simple API for XML (SAX), Schema Generators, and Document Object Model (DOM), just to name a few.

Please take note that ADO.NET *is not* coded in XML but that ADO.NET revolves around XML. Some readers may confuse the terms. Microsoft has integrated the XML technology in its .NET Framework rather tightly. The core foundation of the entire ADO.NET architecture is built upon XML. The ADO.NET itself is not coded in XML; however, it provides the facilities to apply various existing and emerging XML technologies to manipulate data and information. The *System.XML* namespace offers perhaps the richest collection of classes for generating, transmitting, processing, and storing information via XML. In this chapter, we will first have a brief introduction to the structural components of an XML document. Then we will look into the architecture of the XML objects in the .NET Framework. Finally, we will study several major XML.NET objects with many examples.

## An Overview of XML

XML is fast becoming a standard for data exchange in the next generation's Internet applications. XML allows user-defined tags that make XML document handling more flexible than HTML, the conventional language of the Internet. Since XML is the heart and soul of ADO.NET, sound knowledge of XML is imperative for developing applications in ASP.NET. The following section touches on some of the basic concepts of XML.

## What Does an XML Document Look Like?

The idea behind XML is surprisingly simple. The major objective is to organize information in such a way so that human beings can read and comprehend the data and its context; also, the document itself is technology and platform independent. Consider the following text file:

```
F10 Shimano Calcutta 47.76
F20 Bantam Lexica 49.99
```

Obviously, it is difficult to understand exactly what information the above text file contains. Now consider the XML document shown in Figure 3.1. The code is available in the `Catalog1.xml` file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 3.1** Example XML Document (`Catalog1.xml`)

```
<?xml version="1.0"?>
<!-- Chapter8\Catalog1.xml -->
<Catalog>
  <Product>
    <ProductID>F10</ProductID>
    <ProductName>Shimano Calcutta </ProductName>
    <ListPrice>47.76</ListPrice>
  </Product>
  <Product>
    <ProductID>F20</ProductID>
    <ProductName>Bantam Lexica</ProductName>
    <ListPrice>49.99</ListPrice>
  </Product>
</Catalog>
```

The above document is the XML's way of representing data contained in a product catalog. It has many advantages. It is easily readable and comprehensible, it is self-documented, and it is technology independent. Most importantly, it is quickly becoming the universally acceptable data container and transmission format in the current information technology era. Well, welcome to the exciting world of XML!

## Developing & Deploying...

### XML and Its Future

XML is quickly becoming the universal protocol for transferring information from site to site via HTTP. Whereas, the HTML will continue to be the language for displaying documents on the Internet, the developers will start using the power of XML to transmit, exchange, and manipulate data using XML.

XML offers a very simple solution to a complex problem. It offers a standard format for structuring data or information in a self-defined document format. This way, the data are kept independent of the processes that will consume the data. Obviously, the concept behind XML is nothing new. XML happens to be a proper subset of a massive specification named SGML developed by W3C in 1986. The W3C began to develop the standard for XML in 1996 with the motivation that XML would be simpler to use than SGML but that it will have more rigid structure than HTML. Since then, many software vendors have implemented various features of XML technologies. For example, Ariba has built its entire B2B system architecture based on XML, many Web servers (such as Weblogic Server) utilize XML specifications for configuring various server related parameters, Oracle has included necessary parsers and utilities to develop business applications in its 8i/9i suites, and finally, the .NET has also embraced the XML technology.

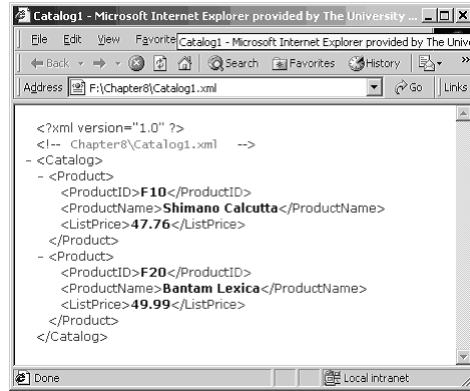
XML contains self-defined data in document format. Hence it is platform independent. It is also easy to transmit a document from a site to another site easily via HTTP. However, the applications of XML do not necessarily have to be limited to conventional Internet applications only. It can be used to communicate and exchange information in other contexts, too. For example, a VB client can call a remote function by passing the function name and parameter values using a XML document. The server may return the result via a subsequent XML document. Basically, that is the technology behind the SOAP (Simple Object Access Protocol).

## Creating an XML Document

We can use Notepad to create an XML document. VS.NET offers an array of tools packaged in the XML Designer to work with XML documents. We will demonstrate the usages of the XML Designer later. Right now, go ahead and

open the Catalog1.xml file Solutions Web site (www.syngress.com/solutions) for this book in IE 5.0 or higher. You will see that the IE displays the document in a very interesting fashion with drill-down features as shown in Figure 3.2.

**Figure 3.2** Catalog1.xml Displayed in IE



## Creating an XML Document in VS.NET XML Designer

It is very easy to create an XML document in VS.NET. Use the following steps to develop an XML document:

1. From the **Project** menu, select **Add New Item**.
2. Select the **XML File** icon in the **Add New Item** dialog box.
3. Enter a name for your XML file.
4. The VS.NET will automatically load the XML Designer and display the XML document template.
5. Finally, enter the contents of your XML document.

The system will display two tabs for two views: the XML view and the Data view of your XML document. These views are shown in Figures 3.3 and 3.4. The XML Designer has many other tools to work with. We will introduce these later in this chapter.

Figure 3.3 The XML View of an XML Document in VS .NET XML Designer

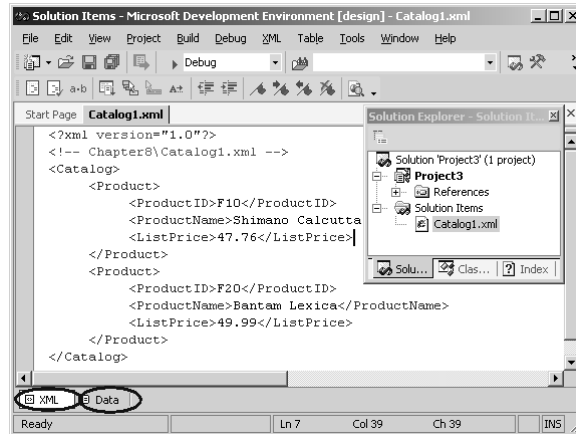
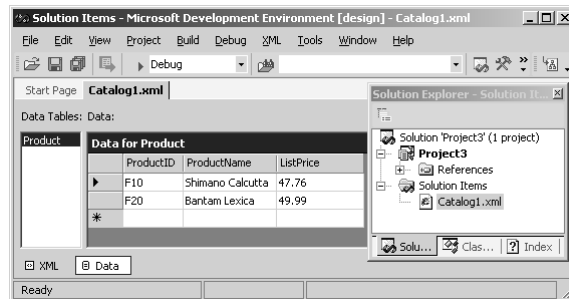


Figure 3.4 The Data View of an XML Document in VS.NET XML Designer



## Components of an XML Document

In this section, we will introduce the major components of an XML document. An XML document contains a variety of constructs. Some of the frequently used ones are as follows:

- Declaration** Each XML document may have the optional entry `<?xml version="1.0"?>`. This standard entry is used to identify the document as an XML document conforming to the W3C (World Wide Web Consortium) recommendation for version 1.0.
- Comment** An XML document may contain html-style comments like `<!--Catalog data -->`.
- Schema or Document Type Definition (DTD)** In certain situations, a schema or DTD may precede the XML document. A schema or

DTD contains the rules about the elements of the document. For example, we may specify a rule like “A product element must have a *ProductName*, but a *ListPrice* element is optional.” We will discuss schemas later in the chapter.

- **Elements** An XML document is mostly composed of elements. An element has a start-tag and end-tag. In between the start-tag and end-tag, we include the content of the element. An element may contain a piece of character data, or it may contain other elements. For example, in the *Catalog1.xml*, the *Product* element contains three other elements: *ProductId*, *ProductName*, and *ListPrice*. On the other hand, the first *ProductName* element contains a piece of character data like Shimano Calcutta.
- **Root Element** In an XML document, one single main element must contain all other elements inside it. This specific element is often called the root element. In our example, the root element is the *Catalog* element. The XML document may contain many *Product* elements, but there must be only one instance of the *Catalog* element.
- **Attributes** Okay, we agree that we didn’t tell you the whole story in our first example. So far, we have said that an element may contain other elements, or it may contain data, or both. Besides these, an element may also contain zero or more so-called attributes. An attribute is just an additional way to attach a piece of data to an element. An attribute is always placed inside the start-tag of an element, and we specify its value using the “*name=value*” pair protocol.

Let us revise our *Catalog1.xml* and include some attributes to the *Product* element. Here, we will assume that a *Product* element will have two attributes named *Type* and *SupplierId*. As shown in Figure 3.5, we will simply add the *Type*=“*Spinning Reel*” and *SupplierId*=“5” attributes in the first product element. Similarly, we will also add the attributes to the second product element. The code shown in Figure 3.5 is also available on the Solutions Web site for the book ([www.syngress.com.solutions](http://www.syngress.com.solutions)).



**Figure 3.5** *Catalog2.xml*

---

```
<?xml version="1.0"?>
<!-- Chapter8/Catalog2.xml -->
<Catalog>
```

---

Continued

**Figure 3.5 Continued**


---

```

<Product Type="Spinning Reel" SupplierId="5">
  <ProductID>F10</ProductID>
  <ProductName>Shimano Calcutta </ProductName>
  <ListPrice>47.76</ListPrice>
</Product>
<Product Type ="Baitcasting Reel" SupplierId="3">
  <ProductID>F20</ProductID>
  <ProductName>Bantam Lexica</ProductName>
  <ListPrice>49.99</ListPrice>
</Product>
</Catalog>

```

---

Let us not get confused with the “attribute” label! An attribute is just an additional way to attach data to an element. Rather than using the attributes, we could have easily modeled them as elements as follows:

```

<Product>
  <ProductID>F10</ProductID>
  <ProductName>Shimano Calcutta </ProductName>
  <ListPrice>47.76</ListPrice>
  <Type>Spinning Reel</Type>
  <SupplierId>5</SupplierId>
</Product>

```

Alternatively, we could have modeled the entire product element to be composed of only attributes as follows:

```

<Product ProductID="F10" ProductName="Shimano Calcutta"
  ListPrice = "47.76" Type="Spinning Reel" SupplierId= "5" >
</Product>

```

At the initial stage, the necessity of an attribute may appear questionable. Nevertheless, they exist in the W3C recommendation, and in most situations these become handy in designing otherwise-complex XML-based systems.

- **Empty Element** We have already mentioned a couple of times that an element may contain other elements, or data, or both. However, an element does not necessarily have to have any of them. If needed, it can be kept totally empty. For example, observe the following element:

```
<Input type="text" id="txtCity" runat="server" />
```

The *empty* element is a correct XML element. The name of the element is *Input*. It has three attributes: *type*, *id*, and *runat*. However, neither does it contain any sub-elements, nor does it contain any explicit data. Hence, it is an *empty* element. We may specify an empty element in one of two ways:

- Just before the “>” symbol of the start-tag, add a slash (/), as shown above, or
- Terminate the element using standard end-tag as follows:

```
<Input type="text" id="txtCity" runat="server" ></Input>
```

Examples of some empty elements are: `<br/>`, `<Pup Age=1 />`, `<Story></Story>`, and `<Mail/>`.

## Well-Formed XML Documents

At first sight, an XML document may appear to be like a standard HTML document with additional user-given tag names. However, the syntax of an XML document is much more rigorous than that of an HTML document. The HTML document enables us to spell many tags incorrectly (the browser would just ignore it), and it is a free world out there for people who are not case-sensitive. For example, we may use `<BODY>` and `</Body>` in the same HTML document without getting into trouble. On the contrary, there are certain rules that must be followed when we develop an XML document. Please, refer to the <http://W3C.org> Web site for the details. Some basic rules, among many others are as follows:

- The document must have exactly one root element.
- Each element must have a start-tag and end-tag.
- The elements must be properly nested.
- The first letter of an attribute’s name must begin with a letter or an underscore.
- A particular attribute name may appear only once in the same start tag.

An XML document that is syntactically correct is often called a *well-formed* document. If the document is not well-formed, Internet Explorer will provide an error message. For example, the following XML document will receive an error message, when opened in Internet Explorer, just because of the case sensitivity of the tag `<product>` and `</Product>`.



```
<?xml version="1.0"?>
<product>
<ProductID>F10</ProductID>
</Product>
```

## Schema and Valid XML Documents

An XML document may be well formed, but it may not necessarily be a valid XML document. A valid XML document is a document that conforms to the rules specified in its Document Type Definition (DTD) or Schema. DTD and Schema are actually two different ways to specify the rules about the contents of an XML document. The DTD has several shortcomings. First, a DTD document does not have to be coded in XML. That means a DTD is itself not an XML document. Second, the data-types available to define the contents of an attribute or element are very limited in DTD. This is why, although VS.NET allows both DTD and schema, we will present only the schema specification in this chapter. The W3C has put forward the candidate proposal for the standard schema specification ([www.w3.org/XML/Schema#dev](http://www.w3.org/XML/Schema#dev)). The XML Schema Definition (XSD) specification by W3C has been implemented in ADO.NET.VS .NET supports the XSD specifications.

A schema is simply a set of predefined rules that describe the data contents of an XML document. Conceptually, it is very similar to the definition of a relational database table. In an XML schema, we define the structure of an XML document, its elements, the data types of the elements and associated attributes, and most importantly, the parent-child relationships among the elements. We may develop a schema in many different ways. One way is to enter the definition manually using Notepad. We may also develop schema using visual tools, such as VS.NET or XML Authority. Many automated tools may also generate a rough-cut schema from a sample XML document (similar to reverse-engineering).

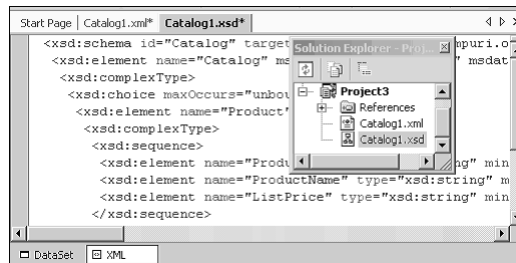
If we do not want to code a schema manually, we may generate a rough-cut schema of a sample XML document using VS.NET XML Designer. We may then polish the rough-cut schema to conform to our exact business rules. In VS.NET, it is just a matter of one click to generate a schema from a sample XML document. Use the following steps to generate a rough-cut schema for our `Catalog1.xml` document (shown in Figure 3.1):

- Open the **Catalog1.xml file** in a VS.NET Project. VS.NET will display the XML document and its XML View and the Data View tabs at the bottom.

- Click on the **XML** menu pad of the Main menu.

That's all! The systems will create the schema named `Catalog1.xsd`. If we double-click on the **Catalog1.xsd** file in the **Solution Explorer**, we will see the screen as shown in Figure 3.6. We will see the *DataSet* view tag and the XML view tag at the bottom of the screen. We will elaborate on the *DataSet* view later in the chapter.

**Figure 3.6** Truncated Version of the XSD Schema Generated by the XML Designer



For discussion purposes, we have also listed the contents of the schema in Figure 3.7. The XSD starts with certain standard entries at the top. Although the code for an XSD may appear complex, there is no need to get overwhelmed by its syntax. Actually, the structural part of an XSD is very simple. An element is defined to contain either one or more *complexType* or *simpleType* data structures. A *complexType* data structure nests other *complexType* or *simpleType* data structures. A *simpleType* data structure contains only data.

In our XSD example (Figure 3.7), the *Catalog* element may contain one or more (unbounded) instances of the *Product* element. Thus, it is defined to contain a *complexType* structure. Besides containing the *Product* element, it may also contain other elements (for example, it could contain an element *Supplier*). In the XSD construct, we specify this rule using a *choice* structure as follows:

```
<xsd:element name="Catalog" msdata:IsDataSet="true">
  <xsd:complexType>
    <xsd:choice maxOccurs="unbounded">
      --- --- ---
      --- --- ---
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Because the *Product* element contains further elements, it also contains a *complexType* structure. This *complexType* structure, in turn, contains a sequence of *ProductId*, and *ListPrice*. The *ProductId* and the *ListPrice* do not contain further elements. Thus, we simply provide their data types in their definitions. The automated generator failed to identify the *ListPrice* element's text as decimal data. We converted its data type to decimal manually. The complete listing of the *Catalog.xsd* is shown in Figure 3.7. The code is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

## NOTE

An XSD is itself a well-formed XML document.

**Figure 3.7** Partial Contents of *Catalog1.xsd*

```
<xsd:schema id="Catalog"
  targetNamespace="http://tempuri.org/Catalog1.xsd"
  xmlns="http://tempuri.org/Catalog1.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  attributeFormDefault="qualified" elementFormDefault="qualified">
  <xsd:element name="Catalog" msdata:IsDataSet="true"
    msdata:EnforceConstraints="False">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="Product">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="ProductID"
                type="xsd:string" minOccurs="0" />
              <xsd:element name="ProductName"
                type="xsd:string" minOccurs="0" />
              <xsd:element name="ListPrice"
                type="xsd:string" minOccurs="0" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Continued

**Figure 3.7 Continued**

```
        </xsd:element>
    </xsd:choice>
</xsd:complexType>
</xsd:element>
</xsd:schema>
```

Minimal knowledge about the XSD schema is required to understand the XML.NET architecture. You will find it especially useful when we discuss the *XmlDataDocument*.

## Developing & Deploying...

### XML Validation in VS.NET

VS.NET provides a number of tools to work on XML documents. One of them enables you to check if a given XML document is well formed. While on the XML view of an XML document, you may use **XML | Validate XML Data** of the main menu to see if the document is well formed. The system displays its findings in the bottom-left corner of the status bar. Similarly, you can use the Schema Validation tool to check if your schema is well formed, too. While on the XML view of the schema, use the **Schema | Validate Schema** of the main menu to perform this task.

However, none of the above tests guarantee that your XML data is valid according to the rules specified in the schema. To accomplish this task, you will need to link your XML document to a particular schema first. Then you can test the validity of the XML document. To assign a schema to an XML document, perform the following steps:

1. Display the XML document in XML view (in the XML Designer).
2. Display its **Property sheet**. (It will be captioned **DOCUMENT**.)
3. Open the drop-down list box at the right-hand side of the **targetSchema**, and select the appropriate schema.
4. Now, go ahead and validate the document using the **XML | Validate XML Data** of the main menu.

Continued

By the way, there are many other third-party software packages that can also test if an XML document is well formed, and if it is valid (against a given schema). In this context, we have found the XML Authority (by TIBCO) and XML Writer (by Wattle Software) to be very good. An excellent tool named XSV is also available from [www.w3.org/2000/09/webdata/xsv](http://www.w3.org/2000/09/webdata/xsv).

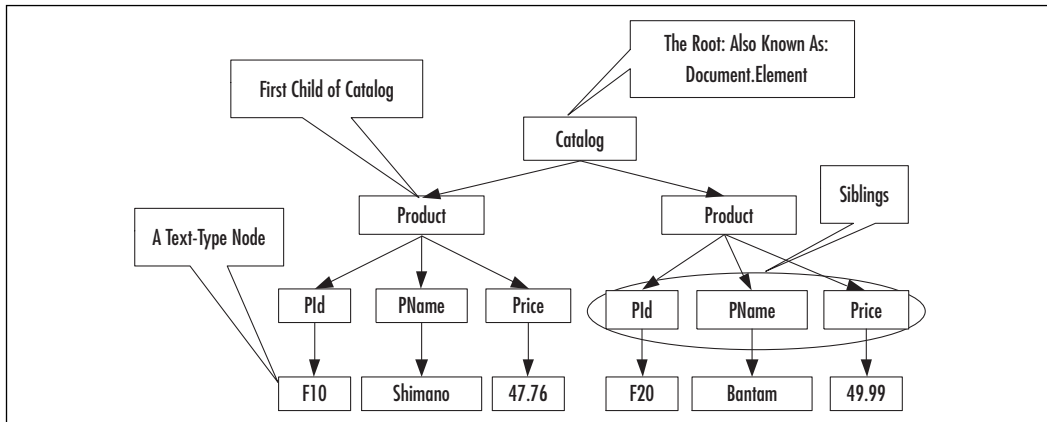
## NOTE

Readers interested in the details of DTD and Schema may explore <http://msdn.microsoft.com/xml/default.asp> and [www.w3.org/XML](http://www.w3.org/XML).

## Structure of an XML Document

In an XML document, the data are stored in a hierarchical fashion. A hierarchy is also referred to as a *tree* in data structures. Conceptually, the data stored in the `Catalog1.xml` can be represented as a tree diagram, as shown in Figure 3.8. Please note that certain element names and values have been abbreviated in the tree diagram, mostly to conserve real estate on the page.

**Figure 3.8** The Tree-Diagram for `Catalog1.xml`



In this figure, each rectangle is a node in the tree. Depending on the context, a node can be of different types. For example, each product node in the figure is an *element-type* node. Each product node happens to be a *child node* of the catalog

node. The catalog node can also be termed as the *parent* of all product nodes. Each product node, in turn, is the parent of its *PIId*, *PName*, and *Price* nodes.

In this particular tree diagram, the bottom-most nodes are *not* of element-type; rather, these are of *text-type*. There could have been nodes for each attribute and its value, too, although we have not shown those in this diagram.

The *Product* nodes are the immediate *descendants* of the *Catalog* node. Both *Product* nodes are *siblings* of each other. Similarly, the *PIId*, *PName*, and *Price* nodes under a specific product node are also siblings of each other. In short, all children of a parent are called siblings.

At this stage, you may have been wondering why we are studying the family history rather than ASP. Well, you will find out pretty soon that all of these terminologies will play major roles in taming the beauties and the beasts of something called XML technology.

## Processing XML Documents Using .NET

The entire ADO.NET Framework has been designed based on XML technology. Many of the ADO.NET data-handling methodologies, including *DataTables* and *DataSets*, use XML in the background, thus keeping it transparent to us. The .NET Framework's *System.Xml* namespace provides a very rich collection of classes that can be used to store and process XML documents. These classes are also often referred to as the XML.NET.

Before we get into the details of the XML.NET objects, let us ask ourselves several questions. As ASP NET developers, what kind of support would we need from .NET for processing XML documents? Well, at the very least, we would like .NET to assist us in creating, reading, and parsing XML documents. Anything else? Okay, if we have adequate cache, we would like to load the entire document in the memory and process it directly from there. If we do not have enough cache, then we would like to read various fragments of an XML document one piece at a time. Do we want more? How about the ability for searching and querying the information contained in an XML document? How about instantly creating an XML document from a database query and sending it to our B2B partners? How about converting an XML document from one format to another format and transmitting it to other servers? Actually, XML.NET provides all of these, and much more!

All of the above questions fall into two major categories:

1. How do we read, parse and write XML documents?
2. How do we store, structure, and process them in the memory?

As mentioned earlier, XML is associated with a growing family of technologies and frameworks. The major trends in this area are W3C DOM, XSLT, XPath, XPath Query, SAX, and XSLT. In XML.NET, Microsoft has incorporated almost all of these frameworks and technologies. It has also added some of its own unique ideas. There is a plethora of alternative XML.NET objects to satisfy our needs and likings. However, it's a jungle out there! In the remainder of this section, we will have a brief glance over this jungle.

## Damage & Defense...

### Legacy Systems and XML

Organizational data stored in legacy systems can be converted to appropriate XML documents, if needed, reasonably easily. There is third-party software like XML Authority by Tibco Extensibility and others, which can convert legacy system's data into XML format. We can also use VS.NET to convert legacy data to XML documents.

## Reading and Writing XML Documents

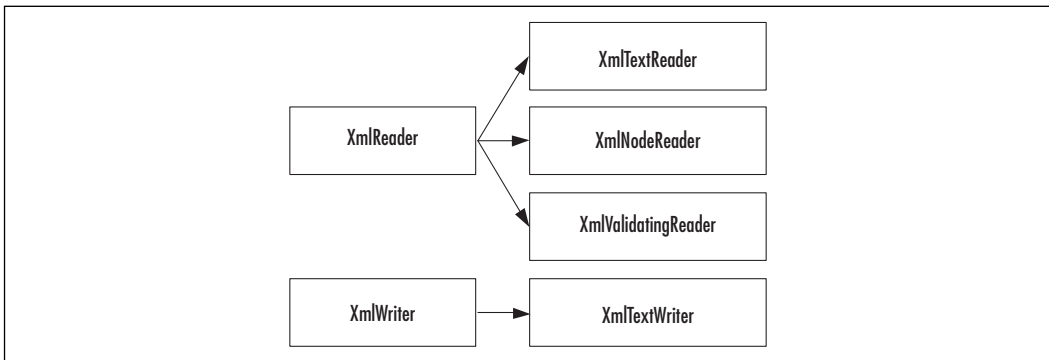
Two primary classes in this group are *XmlReader* and *XmlWriter*. Both of these classes are abstract classes, and therefore we cannot create objects of these classes. Microsoft has provided a number of concrete implementations of both of these classes:

- **XmlTextReader** We may use an object of this class to read non-cached XML data on a forward-only basis. It checks for well-formed XML, but it does not support data validation.
- **XmlNodeReader** An object of this class can be used to access non-cached forward-only data from an XML node. It does not support data validation.
- **XmlValidationReader** This is very similar to the *XMLTextReader*, except that it accommodates XML data validation.

We may create objects of these classes and use their methods and properties. If warranted, we may also extend these classes to provide further specific

functionalities. Fortunately, the *XmlWriter* class has only one concrete implementation: *XmlTextWriter*. It can be used to write XML document on a forward-only basis. These classes and their relationships are shown in Figure 3.9.

**Figure 3.9** Major *XmlReader* and *XmlWriter* Classes



## Storing and Processing XML Documents

Once XML data are read, we need to structure these data in the computer's memory. For this purpose, the major offerings include the *XmlNode* class and the *XPathDocument* class. The *XmlNode* class is an abstract class. There are a number of concrete implementations of this class, too, such as the *XmlDocument*, *XmlAttribute*, *XmlDocumentFragment*, and so on. We will limit our attention to the *XmlDocument* class, and to one of its subsequent extensions named the *XmlDataDocument*. The characteristics of some of these classes are as follows:

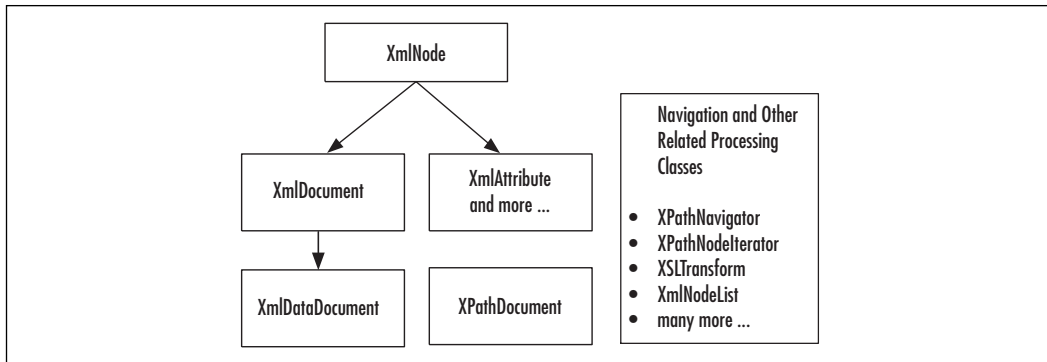
- **XmlDocument** This class structures an XML document according to a DOM tree (as specified in the W3C DOM Core Level 1 and 2 specifications).
- **XmlDataDocument** This class is a major milestone in integrating XML and database processing. It allows two views of the in-cache data: the Relational Table view, and the XML Tree View.
- **XPathDocument** This class employs the XSLT and XPath technologies, and enables you to transform an XML document in to a desired format.

Above classes are essentially used for storing the XML data in the cache. Just storing data in the memory serves us no purpose unless we can process and query these data. The .NET Framework has included a number of classes to



operate on the cached XML data. These classes include *XPathNavigator*, *XPathNodeIterator*, *XSLTransform*, *XmlNodeList*, etc. These classes are shown in Figure 3.10.

**Figure 3.10** Major XML Classes for In-Memory Storage and Processing



## Reading and Parsing Using the *XmlTextReader* Class

The *XmlTextReader* class provides a fast forward-only cursor that can be used to “pull” data from an XML document. An instance of it can be created as follows:

```
Dim myRdr As New XmlTextReader(Server.MapPath("catalog2.xml"))
```

Once an instance is created, the imaginary cursor is set at the top of the document. We may use its *Read()* method to extract fragments of data sequentially. Each fragment of data is distantly similar to a node of the underlying XML tree. The *NodeType* property captures the type of the data fragment read, the *Name* property contains the name of the node, and the *Value* property contains the value of the node, if any. Thus, once a data fragment has been read, we may use the following type of statement to display the node-type, name, and value of the node.

```
Response.Write(myRdr.NodeType.ToString() + " " +  
myRdr.Name + ": " + myRdr.Value)
```

The attributes are treated slightly differently in the *XmlTextReader* object. When a node is read, we may use the *HasAttributes* property of the reader object to see if there are any attributes attached to it. If there are attributes in an element, the *MoveToAttribute(i)* method can be applied to iterate through the attribute collection. The *AttributeCount* property contains the number of attributes of the current element. Once we process all of the attributes, we need to apply the *MoveToElement*

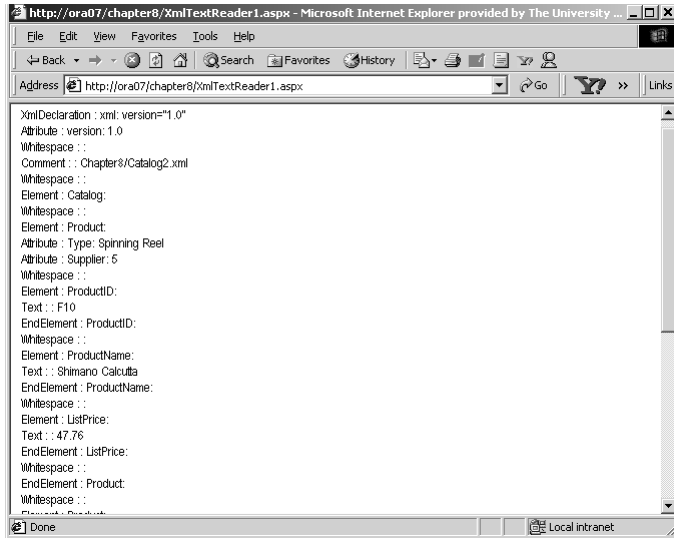
method to move the cursor back to the current element node. Therefore, the following code will display the attributes of an element:

```
If myRdr.HasAttributes Then
    For i = 0 To myRdr.AttributeCount - 1
        myRdr.MoveToAttribute(i)
        Response.Write(myRdr.NodeType.ToString() + " : "+ myRdr.Name _
            + ": " + myRdr.Value + "<br>")
    Next i
    myRdr.MoveToElement()
End If
```

Microsoft has loaded the *XmlDocument* class with a variety of convenient class members. Some of the frequently used methods and properties are *AttributeCount*, *Depth*, *EOF*, *HasAttributes*, *HasValue*, *IsDefault*, *IsEmptyElement*, *Item*, *ReadState*, and *Value*.

## Parsing an XML Document

In this section, we will apply the *XmlTextReader* object to parse and display all data contained in our *Catalog2.xml* (as shown in Figure 3.5) document. The code for this example and its output are shown in Figure 3.11 and Figure 3.12, respectively. The code shown in Figure 3.12 is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). Our objective is to start at the top of the document and then sequentially travel through its nodes using the *XMLTextReader's Read()* method. When there is no more data to read, the *Read()* method returns “false.” Thus, we are able to build the *While myRdr.Read()* loop to process all data. Please review the code (Figure 3.12) and its output cautiously. While displaying the data, we have separated the node-type, node-name, and values using colons. Not all elements have names or values. Hence, you will see many empty names and values after respective colons.

Figure 3.11 Truncated Output of the *XmlTextReader1.aspx* CodeFigure 3.12 *XmlTextReader1.aspx*

```

<!-- Chapter8\xmlTextReader1.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
<Script runat="server">
Sub Page_Load(sender As Object, e As EventArgs)
    Dim myRdr As New XmlTextReader(Server.MapPath("Catalog2.xml"))
    Dim i As Integer
    While myRdr.Read()
        Response.Write(myRdr.NodeType.ToString() + " : " + myRdr.Name _
            + ": " + myRdr.Value + "<br/>")
        If myRdr.HasAttributes Then
            For i = 0 To myRdr.AttributeCount - 1
                myRdr.MoveToAttribute(i)
                Response.Write(myRdr.NodeType.ToString() + " : " + myRdr.Name _
                    + ": " + myRdr.Value + "</br>")
            Next i
            myRdr.MoveToElement()
        End If
    End While

```

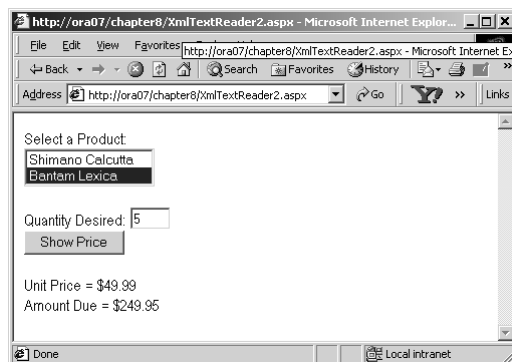
Continued

**Figure 3.12** Continued

```
myRdr.Close()  
End Sub  
</Script>
```

## Navigating through an XML Document to Retrieve Data

In the previous section, we extracted and displayed all data, including the “white-spaces” contained in an XML document. Now, we will illustrate an example where we will navigate through the document and pick up only those data that are necessary for an application. The output of this application is shown in Figure 3.13. In this example, we will display the names of our products in a list box. We will load the list box using the *Product Name* data from the XML file. The user will select a particular product. Subsequently, we will search the XML document to find and display the price of the product. We will travel through the XML file twice, once to load the list box, and once to find the price of a selected product. Please be aware that we could have easily developed the application by building an array or arraylist of the products during the first pass through the XML data, thus avoiding a second pass. Nevertheless, we are reading the file twice just to illustrate various methods and properties of the *XmlTextReader* object.

**Figure 3.13** Output of the Navigation ASPX Example *XmlTextReader2.aspx*

To load the List Box, we will go through the following process: We will load the list box in the *Page\_Load* event. Here, we will read the nodes one at a time. If the node type is of element-type, we will check if its name is *ProductName*. If it is

a *ProductName* node, we will perform a *Read()* to get to its text node and then apply the *myRdr.ReadString()* method to extract the value and load it in the list box. Finally, we will close the reader object. **Caution:** We are assuming that there is no “whitespace” between the *ProductName* and its Text node. If there is a “whitespace,” we will need to put the second *Read()* in a loop until the node-type is Text.

```
While myRdr.Read()
    If XmlNodeType.Element
        If myRdr.Name="ProductName" Then
            myRdr.Read()
            lstProducts.Items.Add(myRdr.ReadString)
        End If
    End If
End While
myRdr.Close()
```

To find the price of the selected product, we will go through the following process: We will include the necessary code in the “onclick” event code of the command button “Show Price.” We will create a second *XmlTextReader* object based on the *Catalog2.xml* file. Of course, we may scan all nodes sequentially to find the price. However, the *XmlTextReader* class enables you to skip undesirable nodes, such as the “whitespace” or the declaration nodes via the *MoveToContent()* method. According to Microsoft, all nonwhitespace, Element, End Element, EntityReference, and EndEntity nodes are *content nodes*. The *MoveToContent()* method checks whether the current node is a content node. If the node is not a content node, then the method skips to the next content node. You need to be careful though. If the current node happens to be a content node, the cursor does not move to the next content node automatically on a further *MoveToContent()*.

Initially, when we instantiate the *reader* object, its node type is *None*. It happens to be a noncontent node. Hence our first *MoveToContent()* statement takes us to a content node. There, we check if it is an Element-type node named “ProductName” and if its *ReadString()* is equal to the name of the selected product. If all are true, then we apply a *Read()* to go to the next node. This *Read()* may take us to a “whitespace” node, and thus we have applied a *MoveToContent()* to get to the ListPrice node. Figure 3.14 shows an excerpt of the relevant code. The complete code is available in *XmlTextReader2.aspx* file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).


**Figure 3.14** Excerpt of XmlTextReader2.aspx

```

Sub showPrice(s As Object, e As EventArgs)
    Dim myRdr2 As New XmlTextReader(Server.MapPath("Catalog2.xml"))
    Dim unitPrice As Double
    Dim qty AS Integer
    Do While Not myRdr2.EOF()
        If (myRdr2.MoveToContent() = XmlNodeType.Element _
            And myRdr2.Name ="ProductName" _
            And myRdr2.ReadString()=lstProducts.SelectedItem.ToString())
            myRdr2.Read()
            If (myRdr2.MoveToContent() = XmlNodeType.Element _
                And myRdr2.Name ="ListPrice")
                unitPrice=Double.Parse(myRdr2.ReadString())
                lblPrice.Text= "Unit Price = " + FormatCurrency(unitPrice)
            Exit Do
        End If
    End If
    myRdr2.Read()
Loop
qty = Integer.Parse(txtQty.Text)
lblAmount.Text = "Amount Due = " + FormatCurrency(qty * unitPrice)
myRdr2.Close()
End Sub

```

By the way, we could have also used the *MoveToContent()* method to load our list box more effectively. However, we just wanted to show the alternative methodologies.

## NOTE

We may also read XML files from remote servers as follows:

```
Dim myRdr As New XmlTextReader("http://ahmed2/Chapter8/Catalog2.xml")
```

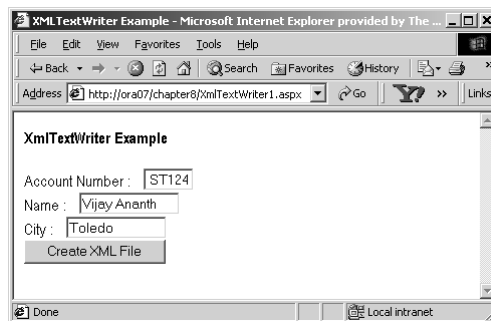
## Writing an XML Document Using the *XmlTextWriter* Class

The *XmlTextWriter* class is a concrete implementation of the *XmlWriter* abstract class. An *XmlTextWriter* object can be used to write data sequentially to an output stream, or to a disk file as an XML document. The data to be written may come from the user's input and/or from a variety of other sources, such as text files, databases, *XmlTextReaders*, or *XmlDocuments*. Its major methods and properties include *Close*, *Flush*, *Formatting*, *WriteAttribues*, *WriteAttributeString*, *WriteComment*, *WriteElementString*, *WriteEndElement*, *WriteEndAttribute*, *WriteEndDocument*, *WriteState*, and *WriteStartDocument*.

## Generating an XML Document Using *XmlTextWriter*

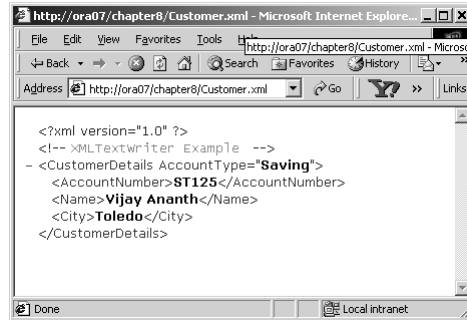
In this section, we will collect user-given data via an .aspx page, and write the information in an XML file. The run-time view of the application is shown in Figure 3.15. On the *click* event of the “Create XML File,” the application will create the XML file (in the disk) and display it back in the browser as seen in Figure 3.16.

**Figure 3.15** Output of the *XmlTextReader2.aspx*



We have included the necessary code in the *click* event of the command button. Our objective is to write the data in a disk file named *Customer.xml*. In the code, first we have created an instance of the *XmlTextWriter* object as follows:

```
Dim myWriter As New XmlTextWriter _
    (Server.MapPath("Customer.xml"), Nothing)
```

**Figure 3.16** Generated XML File

The second parameter “Nothing” is specified to map the file to a UTF-8 format. Then it is just a matter of writing the various elements, attributes, and their values judiciously. Once the file is written, we simply employed the *Response.Redirect(Server.MapPath(“Customer.xml”))* to display the XML documents information in the browser. The complete code for the application is shown in Figure 3.17. Both Customer.xml and XmlTextWriter1.aspx files are available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.17** XmlTextWriter1.aspx

```
<!-- Chapter8\XmlTextWriter1.aspx -->
<%@ Page Language="VB" Debug="True"%>
<%@ Import Namespace="System.Xml"%>
<HTML><HEAD><title>XMLTextWriter Example</title></HEAD>
<body><form runat="server">
<b>XmlTextWriter Example</b><br/><br/>
<asp:Label id="lblAcno" Text="Account Number :" runat="server"/>&nbsp;&nbsp;&nbsp;
<asp:TextBox id="txtAcno" runat="server" width="50" _
text=" ST124" /><br/>
<asp:Label id="lblName" Text="Name :" runat="server" />&nbsp;&nbsp;&nbsp;
<asp:TextBox id="txtName" runat="server" width="100" text="Vijay
Ananth"/><br/>
<asp:Label id="lblCity" Text="City :" runat="server"/>&nbsp;&nbsp;&nbsp;
<asp:TextBox id="txtCity" runat="server" width="100" text="Toledo"/><br/>
<asp:Button id="cmdWriteXML" Text="Create XML File" runat="server"
onclick="writeXML"/>
<br/></form>
```

Continued



**Figure 3.17 Continued**


---

```

<Script Language="vb" runat="server">

Sub writeXML(sender As Object,e As EventArgs)
    Dim myWriter As New XmlTextWriter _
        (Server.MapPath("Customer.xml"), Nothing)
    myWriter.Formatting = Formatting.Indented
    myWriter.WriteStartDocument()      'Start a new document
    ' Write the Comment
    myWriter.WriteComment("XMLTextWriter Example")
    ' Insert an Start element tag
    myWriter.WriteStartElement("CustomerDetails")
    ' Write an attribute
    myWriter.WriteAttributeString("AccountType", "Saving")
    ' Write the Account element and its content
    myWriter.WriteStartElement("AccountNumber","")
    myWriter.WriteString(txtAcno.Text)
    myWriter.WriteEndElement()
    ' Write the Name Element and its data
    myWriter.WriteStartElement("Name","")
    myWriter.WriteString(txtName.Text)
    myWriter.WriteEndElement()
    'Write the City element and its data
    myWriter.WriteStartElement("City","")
    myWriter.WriteString(txtCity.Text)
    myWriter.WriteEndElement()

    'End all the tags here
    myWriter.WriteEndDocument()

    myWriter.Flush()
    myWriter.Close()

    'Display the XML content on the screen
    Response.Redirect(Server.MapPath("Customer.xml"))

```

---

Continued

Figure 3.17 Continued

---

 End Sub

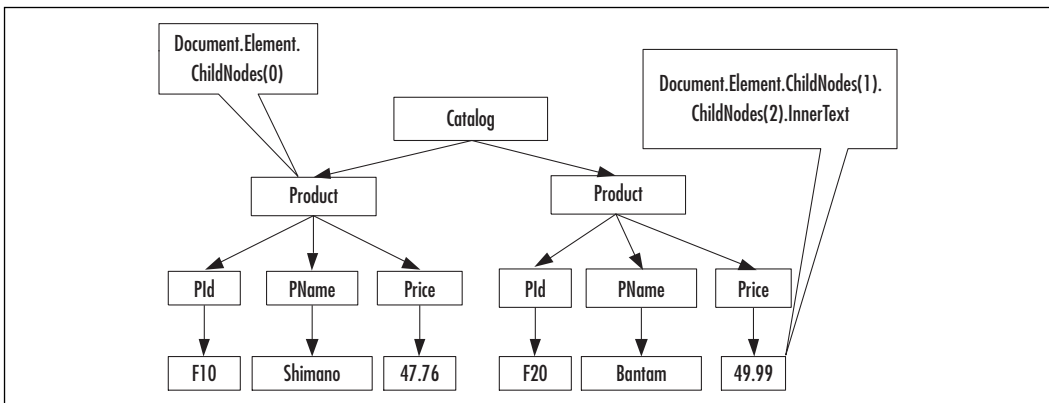
 </Script>

---

## Exploring the XML Document Object Model

The W3C Document Object Model (DOM) is a set of specifications to represent an XML document in the computer's memory. Microsoft has implemented the W3C Document Object Model via a number of .NET objects. The *XmlDocument* is one of these objects. When an *XmlDocument* object is loaded, it organizes the contents of an XML document as a “tree” (as shown in Figure 3.18). Whereas the *XMLTextReader* object provides a forward-only cursor, the *XmlDocument* object provides fast and direct access to a node. However, a DOM tree is cache intensive, especially for large XML documents.

Figure 3.18 Node Addressing Techniques in an XML DOM Tree



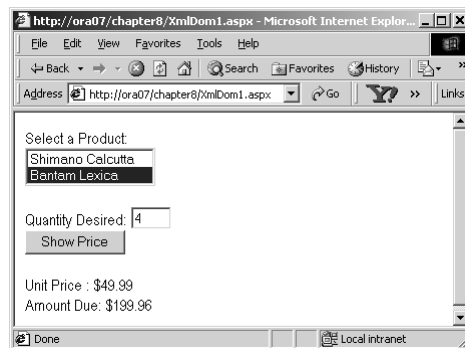
An *XmlDocument* object can be loaded from an *XmlTextReader*. Once it is loaded, we may navigate via the nodes of its tree using numerous methods and properties. Some of the frequently used members are the following: *DocumentElement* (root of the tree), *ChildNodes* (all children of a node), *FirstChild*, *LastChild*, *HasChildNodes*, *ChildNodes.Count* (number of children), *InnerText* (the content of the sub-tree in text format), *Name* (node name), *NodeType*, and *Value* (of a text node) among many others.

If needed, we may address a node using the parent-child hierarchy. The first child of a node is the *ChildNode(0)*, the second child is *ChildNode(1)*, and so on. For example, the first product can be referenced as *DocumentElement.ChildNodes(0)*. Similarly, the price of the second product can be addressed as *DocumentElement.ChildNodes(1).ChildNodes(2).InnerText*.

## Navigating through an *XmlDocument* Object

In this example we will implement our product selection page using the XML document object model. The output of the code is shown in Figure 3.19.

**Figure 3.19** Output of the *XmlDocument* Object Example



Let's go through the process of loading the *XmlDocument* (DOM tree). There are a number different ways to load an *XML Document* object. We will load it using an *XmlTextReader* object. We will ask the reader to ignore the “whitespaces” (more or less to conserve cache). As you can see from the following code, we are loading the tree in the *Page\_Load* event. On “PostBack”, we will not have access to this tree. That is why we are storing the “tree” in a *Session* variable. When the user makes a selection, we will retrieve the tree from the session, and search its node for the appropriate price.

```
Private Sub Page_Load(s As Object, e As EventArgs)
    If Not Page.IsPostBack Then
        Dim myDoc As New XmlDocument()
        Dim myRdr As New XmlTextReader(Server.MapPath("Catalog2.xml"))
        myRdr.WhitespaceHandling = WhitespaceHandling.None
        myDoc.Load(myRdr)
        Session("sessionDoc") = myDoc ' Put it in a session variable
    End If
End Sub
```

Once the tree is loaded, we can load the list box with the *InnerText* property of the *ProductName* nodes.

```
For i = 0 To myDoc.DocumentElement.ChildNodes.Count - 1
    lstProducts.Items.Add _
        (myDoc.DocumentElement.ChildNodes(i).ChildNodes(1).InnerText)
Next i

myRdr.Close()
```

Next, let's investigate how to retrieve the price of a selected product. On click of the **Show Price** button, we simply retrieve the tree from the session, and get to the *Price* node directly. The *SelectedIndex* property of the list box does a favor for us, as its Selected Index value will match the corresponding child's ordinal position in the *Catalog (DocumentElement)*. Figure 3.20 shows an excerpt of the relevant code that is used to retrieve the price of a selected product. The complete code is available in the *XmlDom1.aspx* file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 3.20** Partial Listing of *XmlDom1.aspx*

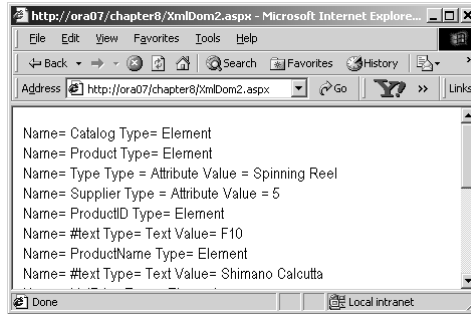
```
Private Sub showPrice(s As Object, e As EventArgs)
    Dim i As Integer
    Dim qty As Integer = 1
    Dim price As Double
    Dim myDoc As New XmlDocument()
    myDoc = Session("sessionDoc")
    i = lstProducts.SelectedIndex ' The Row number selected
    qty = Integer.Parse(txtQty.Text)
    price = Double.Parse _
        (myDoc.DocumentElement.ChildNodes(i).ChildNodes(2).InnerText)
    lblPrice.Text = FormatCurrency(price)
    lblAmount.Text = FormatCurrency(qty * price)
End Sub
```

## Parsing an XML Document Using the *XmlDocument* Object

A *tree* is composed of nodes. Essentially, a node is also a tree because it contains all other nodes below it. A node at the bottom does not have any children; hence,

most likely it will be of a text-type node. We will employ this phenomenon to travel through a tree using a VB recursive procedure. The primary objective of this example is to travel through DOM tree and display the information contained in each of its nodes. The output of this exercise is shown in Figure 3.21.

**Figure 3.21** Parsing an *XmlDocument* Object



We will develop two sub-procedures:

1. **DisplayNode(node As XmlNode)** It will receive a node and check if it is a terminal node. If the node is a terminal node, this sub-procedure will print its contents. If the node is not a terminal node, then the sub-procedure will check if the node has any attributes. If there are attributes, it will print them.
2. **TravelDownATree(tree As XmlNode)** It will receive a tree, and at first it will call the DisplayNode procedure. Then it will pass the sub-tree of the received tree to itself. This is a recursive procedure. Thus, it will actually fathom all nodes of a received tree, and we will get all nodes of the entire tree printed.

The complete listing of the code is shown in Figure 3.22. The code is also available in the file named *XmlDom2.aspx* on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). As usual, we will load the *XmlDocument* in the *Page\_Load()* event using an *XmlTextReader*. After the DOM tree is loaded, we will call the *TravelDownATree* recursive procedure, which will accomplish the remainder of the job.

**Figure 3.22** The Complete Code *XmlDom2.aspx*

```
<!-- Chapter8\xmlDom2.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
```

Continued

## Figure 3.22 Continued

---

```

<Script Language="vb" runat="server">
Sub Page_Load(s As Object, e As EventArgs)
    If Not Page.IsPostBack Then
        Dim myXmlDoc As New XmlDocument()
        Dim myRdr As New XmlTextReader(Server.MapPath("Catalog2.xml"))
        myRdr.WhitespaceHandling = WhitespaceHandling.None
        myXmlDoc.Load (myRdr)
        TravelDownATree(myXmlDoc.DocumentElement)
        myRdr.Close()
    End If
End Sub
Sub TravelDownATree(tree As XmlNode)
    If Not IsNothing(tree) Then
        DisplayNode(tree)
    End If
    If tree.HasChildNodes Then
        tree = tree.FirstChild
        While Not IsNothing(tree)
            TravelDownATree(tree) //Call itself and pass the subtree
            tree = tree.NextSibling
        End While
    End If
End Sub
Sub DisplayNode(node As XmlNode)
    If Not node.HasChildNodes Then
        Response.Write( "Name= " + node.Name + " Type= " _
            + node.NodeType.ToString()+" Value= "+node.Value + "<br/>")
    Else
        Response.Write("Name= " + node.Name + " Type= " _
            + node.NodeType.ToString() + "<br/>")
        If node.NodeType = XmlNodeType.Element Then
            Dim x As XmlAttribute
            For each x In node.Attributes
                Response.Write("Name= " + x.Name + " Type = " _
                    + x.NodeType.ToString()+" Value = "+x.Value + "<br/>")
            End For
        End If
    End If
End Sub

```

---

Continued

**Figure 3.22 Continued**


---

```

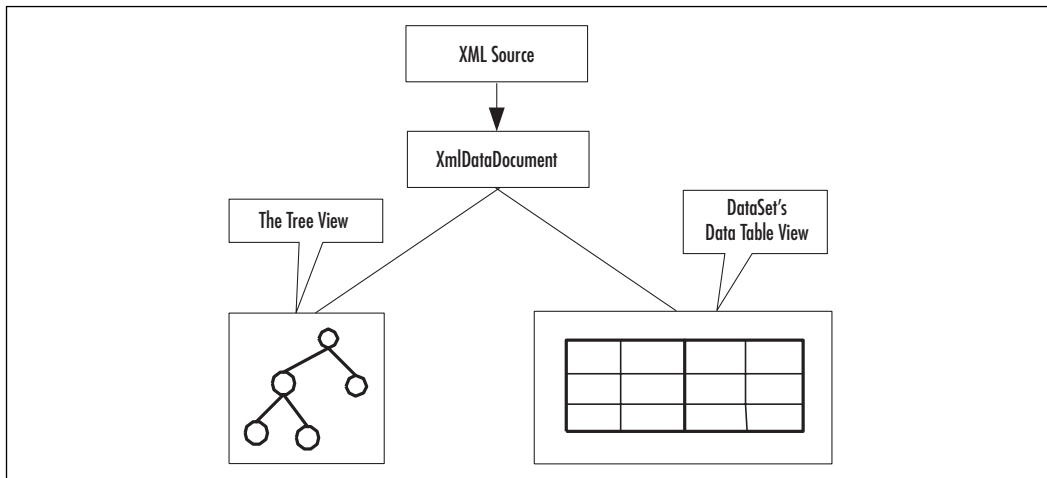
    Next
  End If
End If
End Sub
</Script>

```

---

## Using the *XmlDataDocument* Class

The *XmlDataDocument* class is an extension of the *XmlDocument* class. It more-or-less behaves almost the same way the *XmlDocument* does. The most fascinating feature of an *XmlDataDocument* object is that it provides two alternative views of the same data, the “XML view” and the “relational view.” The *XmlDataDocument* has a property named *DataSet*. It is through this property that *XmlDataDocument* exposes its data as one or more related or unrelated *DataTables*. A *DataTable* is actually an imaginary table-view of XML data. Once we load an *XmlDataDocument* object, we can treat it as a DOM tree, or we can treat its data as a *DataTable* (or a collection of *DataTables*) via its *DataSet* property. Figure 3.23 shows the two views of an *XmlDataDocument*. Because these views are drawn from the same *DataDocument* object, these are automatically synchronized. That means that any changes in any one of them will change the other.

**Figure 3.23** Two Views of an *XmlDataDocument* Object

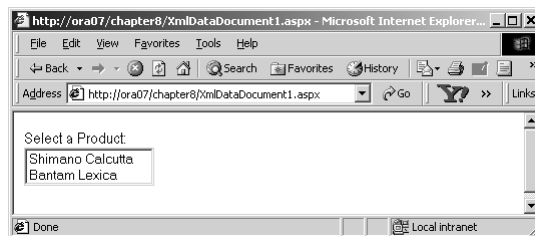
In this section, we will provide three examples.

- We will demonstrate how to load an XML document as an *XmlDataDocument* object, and process it as a Dom tree.
- We will illustrate how to retrieve the data from a *DataTable* view of the *XmlDataDocument*'s *DataSet*.
- Finally, We will demonstrate when and how the *XmlDataDocument* object provides multiple-table views.

## Loading an *XmlDocument* and Retrieving the Values of Certain Nodes

In this section we will load an *XmlDataDocument* using our *Catalog2.xml* file. After we load it, we will retrieve the product names and load them in a list box. The output of this example is shown in Figure 3.24. The code for this application is listed in Figure 3.25, and it is also available in the file named *XmlDataDocument1.aspx* on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.24** Output of *XmlDataDocument1.aspx*



The *XmlDataDocument* is a pleasant object to work with. In this example, the code is pretty straightforward. After we have loaded the *XmlDataDocument*, we have declared an *XmlNodeList* collection named *productNames*. We have populated the collection by using the *GetElementsByTagName*("ProductName") method of the *XmlDataDocument* object. Finally, it is just a matter of iterating through the *productNames* collection and loading each of its members in the list box.

At this stage, you will probably ask why we are not finding the unit price of the selected product. Actually, therein lies the beauty of the *XmlDataDocument*. Because it has extended the *XmlDocument* class, all of the members of the *XmlDocument* class are also available to us. Thus, we could use the same technique as shown in our previous example to find the price. Nevertheless, the reason for



not showing the searching technique here is that we will cover it later when we discuss the *XPathIterator* object.

**Figure 3.25** XmlDataDocument1.aspx

```

<!--\Chapter8\xmlDataDocument1.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
<html><head></head><body><form runat="server">
Select a Product: <br/>
<asp:ListBox id="lstProducts" runat="server" rows = "2" /><br/><br/>
</body></form><html>
<Script Language="vb" runat="server">
Sub Page_Load(s As Object, e As EventArgs)
    If Not Page.IsPostBack Then
        Dim myDataDoc As New XmlDataDocument()
        myDataDoc.Load(Server.MapPath("Catalog2.xml"))
        Dim productNames As XmlNodeList
        productNames= myDataDoc.GetElementsByTagName("ProductName")
        Dim x As XmlNode
        For Each x In productNames
            lstProducts.Items.Add (x.FirstChild().Value)
        Next
    End If
End Sub
</Script>

```

## Using the Relational View of an *XmlDataDocument* Object

In this example, we will process and display the *Catalog3.xml* document's data as a relational table in a *DataGrid*. The *Catalog3.xml* is exactly the same as *Catalog2.xml* except that it has more data. The *Catalog3.xml* file is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). The output of this example is shown in Figure 3.26.

Figure 3.26 Output of *XmlDataDocument DataSet* View Example

ProductID	ProductName	ListPrice	Type	SupplierId
F10	Shumano Calcutta	47.76	Spinning Reel	5
F20	Bantam Lexica	49.99	Baitcasting Reel	3
F30	Quantum Micro	57.11	Spinning Reel	5
F40	Minnow By Daiwa	39.99	Baitcasting Reel	2

If we want to process the XML data as relational data, we need to load the schema of the XML document first. We have generated the following schema for the *Catalog3.xml* using VS.NET. The schema specification is shown in Figure 3.27 (also available on the Solutions Web site for the book).

Figure 3.27 *Catalog3.xsd*

```
<xsd:schema id="Catalog" targetNamespace="http://tempuri.org
  /Catalog3.xsd" xmlns="http://tempuri.org/Catalog3.xsd"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:msdata
  ="urn:schemas-microsoft-com:xml-msdata" attributeFormDefault
  ="qualified" elementFormDefault="qualified">
  <xsd:element name="Catalog" msdata:IsDataSet="true"
    msdata:EnforceConstraints="False">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="Product">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="ProductID" type="xsd:string" minOccurs="0"
                msdata:Ordinal="0" />
              <xsd:element name="ProductName" type="xsd:string"
                minOccurs="0" msdata:Ordinal="1" />
              <xsd:element name="ListPrice" type="xsd:string" minOccurs="0"
                msdata:Ordinal="2" />
            </xsd:sequence>
            <xsd:attribute name="Type" form="unqualified" type="xsd:string"/>
            <xsd:attribute name="SupplierId" form="unqualified"
```

Continued

**Figure 3.27 Continued**


---

```

        type="xsd:string" />
    </xsd:complexType>
</xsd:element>
</xsd:choice>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```

---

**NOTE**

When we create a schema from a sample XML document, VS.NET automatically inserts an *xmlns* attribute to the root element. The value of this attribute specifies the name of the schema. Thus when we created the schema for *Catalog3.xml*, the schema was named *Catalog3.xsd* and VS.NET inserted the following attributes in the root element of *Catalog3.xml*:

```
<Catalog xmlns="http://tempuri.org/Catalog3.xsd">
```

---

In our .aspx code, we loaded the schema using the *ReadXmlSchema* method of our *XmlDataDocument* object as:

```
myDataDoc.DataSet.ReadXmlSchema(Server.MapPath("Catalog3.xsd")).
```

Next, we have loaded the *XmlDataDocument* as:

```
myDataDoc.Load(Server.MapPath("Catalog3.xml")).
```

Since the *DataDocument* provides two views, we have exploited its *DataSet.Table(0)* property to load the *DataGrid* and display our XML file's information in the grid. The complete listing of the code is shown in Figure 3.28. The code is also available in the *XmlDataDocDataSet1.aspx* file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.28 Complete Listing XmlDataDocDataSet1.aspx**


---

```

<!-- Chapter8\XmlDataDocDataSet1.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data" %>

```

---

Continued

## Figure 3.28 Continued

---

```

<html><head></head><body><form runat="server">
Select a Product: <br/>
<asp:DataGrid id="myGrid" runat="server"/>
</body></form></html>
<Script Language="vb" runat="server">
Sub Page_Load(s As Object, e As EventArgs)
    If Not Page.IsPostBack Then
        Dim myDataDoc As New XmlDataDocument()
        ' load the schema
        myDataDoc.DataSet.ReadXmlSchema(Server.MapPath("Catalog3.xsd"))
        ' load the xml data
        myDataDoc.Load(Server.MapPath("Catalog3.xml"))
        myGrid.DataSource = myDataDoc.DataSet.Tables(0)
        myGrid.DataBind()
    End If
End Sub
</Script>

```

---

## Viewing Multiple Tables of a *XmlDataDocument* Object

In many instances, an XML document may contain nested elements. Suppose that a bank has many customers, and a customer has many accounts. We have modeled this simple scenario in an XML document with nested elements. This document, named *Bank1.xml*, is shown in Figure 3.29. It is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



## Figure 3.29 Bank1.xml

---

```

<?xml version="1.0" encoding="utf-8" ?>
<Bank xmlns="http://tempuri.org/Bank1.xsd">
    <Customer>
        <CustomerID>C100</CustomerID>
        <CustomerName>Alfred Smith</CustomerName>
        <City>Toledo</City>
        <Account>

```

---

Continued

Figure 3.29 Continued

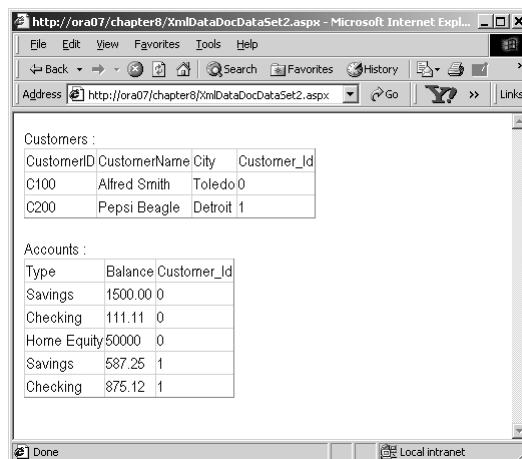
```

        <Type>Savings</Type>
        <Balance>1500.00</Balance>
    </Account>
</Account>
    <Type>Checking</Type>
    <Balance>111.11</Balance>
</Account>
</Account>
    <Type>Home Equity</Type>
    <Balance>50000</Balance>
</Account>
</Customer>
<Customer>
    --- ---
    --- ---
</Customer>
</Bank>

```

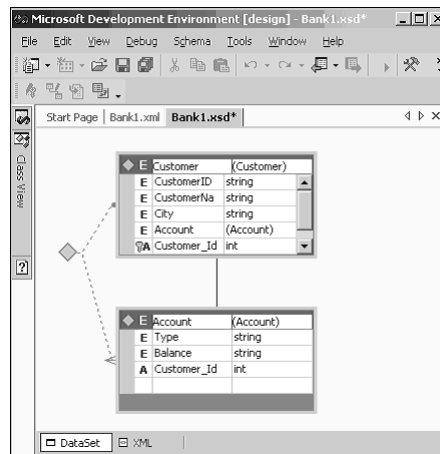
If we load the above XML document and its schema in an *XmlDataDocument* object, it will provide two relational tables' views: one for the customer's information, and the other for the account's information. Our objective is to display the data of these relational tables in two *DataGrids* as shown in Figure 3.30.

Figure 3.30 Displaying Customer and Accounts Data in Two Data Grids



To develop this application, first we had to generate the schema for our Bank1.xml file. We used the VS.NET XML designer to accomplish this task. It is interesting to observe that while creating the schema, VS.NET automatically generates the 1:Many relationship between the *Customer* and *Accounts* elements. To establish the relationship, it also creates an auto-numbered primary key column (*Customer\_Id*) in the *Customer* DataTable. Simultaneously, it inserts the appropriate values of the foreign keys in the *Account* DataTable. The *DataSet* view of the generated schema is shown in Figure 3.31.

**Figure 3.31** *XmlDataDocument DataSet* Representation in Visual Studio .NET



In order to provide the relational view of our XML document (Bank1.xml), VS.NET included the *Customer\_Id* attributes in both *Customer* and *Account* elements in its generated schema. It also generated the necessary schema entries to describe the implied relationship among the *Customer* and *Account* elements. Figure 3.32 shows an excerpt of the generated schema for our XML file. The complete schema is available in a file named Bank1.xsd on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.32** Primary Key and Foreign Key Specifications in the Bank1.xsd

```
<xsd:unique name="Constraint1" msdata:PrimaryKey="true">
    <xsd:selector xpath="//Customer" />
    <xsd:field xpath="@Customer_Id" /></xsd:unique>

<xsd:keyref name="Customer_Account"
    refer="Constraint1"msdata:IsNested="true">
    <xsd:selector xpath="//Account" />
```

**Figure 3.32 Continued**


---

```

        <xsd:field xpath="@Customer_Id" />
</xsd:keyref>

```

---

In the above fragment of the generated schema, the *xsd:unique* element specifies the *Customer\_Id* attribute as the primary key of the *Customer* element. Subsequently, the *xsd:keyref* element specifies the *Customer\_Id* attribute as the foreign key of the *Account* element. XPath expressions have been used to achieve the afore-mentioned objectives.

The complete listing of the application is shown in Figure 3.33. It is also available in the `xmlDataDocDataSet2.aspx` file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). The code is pretty straightforward. We have loaded two data grids from two *DataTables* of the *DataSet*, associated with the *XmlDataDocument* object.

**Figure 3.33 Complete Code of XmlDataDocDataSet2.aspx**


---

```

<!-- Chapter8\XmlDataDocDataSet2.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data" %>
<html><head></head><body><form runat="server">
Customers : <br/>
<asp:DataGrid id="myCustGrid" runat="server"/><br/>
Accounts : <br/>
<asp:DataGrid id="myAcctGrid" runat="server"/><br/>
</body></form></html>
<Script Language="vb" runat="server">
Sub Page_Load(s As Object, e As EventArgs)
    If Not Page.IsPostBack Then
        Dim myDataDoc As New XmlDataDocument()
        ' load the schema
        myDataDoc.DataSet.ReadXmlSchema(Server.MapPath("Bank1.xsd"))
        ' load the xmldata
        myDataDoc.Load(Server.MapPath("Bank1.xml"))
        myCustGrid.DataSource = myDataDoc.DataSet.Tables("Customer")
        myCustGrid.DataBind()
    End If
End Sub

```

---

Continued

**Figure 3.33 Continued**

```
'load the Account grid
myAcctGrid.DataSource = myDataDoc.DataSet.Tables("Account")
myAcctGrid.DataBind()
End If
End Sub
</Script>
```

## NOTE

In a Windows Form, the *DataGrid* control by default provides automatic drill-down facilities for two related *DataTables*. Unfortunately, it does not work in this fashion in a Web form. Additional programming is needed to simulate the drill-down functionality.

In this example, we have illustrated how an *XmlDataDocument* object maps nested XML elements into multiple *DataTables*. Typically, an element is mapped to a table if it contains other elements. Otherwise, it is mapped to a column. Attributes are mapped to columns. For nested elements, the system creates the relationship automatically.

## Querying XML Data Using XPathDocument and XPathNavigator

The *XmlDocument* and the *XmlDataDocument* have certain limitations. First of all, the entire document needs to be loaded in the cache. Often, the navigation process via the DOM tree itself gets to be clumsy. The navigation via the relational views of the data tables may not be very convenient either. To alleviate these problems, the XML.NET has provided the *XPathDocument* and *XPathNavigator* classes. These classes have been implemented using the W3C XPath 1.0 Recommendation ([www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)).

The *XPathDocument* class enables you to process the XML data without loading the entire DOM tree. An *XPathNavigator* object can be used to operate on the data of an *XPathDocument*. It can also be used to operate on *XmlDocument* and *XmlDataDocument*. It supports navigation techniques for selecting nodes,



iterating over the selected nodes, and working with these nodes in diverse ways for copying, moving, and removal purposes. It uses *XPath* expressions to accomplish these tasks.

The *W3C XPath 1.0* specification outlines the query syntax for retrieving data from an XML document. The motivation of the framework is similar to SQL; however, the syntax is significantly different. At first sight, the *XPath* query syntax may appear very complex. But with a certain amount of practice, you may find it very concise and effective in extracting XML data. The details of the *XPath* specification are beyond the scope of this chapter. However, we will illustrate several frequently used *XPath* query expressions. In our exercises, we will illustrate two alternative ways to construct the expressions. The first alternative follows the recent *XPath 1.0* syntax. The second alternative follows *XSL Patterns*, which is a precursor to *XPath 1.0*. Let us consider the following XML document named *Bank2.xml*. The *Bank2.xml* document is shown in Figure 3.34, and it is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). It contains data about various accounts. We will use this XML document to illustrate our *XPath* queries.

**Figure 3.34** Bank 2.xml

---

```
<!-- Chapter8\Bank2.xml -->
<Bank>
  <Account>
    <AccountNo>A1112</AccountNo>
    <Name>Pepsi Beagle</Name>
    <Balance>1200.89</Balance>
    <State>OH</State>
  </Account>
  --- --- ---
  --- --- ---
  <Account>
    <AccountNo>A7833</AccountNo>
    <Name>Frank Horton</Name>
    <Balance>8964.55</Balance>
    <State>MI</State>
  </Account>
</Bank>
```

---

**Sample Query Expression 1:** Suppose that we want the names of all account holders. The following alternative *XPath* expressions will accomplish the job equally well:

- Alternative 1: **descendant::Name**
- Alternative 2: **Bank/Account/Name**

The first expression can be read as “Give me the descendents of all Name nodes.” The second expression can be read as “Give me the Name nodes of the Account nodes of the Bank node.” Both of these expressions will return the same node set.

**Sample Query Expression 2:** We want the records for all customers from Ohio. We may specify any one of the following expressions:

- Alternative 1: **descendant::Account[child::State='OH']**
- Alternative 2: **Bank/Account[child::State='OH']**

**Sample Query Expression 3:** Any one of the following alternative expressions will return the Account node-sets for all accounts with a balance more than 5000.00:

- Alternative 1: **descendant::Account[child::Balance > 5000]**
- Alternative 2: **Bank/Account[child::Balance > 5000.00]**

**Sample Query Expression 4:** Suppose that we want the Account information for those accounts whose names start with the letter “D.”

- Alternative 1: **descendant::account[starts-with(child::Name, 'D')]**
- Alternative 2: **Bank/Account[starts-with(child::Name, 'D')]**

Which of the alternative expressions would you use? That depends on your personal taste and on the structure of the XML document. The second alternative appears to be easier than the first one. However, in the case of a highly nested document, the first alternative will offer more compact expressions. Regardless of the syntax used, please be aware that each of the above queries will return a set of nodes. In our ASP code, we will have to extract the desired information from these sets using an *XPathNodeIterator*.

Okay, now that we have traveled through the *XPath* waters, we are ready to venture into the usages of the *XPathDocument*. In this context, we will provide two examples. The first example will extract the names of the customers from Ohio and load a list box. The second example will illustrate how to find a specific piece of data from an *XPathDocument*.

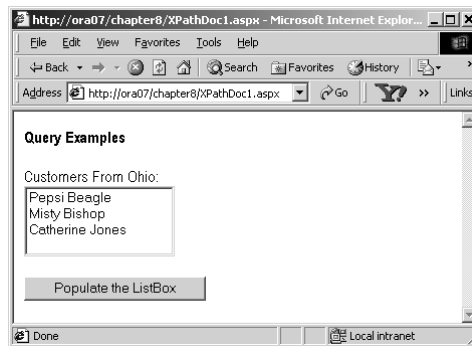
**NOTE**

We found the <http://staff.develop.com/aarons/bits/xpath-builder/> site to be very good in learning *XPath* queries interactively.

## Using *XPathDocument* and *XPathNavigator* Objects

In this section we will use the *XPathDocument* and *XPathNavigator* objects to load a list box from our Bank2.xml file (as shown in Figure 3.34). We will load a list box with the names of customers who are from Ohio. The output of this application is shown in Figure 3.35. The complete code for this application is shown in Figure 3.36. The code is also available in the XPathDoc1.aspx file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.35** Using *XPathDocument* Object



We loaded the Bank2.xml as an *XPathDocument* object as follows:

```
Dim Doc As New XPathDocument(Server.MapPath("Bank2.xml"))
```

At this stage, we need two more objects: an *XPathNavigator* for retrieving the desired node-set, and an *XPathNodeIterator* for iterating through the members of the node-set. These are defined as follows:

```
Dim myNav As XPathNavigator
myNav= myDoc.CreateNavigator()
Dim myIter As XPathNodeIterator
myIter=myNav.Select("Bank/Account[child::State='OH']/Name")
```



## Using *XPathDocument* and *XPathNavigator* Objects for Document Navigation

This section will illustrate how to search an *XPathDocument* using a value of an attribute, and using a value of an element. We will use the `Bank3.xml` to illustrate these. A partial listing of the `Bank3.xml` is shown in Figure 3.37. The complete code is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.37** `Bank3.xml`

---

```
<!-- Chapter8\Bank3.xml -->
<Bank>
  <Account AccountNo="A1112">
    <Name>Pepsi Beagle</Name>
    <Balance>1200.89</Balance>
    <State>OH</State>
  </Account>
  --- --- ---
  --- --- ---
</Bank>
```

---

The *Account* element of the above XML document contains an attribute named *AccountNo*, and three other elements. In this example, we will first load two combo boxes, one with the account numbers, and the other with the account holder's names. The user will select an account number and/or a name. On the click event of the command buttons, we will display the balances in the appropriate text boxes. The output of the application is shown in Figure 3.38. The application has been developed in an `.aspx` file named `XpathDoc2.aspx`. Its complete listing is shown in Figure 3.39. The code is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

To search for a particular value of an attribute (e.g., of an account number) we have used the following expression:

```
Bank/Account[@AccountNo=' '+accNo+' ']/Balance
```

To search for a particular value of an element (e.g., of an account holder's name), we have used the following expression:

```
descendant::Account[child::Name=' '+accName+' ']/Balance
```





**Figure 3.39 Continued**

---

```
' Query to get the balance from AccountNo
myIter=myNav.Select("Bank/Account[@AccountNo='"+accNo+"']/Balance")
myIter.MoveNext()
'Display the values of Balance
txtBalance1.Text=FormatCurrency(myIter.Current.Value)
' Query to get the balance from Name
myIter = myNav.Select _
    ("descendant::Account[child::Name='"+accName+"']/Balance")
myIter.MoveNext()
'Display the values of Balance
txtBalance2.Text=FormatCurrency(myIter.Current.Value)
End Sub
</Script>
```

---

## Transforming an XML Document Using XSLT

Extensible Stylesheet Language Transformations (XSLT) is the transformation component of the XSL specification by W3C ([www.w3.org/Style/XSL](http://www.w3.org/Style/XSL)). It is essentially a template-based declarative language, which can be used to transform an XML document to another XML document or to documents of other types (e.g., HTML and Text). We can develop and apply various XSLT templates to select, filter, and process various parts of an XML document. In .NET, we can use the *Transform()* method of the *XSLTransform* class to transform an XML document.

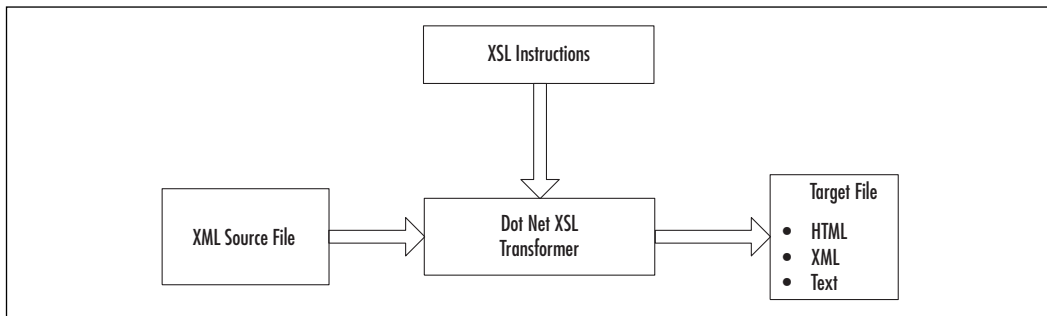
Internet Explorer (5.5 and above) has a built-in XSL transformer that automatically transforms an XML document to an HTML document. When we open an XML document in IE, it displays the data using a collapsible list view. However, the Internet Explorer cannot be used to transform an XML document to another XML document. Now, why would we need to transform an XML document to another XML document? Well, suppose that we have a very large document that contains our entire catalog's data. We want to create another XML document from it, which will contain only the *productId* and *productNames* of those products that belong to the "Fishing" category. We would also like to sort the elements in the ascending order of the unit price. Further, we may want to add a new element in each product, such as "Expensive" or "Cheap" depending



on the price of the product. To solve this particular problem, we may either develop relevant codes in a programming language like C#, or we may use XSLT to accomplish the job. XSLT is a much more convenient way to develop the application, because XSLT has been developed exclusively for these kind of scenarios.

Before we can transform a document, we need to provide the Transformer with the instructions for the desired transformation of the source XML document. These instructions can be coded in XSL. We have illustrated this process in Figure 3.40.

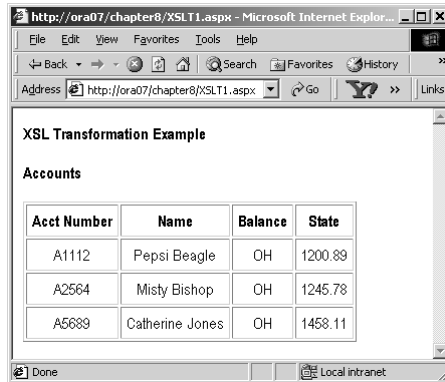
**Figure 3.40** XSL Transformation Process



In this section, we will demonstrate certain selected features of XSLT through some examples. The first example will apply XSLT to transform an XML document to an HTML document. We know that the IE can automatically transform an XML document to a HTML document and can display it on the screen in collapsible list view. However, in this particular example, we do not want to display all of our data in that fashion. We want to display the filtered data in tabular fashion. Thus, we will transform the XML document to a HTML document to our choice (and not to IE's choice). The transformation process will select and filter some XML data to form an HTML table. The second example will transform an XML document to another XML document and subsequently write the resulting document in a disk file, as well as display it in the browser.

## Transforming an XML Document to an HTML Document

In this example, we will apply XSLT to extract the account's information for Ohio customers from the Bank3.xml (as shown in Figure 3.37) document. The extracted data will be finally displayed in an HTML table. The output of the application is shown in Figure 3.41.

**Figure 3.41** Transforming an XML Document to an HTML Document

If we need to use XSLT, we must at first develop the XSLT style sheet (e.g., XSLT instructions). We have saved our style sheet in a file named XSLT1.xsl. In this style sheet, we have defined a template as `<xsl:template match="/"> ... </xsl:template>`. The `match="/"` will result in the selection of nodes at the root of the XML document. Inside the body of this template, we have first included the necessary HTML elements for the desired output.

The `<xsl:for-each select="Bank/Account[State='OH']">` tag is used to select all *Account* nodes for those customers who are from "OH." The value of a node can be shown using a `<xsl:value-of select=attribute or element name>`. In case of an attribute, its name must be prefixed with an @ symbol. For example, we are displaying the value of the State node as `<xsl:value-of select="State"/>`. The complete listing of the XSLT1.xsl file is shown in Figure 3.42. The code is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). In the .aspx file, we have included the following asp:xml control.

```
<asp:xml id="ourXSLTransform" runat="server"
    DocumentSource="Bank3.xml" TransformSource="XSLT1.xsl" />
```

While defining this control, we have set its *DocumentSource* attribute to "Bank3.xml", and its *TransformSource* attribute to XSLT1.xsl. The complete code for the .aspx file, named XSLT1.aspx, is shown in Figure 3.43. It is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.42** Complete Code for XSLT1.xsl

```
<?xml version="1.0" ?>
<!-- Chapter 8\XSLT1.xsl -->
<xsl:stylesheet version="1.0"
```

Continued

[www.syngress.com](http://www.syngress.com)

Figure 3.42 Continued

---

```

    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:template match="/">
  <h4>Accounts</h4>
  <table border="1" cellpadding="5">
    <thead><th>Acct Number</th><th>Name</th>
    <th>Balance</th><th>State</th></thead>

    <xsl:for-each select="Bank/Account[State='OH']" >
      <tr align="center">
        <td><xsl:value-of select="@AccountNo"/></td>
        <td><xsl:value-of select="Name"/></td>
        <td><xsl:value-of select="State"/></td>
        <td><xsl:value-of select="Balance"/></td>
      </tr>
    </xsl:for-each>
  </table>
</xsl:template>
</xsl:stylesheet>

```

---

Figure 3.43 XSLT1.aspx

---

```

<!-- Chapter8\XSLT1.aspx -->
<%@ Page Language="VB" Debug="True"%>
<%@ Import Namespace="System.Xml"%>
<%@ Import Namespace="System.Xml.Xsl"%>
<html><head></head><body><form runat="server">
<b>XSL Transformation Example<nbsp;</b><br/>
<asp:Xml id="ourXSLTransform" runat="server"
    DocumentSource="Bank3.xml" TransformSource="XSLT1.xsl"/>
</form></body></html>

```

---

## Transforming an XML Document into Another XML Document

Suppose that our company has received an order from a customer in XML format. The XML file, named OrderA.xml, is shown in Figure 3.44. The file is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 3.44** An Order Received from a Customer in XML Format (OrderA.xml)

---

```
<?xml version="1.0" ?>
<!-- Chapter 8\OrderA.XML -->
<Order>
  <Agent>Alfred Bishop</Agent>
  <Item>50 GPM Pump</Item>
  <Quantity>10</Quantity>
  <Date>
    <Month>8</Month>
    <Day>24</Day>
    <Year>2001</Year>
  </Date>
  <Customer>Pepsi Beagle</Customer>
</Order>
```

---

Now we want to transmit a purchase order to our supplier to fulfill the previous order. Suppose that the XML format of our purchase order is different from that of our client as shown in Figure 3.45. The OrderB.xml file is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 3.45** The Purchase Order to Be Sent to the Supplier in XML Format (OrderB.xml)

---

```
<?xml version="1.0" encoding="utf-8"?>
<Order>
  <Date>2001/8/24</Date>
  <Customer>Company A</Customer>
  <Item>
    <Sk>P 25-16:3</Sk>
```

---

Continued

**Figure 3.45 Continued**


---

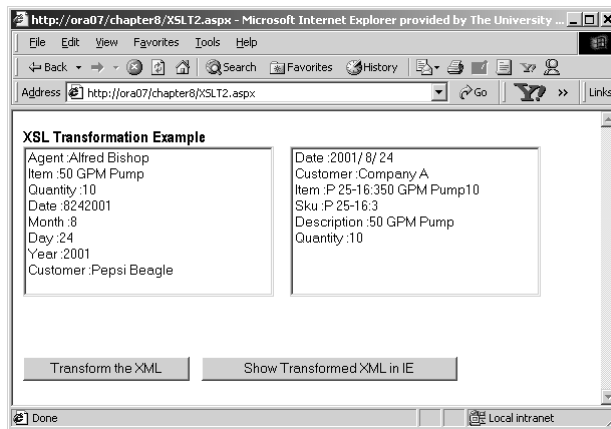
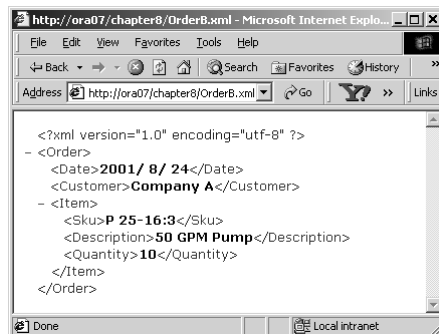
```

<Description>50 GPM Pump</Description>
<Quantity>10</Quantity>
</Item>
</Order>

```

---

The objective of this example is to automatically transform OrderA.xml (Figure 3.44) to OrderB.xml (Figure 3.45). The outputs of this application are shown in Figures 3.46 and 3.47.

**Figure 3.46 Transformation of an XML Document to Another XML Document****Figure 3.47 The Target XML File as Displayed in Internet Explorer**

We have developed an XSLT file (shown in Figure 3.48) to achieve the necessary transformation. In the XSLT code, we have used multiple templates. The complete listing of the XSLT code is shown in Figure 3.48. The code is also

available in the order.xml file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 3.48** Complete Listing of order.xml

```
<?xml version="1.0" ?>
<!-- Chapter 8\order.xml -->
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="xml" indent="yes" />
<xsl:template match="/">
  <Order>
    <Date>
      <xsl:value-of select="/Order/Date/Year" />/
      <xsl:value-of select="/Order/Date/Month" />/
      <xsl:value-of select="/Order/Date/Day" />
    </Date>
    <Customer>Company A</Customer>
    <Item>
      <xsl:apply-templates select="/Order/Item" />
      <Quantity><xsl:value-of select="/Order/Quantity"/></Quantity>
    </Item>
  </Order>
</xsl:template>
<xsl:template match="Item">
  <Sku>
    <xsl:choose>
      <xsl:when test=". = '50 GPM Pump'">P 25-16:3</xsl:when>
      <xsl:when test=". = '100 GPM Pump'">P 35-12:5</xsl:when>
      <!--other Sku would go here-->
      <xsl:otherwise>00</xsl:otherwise>
    </xsl:choose>
  </Sku>
  <Description>
    <xsl:value-of select="." />
  </Description>
</xsl:template>
</xsl:stylesheet>
```



**Figure 3.49 Continued**


---

```

myIterator =myNav.Select("/Order")
myIterator=myNav.SelectDescendants(XPathNodeType.Element,false)
myIterator.MoveNext()
While myIterator.MoveNext()
    ' Add the Items to the DropDownList
    lstInitial.Items.Add _
        (myIterator.Current.Name+" :"+myIterator.Current.Value)
End While
End If
End Sub

Sub showTransformed(sender As Object,e As EventArgs)
    ' Load the XML Document
    Dim myDoc As New XPathDocument(Server.MapPath("OrderA.xml"))
    ' Declare the XSLTransform Object
    Dim myXsltDoc As New XSLTransform
    ' Create the filestream to write a XML file
    Dim myfileStream As New FileStream _
        (Server.MapPath ("OrderB.xml"),FileMode.Create,FileShare.ReadWrite)
    ' Load the XSL file
    myXsltDoc.Load(Server.MapPath("order.xsl"))
    ' Transform the XML file according to XSL Document
    myXsltDoc.Transform(myDoc,Nothing,myfileStream)
    myfileStream.Close()
    lstFinal.Items.Clear
    Dim myDoc2 As New XPathDocument(Server.MapPath("OrderB.xml"))
    Dim myNav As XPath.XPathNavigator
    Dim myIterator As XPath.XPathNodeIterator
    ' Set nav object
    myNav = myDoc2.CreateNavigator()
    ' Iterate through all the attributes of the descendants
    myIterator =myNav.Select("/Order")
    myIterator=myNav.SelectDescendants(XPathNodeType.Element,false)
    myIterator.MoveNext()
    While myIterator.MoveNext()

```

---

Continued



**Figure 3.49 Continued**


---

```

' Add the Items to the DropDownList
lstFinal.Items.Add _
    (myIterator.Current.Name+ " : "+myIterator.Current.Value)
End While
End Sub
Sub showTarget(sender As Object,e As EventArgs)
    Response.Redirect(Server.MapPath("OrderB.xml"))
End Sub
</Script>

```

---

## Working with XML and Databases

Databases are used to store and manage organization's data. However, it is not a simple task to transfer data from the database to a remote client or to a business partner, especially when we do not clearly know how the client will use the sent data. Well, we may send the required data using XML documents. That way, the data container is independent of the client's platform. The databases and other related data stores are here to stay, and XML will not replace these data stores. However, XML will undoubtedly provide a common medium for exchanging data among sources and destinations. It will also allow various software to exchange data among themselves. In this context, the XML forms a bridge between ADO.NET and other applications. Since XML is integrated in the .NET Framework, the data transfer using XML is lot easier than it is in other software development environments. Data can be exchanged from one source to another via XML. The ADO.NET Framework is essentially based on *Datasets*, which, in turn, relies heavily on XML architecture. The *DataSet* class has a rich collection of methods that are related to processing XML. Some of the widely used ones are *ReadXml*, *WriteXml*, *GetXml*, *GetXmlSchema*, *InferXmlSchema*, *ReadXmlSchema*, and *WriteXmlSchema*.

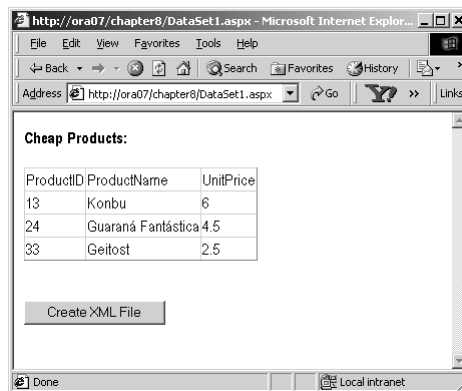
In this context, we will provide two simple examples. In the first example, we will create a *DataSet* from a SQL query, and write its contents as an XML document. In the second example, we will read back the XML document generated in the first example and load a *DataSet*. What are the prospective uses of these examples? Well, suppose that we need to send the products data of our fishing products to a client. In earlier days, we would have sent the data as a text file. But

in the .NET environment, we can instead develop a XML document very fast by running a query, and subsequently send the XML document to our client. What is the advantage? It is fast, easy, self-defined, and technology independent. The client may use any technology (like VB, Java, Oracle, etc.) to parse the XML document and subsequently develop applications. On the other hand, if we receive an XML document from our partners, we may as well apply XML.NET to develop our own applications.

## Creating an XML Document from a Database Query

In this section, we will populate a *DataSet* with the results of a query to the *Products* table of SQL Server 7.0 Northwind database. On the click event of a command button, we will write the XML file and its schema. (The output of the example is shown in Figure 3.50). We have developed the application in an .aspx file named DataSet1.aspx. The complete listing of the .aspx file is shown in Figure 3.51. The file is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 3.50** Output of DataSet1.aspx Application



The XML file created by the application is as follows:

```
<myXMLProduct>
  <dtProducts>
    <ProductID>13</ProductID>
    <ProductName>Konbu</ProductName>
    <UnitPrice>6</UnitPrice>
```

```

    </dtProducts>
    ---  ---  ---
    ---  ---  ---
</myXMLProduct>

```

The code for the illustration is straightforward. The *DataSet's WriteXml* and *WriteXmlSchema* methods were used to accomplish the desired task.

### Figure 3.51 Complete Listing DataSet1.aspx

```

<!-- Chapter8\DataSet1.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<html><head></head><body><form runat="server">
<b>Cheap Products:</b> <br/><br/>
<asp:DataGrid id="myGrid" runat="server"/><br/><br/>
<asp:Button id="cmdWriteXML" Text="Create XML File" runat="server"
    onclick="writeXML"/>
</body></form></html>
<Script Language="vb" runat="server">
Sub Page_Load(s As Object, e As EventArgs)
    If Not Page.IsPostBack Then
        Dim myDataSet As New DataSet("myXMLProduct")
        Dim myConn As New _
            SqlConnection("server=ora07;uid=sa;pwd=ahmed;database=Northwind")
        Dim myDataAdapter As New SqlDataAdapter _
            ("SELECT ProductID,ProductName,UnitPrice FROM Products WHERE
                UnitPrice <7.00",myConn)
        myDataAdapter.Fill(myDataSet,"dtProducts")
        myGrid.DataSource=myDataSet.Tables(0)
        myGrid.DataBind
        Session("sessDs")=myDataSet
    End If
End Sub

```

---

Continued

## Figure 3.51 Continued

```

Sub writeXML(s As Object, e As EventArgs)
Dim myFs1 As New FileStream _
    (Server.MapPath _
        ("myXMLData.xml"), FileMode.Create, FileShare.ReadWrite)
Dim myFs2 As New FileStream(Server.MapPath _
    ("myXMLData.xsd"), FileMode.Create, FileShare.ReadWrite)
Dim myDataSet As New DataSet _
    myDataSet=Session("sessDs")
' Use the WriteXml method of DataSet object to write an XML file
' from the DataSet
myDataSet.WriteXml(myFs1)
myFs1.Close()
myDataSet.WriteXmlSchema(myFs2)
myFs2.Close()

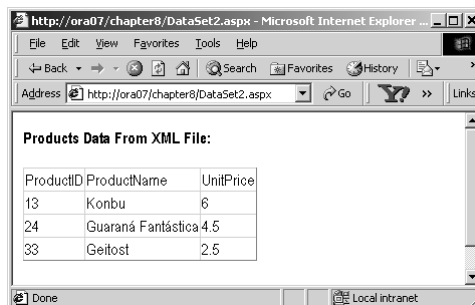
End Sub
</Script>

```

## Reading an XML Document into a DataSet

Here, we will read back the XML file created in the previous example (as shown in Figure 3.50) and populate a *DataSet* in the *Page\_Load* event of our .aspx file. We will use the *ReadXml* method of the *DataSet* object to accomplish this objective. The output of the application is shown in Figure 3.52. The application has been developed in an .aspx file named *DataSet2.aspx*. The complete code for this application is shown in Figure 3.53. The code is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). The code is self-explanatory.

### Figure 3.52 Output of DataSet2.aspx Application




**Figure 3.53** Complete Listing of DataSet2.aspx
 

---

```

<!-- Chapter8\DataSet2.aspx -->
<%@ Page Language = "VB" Debug = "True" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.IO" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<html><head></head><body><form runat="server">
<b>Products Data From XML File:</b> <br/><br/>
<asp:DataGrid id="myGrid" runat="server"/><br/><br/>
</body></form></html>

<Script Language="vb" runat="server">
Sub Page_Load(s As Object, e As EventArgs)
  If Not Page.IsPostBack Then
    Dim myDataSet As New DataSet("myXMLProduct")
    Dim myFs As New FileStream _
      (Server.MapPath("myXMLData.xml"), FileMode.Open, FileShare.ReadWrite)
    myDataSet.ReadXml(myFs)
    myGrid.DataSource=myDataSet.Tables(0)
    myGrid.DataBind
    myFs.Close
  End If
End Sub
</Script>

```

---

## Summary

In this chapter, we have introduced the basic concepts of XML, and we have provided a concise overview of the .NET classes available to read, store, and manipulate XML documents. The examples presented in this chapter also serve as good models for developing business applications using XML and ASP.NET.

The .NET's *System.Xml* namespace contains probably the richest collection of XML-related classes available thus far in any other software development platform. The *System.Xml* namespace has been further enriched by the recent addition of *XPathDocument* and *XPathNavigator* classes. We have tried to highlight these new features in our examples. Since XML can be enhanced using a family of technologies, there are innumerable techniques a reader should judiciously learn from other sources to design, develop, and implement complex real-world applications.

## Solutions Fast Track

### An Overview of XML

- ☑ XML stands for eXtensible Markup Language. It is a subset of a larger framework named SGML. The W3C developed the specifications for SGML and XML.
- ☑ XML provides a universal way for exchanging information between organizations.
- ☑ XML cannot be singled out as a stand-alone technology. It is actually a framework for exchanging data. It is supported by a family of growing technologies such as XML parsers, XSLT transformers, XPath, XLink, and Schema Generators.
- ☑ An XML document may contain Declaration, Comment, Elements, and Attributes.
- ☑ An XML element has a start-tag and an end-tag. An element may contain other elements, or character data, or both.
- ☑ An attribute provides an additional way to attach a piece of data to an element. An attribute must always be enclosed within start-tag of an element, and its value is specified using double quotes.

- ☑ An XML document is said to be well formed when it satisfies a set of syntax-related rules. These rules include the following:
  - The document must have exactly one root element.
  - Each element must have a start-tag and end-tag.
  - The elements must be properly nested.
- ☑ An XML document is case sensitive.
- ☑ DTD and schema are essentially two different ways to specify the rules about the contents of an XML document.
- ☑ An XML schema contains the structure of an XML document, its elements, the data types of the elements and associated attributes including the parent-child relationships among the elements.
- ☑ VS.NET supports the W3C specification for XML Schema Definition (also known as XSD).
- ☑ XML documents store data in hierarchical fashion, also known as a node tree.
- ☑ The top-most node in the node tree is referred to as the root.
- ☑ A particular node in a node tree can be of *element-type*, or of *text-type*. An element-type node contains other element-type nodes or text-type nodes. A text-type node contains only data.

## Processing XML Documents Using .NET

- ☑ The *System.Xml* namespace contains *XmlTextReader*, *XmlValidatingReader*, and *XmlNodeReader* classes for reading XML Documents. The *XmlTextWriter* class enables you to write data as XML documents.
- ☑ *XmlDocument*, *XmlDataDocument*, and *XPathDocument* classes can be used to structure XML data in the memory and to process them.
- ☑ *XPathNavigator* and *XPathNodeIterator* classes enable you to query and retrieve selected data using *XPath* expressions.

## Reading and Parsing Using the *XmlTextReader* Class

- ☑ The *XmlTextReader* class provides a fast forward-only cursor to pull data from an XML document.
- ☑ Some of the frequently used methods and properties of the *XmlTextReader* class include *AttributeCount*, *Depth*, *EOF*, *HasAttributes*, *HasValue*, *IsDefault*, *IsEmptyElement*, *Item*, *ReadState*, and *Value*.
- ☑ The *Read()* of an *XmlTextReader* object enables you to read data sequentially. The *MoveToAttribute()* method can be used to iterate through the attribute collection of an element.

## Writing an XML Document Using the *XmlTextWriter* Class

- ☑ An *XmlTextWriter* class can be used to write data sequentially to an output stream, or to a disk file as an XML document.
- ☑ Its major methods and properties include *Close*, *Flush*, *Formatting*, *WriteAttributes*, *WriteAttributeString*, *WriteComment*, *WriteElementString*, *WriteElementString*, *WriteEndAttribute*, *WriteEndDocument*, *WriteState*, and *WriteStartDocument*.
- ☑ Its constructor contains a parameter that can be used to specify the output format of the XML document. If this parameter is set to “Nothing,” then the document is written using UTF-8 format.

## Exploring the XML Document Object Model

- ☑ The W3C Document Object Model (DOM) is a set of the specifications to represent an XML document in the computer’s memory.
- ☑ *XmlDocument* class implements both the W3C specifications (Core level 1 and 2) of DOM.
- ☑ *XmlDocument* object also allows navigating through XML node tree using *XPath* expressions.
- ☑ *XmlDataDocument* is an extension of *XmlDocument* class.



- ☑ It can be used to generate both the XML view as well as the relational view of the same XML data.
- ☑ *XmlDataDocument* contains a *DataSet* property that exposes its data as relational table(s).

## Querying XML Data Using *XPathDocument* and *XPathNavigator*

- ☑ *XPathDocument* class allows loading XML data in fragments rather than loading the entire DOM tree.
- ☑ *XPathNavigator* object can be used in conjunction with *XPathDocument* for effective navigation through XML data.
- ☑ *XPath* expressions are used in these classes for selecting nodes, iterating over the selected nodes, and working with these nodes for copying, moving, and removal purposes.

## Transforming an XML Document Using XSLT

- ☑ You can use XSLT (XML Style Sheet Language Transformations) to transform an XML document to another XML document or to documents of other types (e.g., HTML and Text).
- ☑ XSLT is a template-based declarative language. We can develop and apply various XSLT templates to select, filter, and process various parts of an XML document.
- ☑ In .NET, you can use the *Transform()* method of *XSLTransform* class to transform an XML document.

## Working with XML and Databases

- ☑ A *DataSet's ReadXml()* can read XML data as *DataTable(s)*.
- ☑ You can create an XML document and its schema from a database query using *DataSet's WriteXml()* and *WriteXmlSchema()*.
- ☑ Some of the widely used ones include *ReadXml*, *WriteXml*, *GetXml*, *GetXmlSchema*, *InferXmlSchema*, *ReadXmlSchema*, and *WriteXmlSchema*.

# Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** What is the difference between DOM Core 1 API and Core 2 API?

**A:** DOM Level 2 became an official World Wide Web Consortium (W3C) recommendation in late November 2000. Although there is not much of difference in the specifications, one of the major features was the namespaces in XML being added, which was unavailable in prior version. DOM Level 1 did not support namespaces. Thus, it was the responsibility of the application programmer to determine the significance of special prefixed tag names. DOM Level 2 supports namespaces by providing new namespace-aware versions of Level 1 methods.

**Q:** What are the major features of System.XML in the Beta 2 edition?

**A:** The most significant change in the Beta 2 edition was the restructuring of the *XmlNavigator* Class. *XmlNavigator* initially was designed as an alternative to the general implementation of DOM. Since Microsoft felt that there was a mismatch in the *XPath* data model and DOM-based data model, *XmlNavigator* was redesigned to *XpathNavigator*, employing a read-only mechanism. It was conceived of using with *XPathNodeIterator* that acts as an iterator over a node set and can be created many times per *XPathNavigator*.

Alternatively, one can have the DOM implementation as *XmlNode*, and methods such as *SelectNodes()* and *SelectSingleNode()* can be used to iterate through a node set. A typical code fragment would look like this:

```
Dim nodeList as XmlNodeList
Dim root as XmlElement = Doc.DocumentElement
nodeList = root.SelectNodes("descendant::account[child::State='OH']")
Dim entry as XmlNode
    For Each entry in nodeList
        'Do the requisite operations
    Next
```

Although *XPathNavigator* is implemented as a read-only mechanism to manipulate the XML documents, it can be noted that certain other classes like *XmlTextWriter* can be implemented over *XPathNavigator* to write to the document.

**Q:** How is *XPath* different from XSL Patterns?

**A:** XSL Patterns are predecessors of *XPath 1.0* that have been recognized as a universal specification. Although similar in syntax, there are some differences between them. XSL pattern language does not support the notion of axis types. On the other hand, the *XPath* supports axis types. Axis types are general syntax used in *Xpath*, such as descendant, parent, child, and so on. Assume that we have an XML document with the root node named *Bank*. Further, assume that the *Bank* element contains many *Account* elements, which in turn contains *account number*, *name*, *balance*, and *state* elements. Now, suppose that our objective is to retrieve the *Account* data for those customers who are from Ohio. We can accomplish the search by using any one of the following alternatives:

- XSL Pattern Alternative: `Bank/Account[child::State='OH']`
- XSL Path 1.0 Alternative: `descendant::Account[child::State='OH']`

Which of the above alternatives would you use? That depends on your personal taste and on the structure of the XML document. In case of a very highly nested XML document, the XSL Path offers more compact search string.

## Information Exchange Using the Simple Object Access Protocol (SOAP)

### Solutions in this chapter:

- The Case for Web Services
- Working with Web Services
- Advanced Web Services
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

# Introduction

The growth of the Internet demands that businesses provide clients with a better, more efficient user experience. Existing technologies have made it very difficult to make applications communicate with each other across businesses. The varied resources used, such as operating systems (OSs), programming languages and object models, pose big challenges to application integrators.

Web Services have been created to solve the interoperability of applications across operating systems, programming languages, and object models. Web Services can achieve this by relying on well supported Internet standards, such as Hypertext Transfer Protocol (HTTP) and Extensible Markup Language (XML).

In this chapter, we tell you why Web Services are an important new development in the area of Internet standards, and what business problems they address. We talk about the Simple Object Access Protocol (SOAP), which lets you exchange data and documents over the Internet in a well-defined way, and related standards to describe and discover Web Services. Finally, we cover techniques for error handling and state management and discuss how Web Services integrate with the Microsoft .NET platform.

## The Case for Web Services

In a broad sense, Web Services may be defined as “Internet-based modular applications that perform specific business tasks and conform to a specific technical format,” to quote Mark Colan from IBM. If you accept this definition, you may have very well already developed a number of Web Services. However, the crux of this definition is the “specific technical format.” Similar to the way a network becomes more and more useful with the number of systems participating on that network, data interchange between those systems becomes more and more powerful as the interchange conforms to a common format. Everybody can come up with their own protocols to exchange data, and in the past, many people indeed have designed such protocols, but to make distributed application development a reality and have it be truly useful, clearly a common, open, standards-based, universally adopted mechanism needs to be agreed upon. And this is where the more narrow definition of a Web Service comes in: A Web Service is a Web application using the SOAP protocol.

## The Role of SOAP

SOAP stands for *Simple Object Access Protocol*. SOAP was designed with the following three goals in mind:

- It should be optimized to run on the Internet.
- It should be simple and easy to implement.
- It should be based on XML.

SOAP is an open Internet standard. It was originally proposed by IBM, Ariba, and Microsoft, and the W3C has taken on the initiative to develop it further. The current version is SOAP 1.1 (April 2000). You can find the specifications at [www.w3.org/TR/SOAP](http://www.w3.org/TR/SOAP). Work is currently under way on version 1.2 (see the W3C working draft at [www.w3.org/TR/soap12](http://www.w3.org/TR/soap12)), which is, in our opinion, only a minor revision. You can join the authoritative discussion list for SOAP by going to <http://discuss.develop.com/soap.html>.

SOAP, somewhat contrary to its name, is fundamentally just a protocol that lets two systems—a client and a server—exchange data. Of course, the client system may be, and often is, just another server machine, not a human end user.

Although the SOAP specification was written in such a way as to be implemented on a variety of Internet transport protocols, it is most often used on top of HTTP. In our discussions that follow, when we talk about SOAP and Web Services, we always mean SOAP over HTTP (or Secure HTTP [HTTPS], for that matter).

SOAP supports two message patterns: the first is a simple one-way exchange, where a client issues a request against a server, and will not receive an answer back. We focus in this chapter on the second message pattern, which consists of a request-response interaction, familiar to all Web developers. A client issues an HTTP request for a resource on a server, and the server replies by sending an HTTP response. SOAP adds to that a standard way to pass data back and forth, including a standard way to report errors back to the client. In traditional Web applications, the only thing that's standardized in a Web request is the URL, the HTTP verb (*GET*, *PUT*, and so on), and some of the HTTP headers. Everything else is specific to the application at hand, particularly as it relates to the passing of application-specific data and data structures. A client can, say, *POST* additional information using the form submission mechanism. But imagine that you'd like to post a series of floating point numbers to a server. How would you do that? How would you ensure that the server understands what you're sending it? How

would you ensure that the data goes to the right place on the server? SOAP addresses these challenges by defining the following:

- A mechanism to pass simple and structured data between clients and servers using a standard XML syntax
- A mechanism to call objects running remotely on a server

SOAP has two faces. On the one hand, stressing the second item in the preceding list, you can look at it as a remote procedure call (RPC) protocol familiar to anybody who has worked with distributed object models in the past. On the other hand, putting more emphasis on the first item, you can consider it a standardized way to interchange (XML) documents.

However, SOAP being a “simple” protocol, it does not by itself define a number of added-value mechanisms familiar to application developers using not-so-simple protocols (such as Common Object Request Broker Architecture [CORBA] or Component Object Model [COM]/Distributed COM [DCOM]):

- Security
- Transaction management
- Guaranteed delivery

## Why SOAP?

SOAP is not the first attempt at standardizing on an RPC and document interchange mechanism, and it may not be the last one. In the RPC area, previous attempts include CORBA and COM/DCOM, which originated in the client-server world, but both of which now include functionality to work more or less well on the Internet, and David Winer’s XML-RPC (see [www.xmlrpc.com/spec/](http://www.xmlrpc.com/spec/)), which was designed from the ground up to work over the Internet. In the document area, we have seen EDI come (and go?). What makes SOAP important and, quite frankly, remarkable, is that it is supported by *all* major players in the business, including, from the very beginning, IBM and Microsoft, and more recently, Sun Microsystems. The same universal support is true of a number of related standards, such as Web Services Description Language (WSDL) and Universal Description, Discovery, and Integration (UDDI), which we discuss later in this chapter.

As Microsoft developers, we should take notice of the fact that the new Microsoft .NET Framework is currently the *only* system platform that was designed *from the ground up* based on Web Services and SOAP.

## Why Web Services?

The recent emphasis on Web Services denotes a noteworthy shift in application development: away from insular, monolithic solutions and towards truly distributed, modular, open, interenterprise Internet-based applications. The hope certainly is that Web Services will do to enterprise applications what the World Wide Web did to interactive end user applications. In our opinion, then, Web Services are primarily a technique that allows disparate *server* systems to talk to each other and exchange information, and maybe less a mechanism directly encountered by human end users, for which the Web and the traditional Web browser remains the primary data access point. If you have ever been involved in trying to integrate different systems from different vendor companies, you know how painful an endeavor this can be. Integrating one system with one other system, although often very complex, can usually be done *somehow*, but integrating many systems with many other systems is really beyond the capabilities of any of the current middleware solutions, particularly if done intercompanywide over public networks. SOAP and Web Services offer hope here, because that technique is simple, and because it is a universally accepted standard.

We should all imagine a whole new class of applications appearing on the horizon very soon: massively distributed applications, integrating data from many sources from many different systems all over the world, very fault-tolerant, and accessible at all times from anywhere.

## Wiring Up Distributed Objects—The SOAP Protocol

SOAP is the standard used to exchange data over the Internet using Web Services. SOAP is commonly referred to as a *wiring protocol*. As with many other standards, it is often more helpful to see some examples of the standard in action before moving on to reading the standards document. Using Visual Studio.NET, it is very easy to create simple Web Services and see how data is being exchanged. Because SOAP is based on XML and not a binary protocol, such as DCOM, you can inspect the data exchange in detail using a network tunneling tool and see exactly what is going on under the hood.

## Creating Your Very First Web Service

Let's look at a SOAP exchange between a client and a server by way of a few examples. Although Web Services are most interesting when used to couple



server computers, our examples are more geared towards end users interacting with a Web Service server; we only do this to keep the examples reasonably simple and self-contained.

As mentioned earlier, we look only at SOAP as implemented over the HTTP protocol. Also, we initially focus on SOAP as an RPC mechanism.

Let's start by setting up a simple echo Web Service. This service simply returns whatever character string a user submits. Creating a class that echoes its input is fairly straightforward as shown in Figure 4.1.

### Figure 4.1 Echo Method

---

```
namespace soapExamples
{
    public class simpleService {

        public simpleService() {
        }

        public string echo(string input) {
            return input;
        }
    }
}
```

---

How can you now make this into a Web Service? In other words, what is needed to make this method accessible to everybody in the world who has an Internet connection and knows where to find your method?

It may be hard to believe initially, but all that's needed using the .NET Framework is—apart from an Internet Information Server (IIS) Web server, of course—two tiny little changes:

- Your class *simpleService* needs to inherit from *System.Web.Services.WebService*.
- Your method *echo* needs to be decorated with the *System.Web.Services.WebMethod* attribute.

See Figure 4.2 for your first fully functioning Web Service. Note that the complete code for the *echo* Web method is in the directory `soapExamples/` on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 4.2 Echo Web Method (simpleService.asmx.cs)**

```
namespace soapExamples
{
    public class simpleService : System.Web.Services.WebService
    {
        public simpleService() {
        }

        protected override void Dispose( bool disposing ) {
        }

        [System.Web.Services.WebMethod]
        public string echo(string input) {
            return input;
        }
    }
}
```

Let's now open up the Visual Studio.NET integrated development environment and create the echo Web Service from scratch, proceeding as follows:

1. Create a new ASP.NET Web Service called *soapExamples*: Go to **File | New | Project**, choose the entry **ASP.NET Web Service** under the **Visual C# Projects** folder, keep the default Location, and enter **soapExamples** as the Name of the project (see Figure 4.3). This will set up a new virtual directory of the same name (see Figure 4.4).
2. Visual Studio.NET will then configure the necessary FrontPage server extensions, define an assembly, and create supporting project files for you. Annoyingly, the wizard also creates a default Web Service file called *Service1.asmx*, which you may remove in the Solution Explorer by right-clicking on the file and selecting **Delete**. Or, you can simply rename that file to **simpleService.asmx** in the Solution Explorer and proceed with Step 4.
3. Now you create your actual Web Service: Right-click on the **soapExamples** project in the Solution Explorer, and choose **Add | Add New Item**. Choose **Web Service** from the list of available templates, and call it **simpleService.asmx** (see Figure 4.5).

Figure 4.3 Setting Up a New ASP.NET Web Service

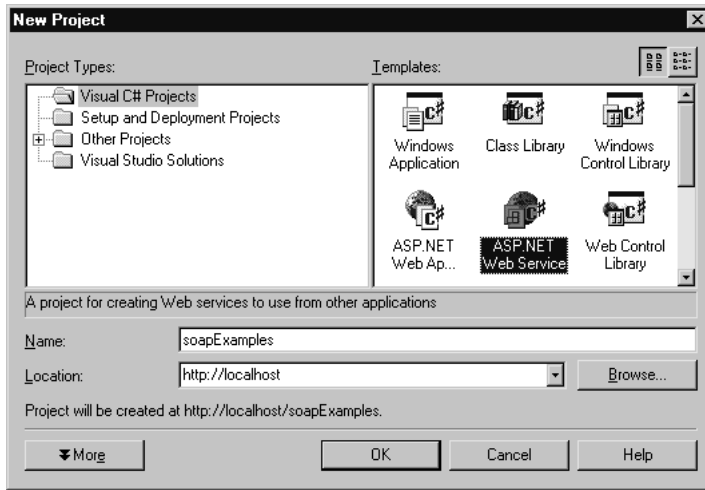


Figure 4.4 Visual Studio.NET Automatically Sets Up a New Web

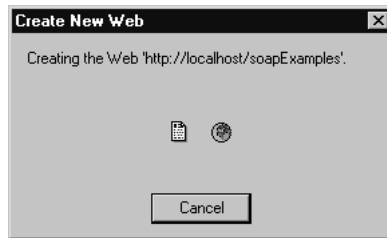
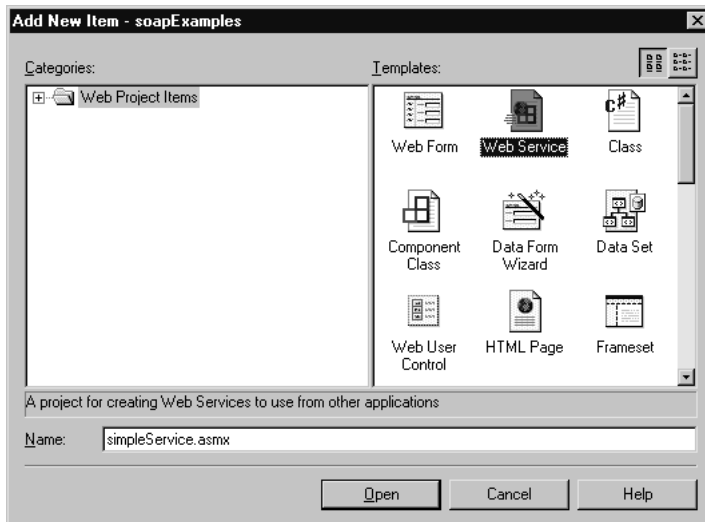


Figure 4.5 Creating a New Web Service



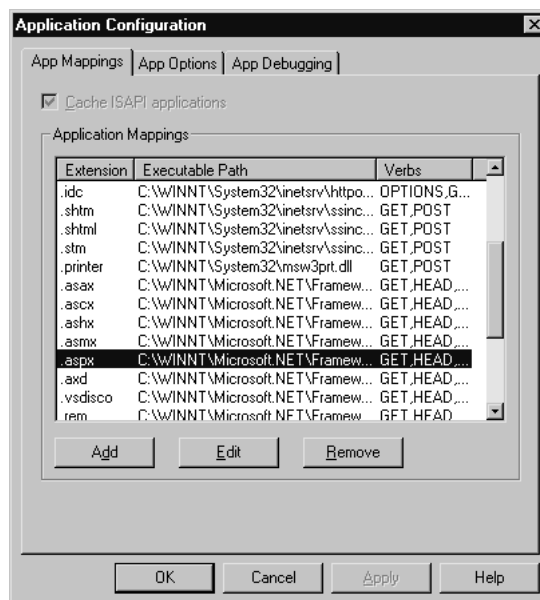
4. Select the Web Service **simpleService.asmx** in the Solution Explorer, and click on the little **View Code** icon to see the code for this Web Service added by the wizard.
5. Replace the code with the code for this class shown in Figure 4.2.
6. The last step is the most remarkable step if you've been used to traditional ASP developing. Compile your project: select **Build | Build** from the User menu, or press **Ctrl+Shift+B**. In other words, ASP.NET applications, such as a Web Service application, are *compiled* applications (and yes, it will create a .NET DLL for you!).

### *How Does Visual Studio.NET Organize Your Project?*

When you tell Visual Studio.NET to create a new Web Service application, the following process happens, using this section's example of an application called *soapExamples*:

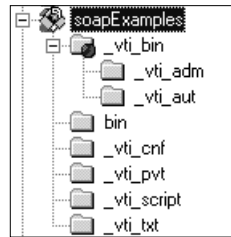
1. A new IIS virtual directory called *soapExamples* is created in %SystemDrive%\InetPub\wwwroot\. As part of the .NET Framework installation, application mappings were already added to map .NET specific file extensions, such as .aspx, to the .NET DLL aspnet\_isapi.dll, located in %SystemRoot%\Microsoft.NET\Framework\v1.0.2914\, which handles .NET-specific Web requests (see Figure 4.6).

**Figure 4.6** Mapping .NET File Extensions



2. The IIS directory is converted to a FrontPage Server Extensions Web, allowing for Visual Studio.NET design support
3. Under the IIS virtual directory, a variety of standard FrontPage directories are created (see Figure 4.7).

**Figure 4.7** Directory Structure for New ASP.NET Web Service



4. The bin directory is created underneath the IIS virtual directory. It will contain the compiled application.
5. A number of files are created and placed in the IIS virtual directory, as described in Table 4.1.

**Table 4.1** Files Created by Visual Studio.NET for *soapExamples* Web Service

File Name	Description
soapExamples.csproj	XML file containing project-level settings, such as a list of all files contained in this project.
soapExamples.csproj.webinfo	XML file containing Web-related project-level settings, such as the URL to start this application.
soapExamples.vsdisco	XML file containing DISCO dynamic discovery information for this Web Service.
AssemblyInfo.cs	C# class defining assembly metadata, such as version number information.
Web.Config	XML file containing configuration for the Web Service, such as security, session handling, and debug settings.
Global.asax	Equivalent to Global.asa file in plain ASP. Points to C# class file Global.asax.cs.
Global.asax.cs	C# class file containing instructions on what to do during events generated by ASP.NET, such as when a new application starts or shuts down.

Continued

**Table 4.1** Continued

File Name	Description
Global.asax.resx	Resource file to store localization information for Global.asax. Empty by default.
Service1.asmx	Sample Web Service file, pointing to C# class file Service1.asmx.cs, created automatically by Visual Studio.NET.
Service1.asmx.cs	Sample C# Web Service class file, created automatically by Visual Studio.NET.
Service1.asmx.resx	Sample Web Service resource file to store localization information for Service1.asmx. Empty by default. Created automatically by Visual Studio.NET.

6. A directory called soapExamples is created in %USERPROFILE%\My Documents\Visual Studio Projects\. Two files are created: soapExamples.sln, a text file containing information as to what projects are contained in the Visual Studio.NET solution, and soapExamples.suo, a binary solution configuration file that cannot be edited directly.
7. A directory called soapExamples is created in %USERPROFILE%\VSWebCache\ATURTSCHI\. This directory and various subdirectories created underneath it contain the cached version of your Web Service. You should normally not need to make any changes here, although it can happen that the files here get out of synch with the files in the “normal” Web directory underneath InetPub\wwwroot, in which case you may have to manually copy some files around.

## Developing & Deploying...

### Separating Design and Code

Microsoft .NET makes a big step forward in neatly separating Web page *design* from Web page *code*. There are actually two files for every Web page: One file that holds all visual elements of a Web page, and another file linked to it that holds the business logic for that page. Web Services

Continued

are ASP.NET Web applications, and therefore incorporate the same mechanism. Because Web Services don't have a user interface as such, the only content of the Web Service Web page is a directive linking it to the Web Service class that contains all the code to handle Web Service requests.

For the *simpleService* Web Service, the corresponding "front end" file, *soapExamples.asmx*, looks as follows:

```
<%@ WebService Language="c#" Codebehind="simpleService.asmx.cs"
    Class="soapExamples.simpleService" %>
```

The *Codebehind* attribute points to the Web Service class file, which by default has the same name as the ASMX file, with a file extension appended reflecting the programming language used, in this case *.cs* for C#.

In order to keep things "simple," the Visual Studio.NET user interface does not keep those two files apart, which may lead a little bit to confusion. Instead, similar to what you may be used to in the Visual Basic 6 form designer, you switch between design mode (the Web form), and code mode (the underlying code) by clicking the corresponding icons in the Solution Explorer. However, and this may throw you off a bit initially, the files that keep the design and code content really *are* different files; however, Solution Explorer pretends that only one of the files, namely the one containing the page design, exists. You can force Solution Explorer to show you the file containing the page code by clicking the **Show All Files** icon, however even when you then explicitly click the code file, Visual Studio.NET will *still* show you the design page, not the code page.

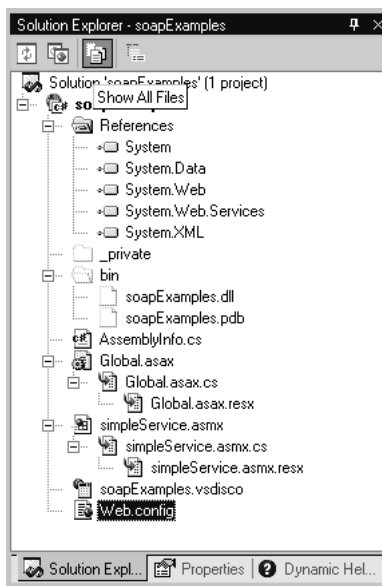
Not all of those files can be made visible in Visual Studio.NET. However, you can see many of them by clicking on the **Show All Files** icon in the Solution Explorer (see Figure 4.8).

## Running Your Very First Web Service

Now that you have developed a simple Web Service, you would obviously like to see it in action, if only to check that everything works the way you expect it to work. Because a Web Service at its core really isn't anything else than a very special Web application, you have the usual means of testing and debugging at your disposal. These are running the Web Service through Visual Studio.NET, our preferred integrated development platform, or calling it through a custom client

application, such as a simple Visual Basic script. In addition, you have the option of *automatically* generating a client application that calls your Web Service through the Web Reference mechanism. Let's go through each of these three scenarios in detail.

**Figure 4.8** Showing All Files through Solution Explorer



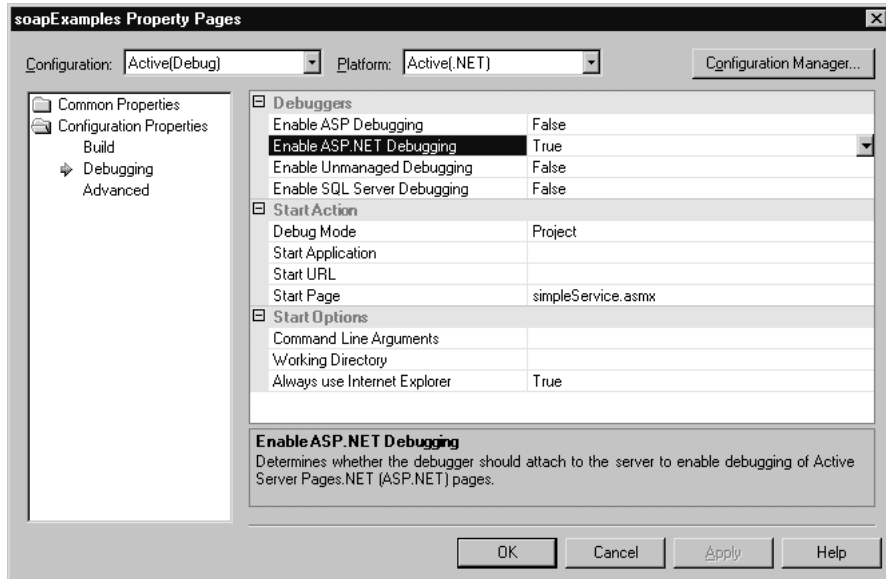
### *Testing a Web Service Using Integrated Visual Studio.NET Debugging*

If you want to test your Web Service through the debugger that comes with Visual Studio.NET, you first need to check and/or change some settings to enable Visual Studio.NET to debug the application properly:

1. Click on the file **Web.config** in the Solution Explorer. Scan through it and make sure that the *debug* attribute of the *compilation* element is set to True (which is the default). This will cause debug information to be included in the compiled DLL. Obviously, you want to change this setting once you're ready to deploy your application.
2. Go to Solution Explorer and right-click on the **soapExamples** project folder to select its Properties. Under the Configuration Properties folder, click **Debugging** and make sure that ASP.NET Debugging is enabled, as shown in Figure 4.9.

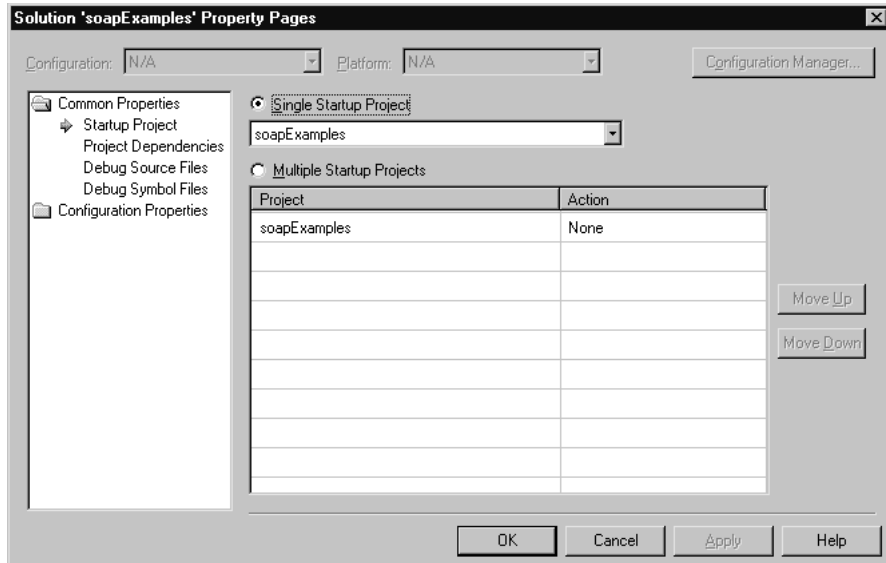


**Figure 4.9** Enabling ASP.NET Debugging



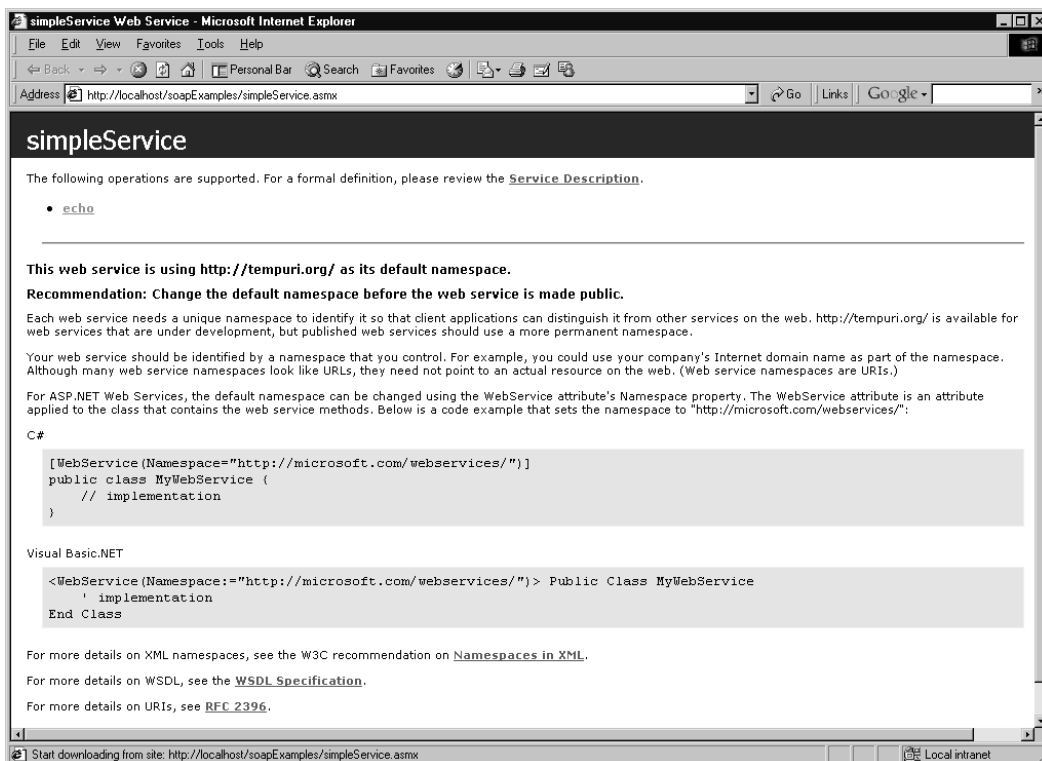
3. Right-click on the file **simpleService.asmx**, which is the file defining your actual Web Service, and select **Set As Start Page**. (Or, you can select this service as the solution Startup Project through the Properties page of the soapExamples solution, as shown in Figure 4.10).

**Figure 4.10** Defining a Startup Project



4. You can now start testing your application by pressing **F5** or by choosing **Debug | Start** through the User menu. As usual, you can set breakpoints anywhere in your code by simply pressing **F9** or selecting **Debug | New Breakpoint** on a line of code through the User menu.
5. Visual Studio.NET will recompile the application, just to be sure, and launch Internet Explorer (see Figure 4.11).

**Figure 4.11** Starting the Web Service in Debug Mode



Note the URL convention used by Microsoft .NET. Immediately after the host name (*localhost*) is the name of the application (*soapExamples*), followed by the name of the Web Service (*simpleService*), or rather, the name of the corresponding Web Service definition file, which has the .asmx file extension.

ASP.NET runtime warns you that you are using the default namespace `http://tempuri.org`. Every .NET class lives in a namespace. Similarly, every Web Service *must* live in a namespace that is exposed globally. This Web Service namespace allows application developers worldwide to distinguish their Web Services from Web Services built by other people. The URL under which a Web Service

can be reached, in this case `http://localhost/soapExamples/simpleService.asmx`, is only an attribute of a *concrete instance* of a Web Service; this Web Service could potentially live on many servers. So, you need to give your Web Service a distinguishing name through the usage of a namespace. By default, ASP.NET will use `http://tempuri.org/`, but you should really change this. Namespaces are created by using a URI (Uniform Resource Identifier), which really can be anything (see [www.faqs.org/rfcs/rfc2396.html](http://www.faqs.org/rfcs/rfc2396.html) for an explanation of URIs). Common choices include using your DNS entry in order to get a unique name.

Let's then take the namespace related runtime warning in Figure 4.11 seriously; stop the debugger by pressing **Shift-F5**, and include Web Service namespace definitions in the code; `urn:schemas-syngress-com-soap` seems like a good URI, and then simply add a namespace attribute with that value next to the Web Service class definition, as shown in Figure 4.12 (changes in bold). The code for Figure 4.12 can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 4.12** Including a Namespace Definition (`simpleService.asmx.cs`)

```
namespace soapExamples
{
    [System.Web.Services.WebServiceAttribute(
        Namespace="urn:schemas-syngress-com-soap")]
    public class simpleService : System.Web.Services.WebService
    {
        public simpleService() {
        }

        [System.Web.Services.WebMethod]
        public string echo(string input) {
            return input;
        }
    }
}
```

After recompiling and restarting the application, you are presented with a screen as in Figure 4.13.

We look at the service description in the next section on WSDL; for now, just click on the **echo** link, which will take you to the welcome screen for the echo Web Service method, as depicted in Figure 4.14.

Figure 4.13 Web Service in Debug Mode after a Namespace Has Been Added

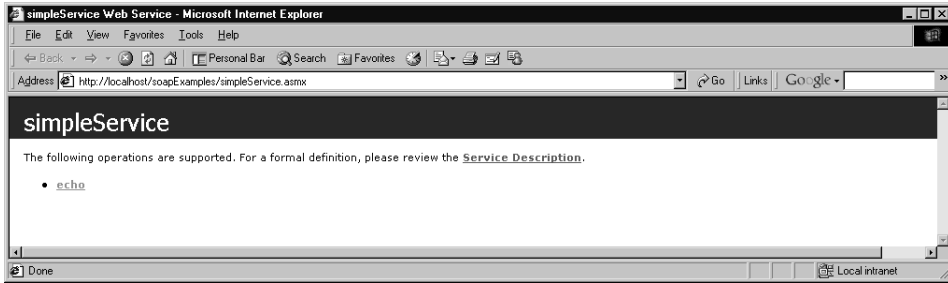


Figure 4.14 The echo Web Service Method



The URL convention adopted by Microsoft .NET for the Web method welcome screen is to append the name of the exposed Web method through an *op=WebMethodName* URL parameter, in this case *op=echo*. To actually call the Web method, the convention is to just add the name of the Web method to the URL, and to append the name of input parameters as URL parameters to the end, as you'll see in a second.

Enter a value, say "Hello World", in the text box labeled *input*, which by the way, corresponds of course to the only input parameter you have defined for the echo Web method, and click **Invoke**. This then takes you to the output screen, as shown in Figure 4.15.

**Figure 4.15** Output of the *echo* Web Method



The input has been echoed in something that clearly looks like XML. What has happened here? As it turns out, this hasn't quite been SOAP yet, but something close. As you can see in Figure 4.14, Microsoft offers you *three ways* to call a Web Service:

- Through a straight HTTP GET
- Through a straight HTTP POST
- Through SOAP

Calling a Web Service through an HTTP GET is a simplified way to call a Web Service. Particularly, it allows you to call a Web Service through a Web browser. The only thing you need to do is to append the method name to the URL of the Web Service, and to add the parameters the way you would usually add variables when submitting an HTML form in a Web application:

```
http://localhost/soapExamples/simpleService.asmx/echo?input=Hello+World
```

The result that you get from this call (see the following):

```
<?xml version="1.0" encoding="utf-8" ?>
<string xmlns="urn:schemas-syngress-com-soap">Hello World</string>
```

This is an XML-ish representation of the fact that the return argument is a string, living in the *urn:schemas-syngress-com-soap* namespace, and having a value of “Hello World”.

As you can imagine, this technique will work only for the very simplest of Web Services. What if you wanted to pass a complex data type to the Web Service? What if you wanted to pass an XML document to the Web Service?

The *POST* method offered to you by Visual Studio.NET in Figure 4.14 is very similar to the *GET* method, the only difference being that the parameter values are put into the body of the HTTP request, exactly the way you would if you *POST*ed information to a Web application through a form.

This technique of calling Web Services through simple HTTP *GET*s and *POST*s is *not* the standard way of calling Web Services. It is very inflexible, and in fact not supported by most vendors. On the other hand, until such time as SOAP will become universal and supported natively by all client applications, you may find simple *GET*s and *POST*s useful in cases where clients don't yet understand SOAP, but do have XML processing capabilities, as is the case with Macromedia Flash 5.0.

Our suggestion, then, is to forgo convenience, and use the SOAP protocol for calling Web Services from the very start. Unfortunately, this means that you have to do a little bit more work.

### *Testing a Web Service Using a Client Script*

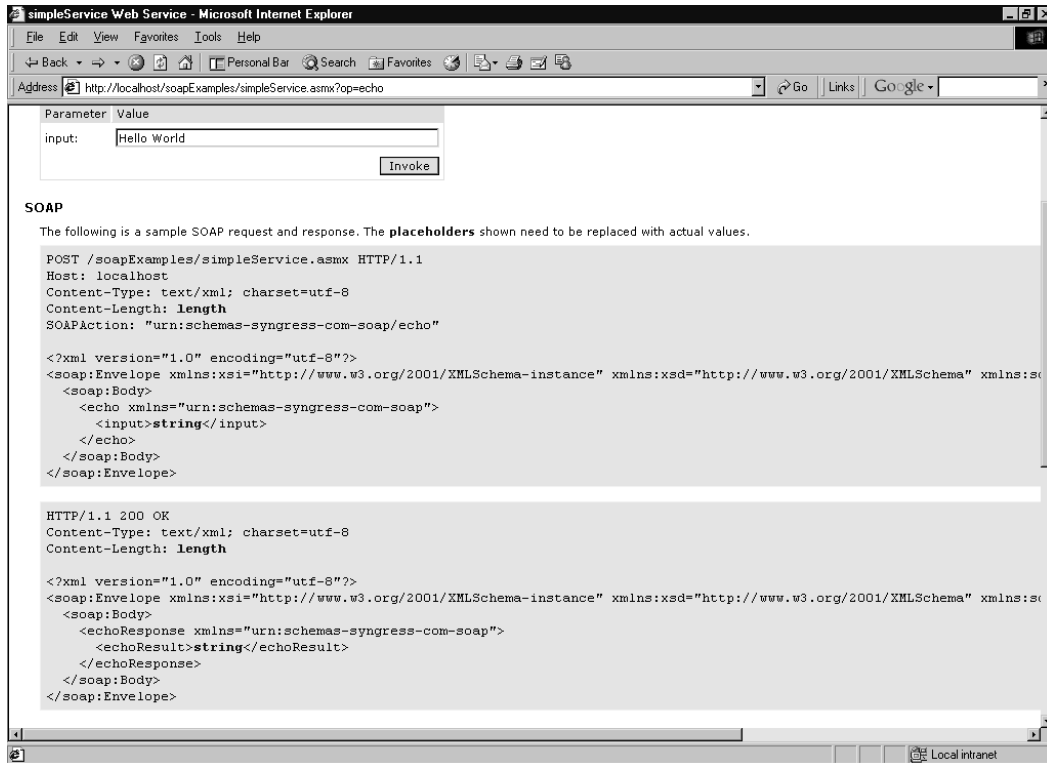
What do you need to do to call the *echo* Web method through proper SOAP? On the Web Service overview screen, as depicted in Figure 4.16, you can get all the information you need.

You can make the following observations for the *SOAP request*:

- A SOAP request is issued using an HTTP *POST*.
- The request is *POST*ed to the Web Service ASMX page (<http://localhost/soapExamples/simpleService.asmx>, in this case).
- SOAP uses an additional HTTP header, called *SOAPAction*, that contains the URI of the Web Service followed by the Web method name (*urn:schemas-syngress-com-soap/echo* in this case).
- The HTTP body of the *POST* contains an XML document, called the *SOAP envelope*, delimited by an `<Envelope>` tag.

- The SOAP envelope itself has a `<Body>` element, and within that element are elements defining the Web method you are calling (`<echo>`) and what parameters it takes (`<input>`).

**Figure 4.16** The SOAP Section of the Web Service Overview Screen



For the *SOAP response*, in turn:

- The SOAP response is a normal HTTP response.
- The HTTP body of the SOAP response contains an XML document, called the *SOAP envelope*, that has the same structure as the SOAP request envelope discussed in the preceding list.
- The SOAP envelope itself has a `<Body>` element, and within that body element are elements declaring the response from the Web method (the default is adding the word *Response* to the method name (that is, `<echoResponse>`), along with the return argument (the default is adding the word *Result* to the method name, that is, `<echoResult>` here).

A detailed discussion of the SOAP protocol is well beyond the scope of this book, however, the basic structure of the SOAP protocol is already apparent:

- Requests are *POST*ed to a server, which in turn issues a response to the client.
- All requests and responses are XML documents, that start with `<Envelope>` and `<Body>` elements. Method names show up within the SOAP Body section, and method arguments and return values in turn show up within the method section.
- The server finds the Web class that handles the request through a combination of the URL to the corresponding ASMX file in the HTTP request, the *SOAPAction* header, and the XML element having the name of the Web method to call following immediately after the SOAP *Body* element.

Because Visual Studio.NET does not currently support directly calling a Web method through SOAP (unless you use Web References, which you will do in the next subsection), let's write a little standalone Visual Basic VBS script instead. Simply take the SOAP request shown in Figure 4.16 and *POST* that information to the Web Server using the Microsoft.XMLHTTP ActiveX control, as shown in Figure 4.17. The code for the script shown in Figure 4.17 is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 4.17** VBS Script to Test the *echo* Web Method (*echo.vbs*)

```
myWebService = "http://localhost/soapExamples/simpleService.asmx"
myMethod = "urn:schemas-syngress-com-soap/echo"

' ** create the SOAP envelope with the request
s = ""
s = s & "<?xml version=""1.0"" encoding=""utf-8""?>" & vbCrLf
s = s & "<soap:Envelope "
s = s & "   xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance"" "
s = s & "   xmlns:xsd=""http://www.w3.org/2001/XMLSchema"" "
s = s & "   xmlns:soap=""http://schemas.xmlsoap.org/soap/envelope/" ">"
s = s & vbCrLf
s = s & "   <soap:Body>" & vbCrLf
s = s & "       <echo xmlns=""urn:schemas-syngress-com-soap"">" & vbCrLf
s = s & "           <input>Hello World</input>" & vbCrLf
s = s & "       </echo>" & vbCrLf
```

Continued

[www.syngress.com](http://www.syngress.com)



**Figure 4.17 Continued**


---

```

s = s & " </soap:Body>" & vbCrLf
s = s & "</soap:Envelope>" & vbCrLf

msgbox(s)

set requestHTTP = CreateObject("Microsoft.XMLHTTP")

msgbox("xmlhttp object created")

requestHTTP.open "POST", myWebService, false
requestHTTP.setRequestHeader "Content-Type", "text/xml"
requestHTTP.setRequestHeader "SOAPAction", myMethod
requestHTTP.Send s

msgbox("request sent")

set responseDocument = requestHTTP.responseXML

msgbox("http return status code: " & requestHTTP.status)
msgbox(responseDocument.xml)

```

---

Because this is a simple Visual Basic script file, you can run it by simply double-clicking on it in Windows Explorer, which will start Windows Scripting Host. The script will show us the SOAP request (see Figure 4.18), send it to the server, tell us that it received an HTTP 200 status return code (see Figure 4.19), which means that everything went smoothly, and then display the SOAP response that includes the echoed input parameter (see Figure 4.20).

**Figure 4.18 Sending a SOAP Request**

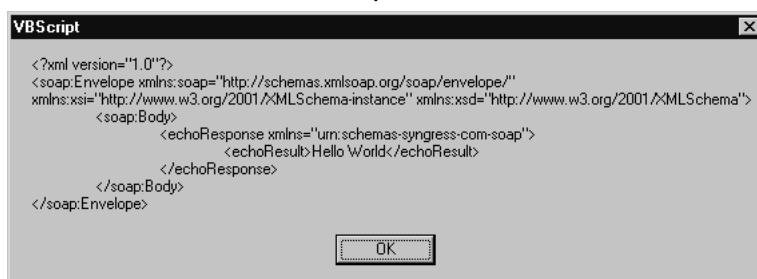

The screenshot shows a window titled "VBScript" with a close button in the top right corner. The main area contains the following XML code:

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <echo xmlns="urn:schemas-syngress-com-soap">
      <input>Hello World</input>
    </echo>
  </soap:Body>
</soap:Envelope>

```

At the bottom center of the dialog box is an "OK" button.

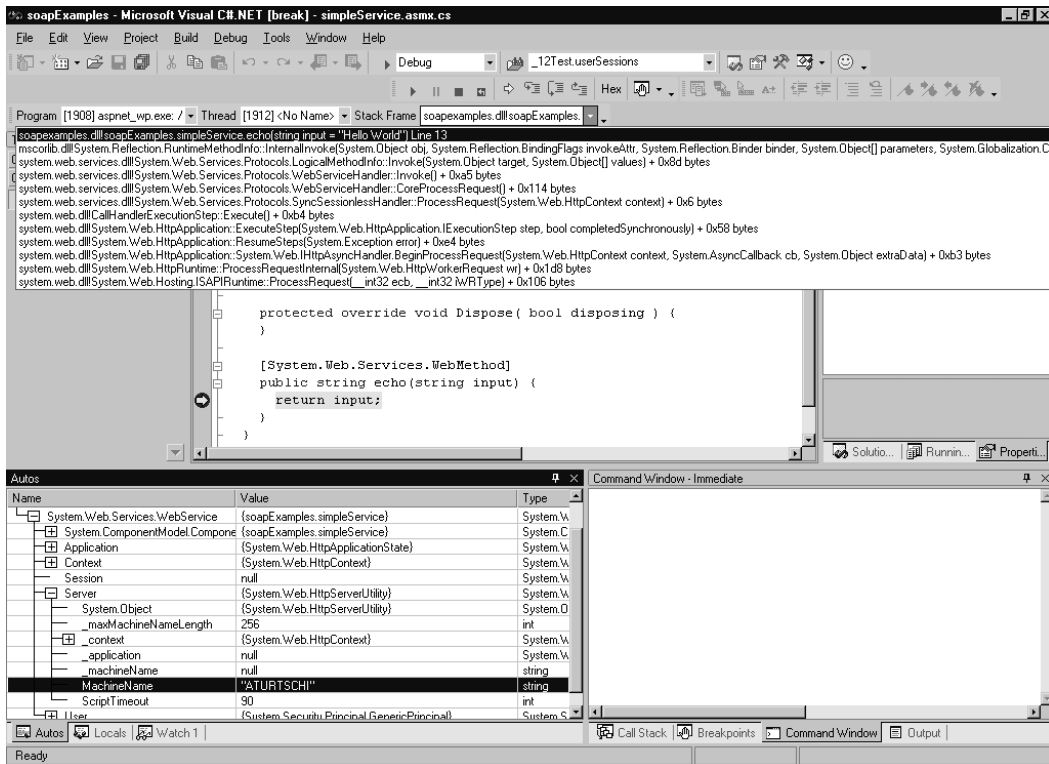
**Figure 4.19** Retrieving a Successful Http Status Code**Figure 4.20** The Successful SOAP Response

The truly amazing fact, however, is that you can run this script, which is not connected in any way to your Visual Studio.NET project, in debug mode. In other words, if you set a breakpoint in one of your project files, start the debugger (by just pressing **F5**), and then go to Windows Explorer or to a command line and run the script in Figure 4.17; execution *will* stop at your breakpoints. See Figure 4.21 for a depiction of the *echo* Web method, paused right before it returns the response back to the client. Notice, for example, the complicated call stack right, which gives you an idea of the heavy lifting that the .NET Framework does for you in order for Web Services to work properly.

Stop for a moment and consider what you have so far done in this section. Nothing prevents you from taking the Visual Basic script you just created and including it as client-side script in a traditional Web page (other than the fact that your clients will need to use Internet Explorer on a Windows platform, of course). If you do this, you have just created a Web Service client application that runs inside a browser window, making your *echo* service accessible to everybody who has an Internet connection and knows how to find your service.

So far, the only thing you have done is pass a string argument back and forth. The SOAP specification goes a lot further, as you can imagine; it defines a standard for passing a number of basic data types, complex data structures, and XML documents between a SOAP client and a SOAP server. You can also serialize objects and pass them over the wire. You will see examples of this in the section “Working with Web Services,” later on in this chapter.

## Figure 4.21 Stopping Your Application at a Breakpoint



You can find the complete code for the `echo` Web method on the Solutions Web Site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the directory `soapExamples/`.

### Testing a Web Service Using a Web Reference

Lastly, you can run and test a Web Service application by letting Visual Studio.NET create a .NET client proxy class for you, *automatically*. This proxy class contains one method for each Web method exposed by the Web Service. The tasks of creating the correct SOAP envelope, sending the data over the wire through HTTP, waiting for the response back from the server, and parsing the SOAP response envelope for the return value are all done for you. This may very well end up being your method of choice, because you don't need to worry about the details of the SOAP protocol, but can concentrate on solving the higher-level business problems at hand. However, to do this, you need to create a separate .NET client application, and then let Visual Studio.NET glue the two together by adding a reference to your Web Service server application. In order

to do this, however, we need to first talk about how Web Services can be described and discovered by potential clients.

## Developing & Deploying...

### Deploying Web Services

How do you deploy a Web Service, such as the *soapExamples* service you just created? The good news is that because Web Services are really just a special kind of an .NET Web application, that is they run under ASP.NET, deploying a Web Service is no different than deploying any other ASP.NET application: You simply create a new IIS virtual directory on the target server, copy all files from your project into the new location, and you're done. Before you do that, though, be sure to compile your Web Service with all debug information removed for better performance (see the section "Testing A Web Service Using Integrated Visual Studio.NET Debugging" earlier in this chapter for details).

However, in the real world, Web Services will likely often act as wrappers around legacy systems, such as database systems or enterprise applications. The difficulty, then, of deploying a Web Service will not be deploying the Web Service as such, but making sure that the Web Service works well together with those legacy systems.



## Working with Web Services

In this section, we want to showcase more examples of Web Services, and how the various standards work together. You can find the code of these examples on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the directory *soapExamples/*.

### Passing Complex Data Types

In this example, you will create a Web method that returns the arithmetic mean of a set of integer valued data points. You can call this method *arithmeticMean* and let it be part of the *simpleService* Web Service started at the beginning of this chapter.

The *arithmeticMean* method takes as argument an integer-valued array of data, called *arrayInput*, and returns a floating point value, as detailed in Figure 4.22. The code for Figure 4.22 is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 4.22** Web Method to Compute the Arithmetic Mean (simpleService.asmx.cs)

```

01: [SoapDocumentMethodAttribute(Action="arithmeticMean",
02:   RequestNamespace="urn:schemas-syngress-com-soap",
03:   RequestElementName="arithmeticMean",
04:   ResponseNamespace="urn:schemas-syngress-com-soap",
05:   ResponseElementName="arithmeticMeanResponse")]
06: [WebMethod(Description="Computes the " +
07:   "arithmetic means of an array of input parameters")]
08: public float arithmeticMean (int[] arrayInput) {
09:   if ((arrayInput == null) || (arrayInput.Length < 1)) {
10:     throw new Exception("No input data...");
11:   } else {
12:     int sum = 0;
13:     for(int i=0; i<arrayInput.Length; i++) {
14:       sum += arrayInput[i];
15:     }
16:     return (float)((float)sum / (float)arrayInput.Length);
17:   }
18: }

```

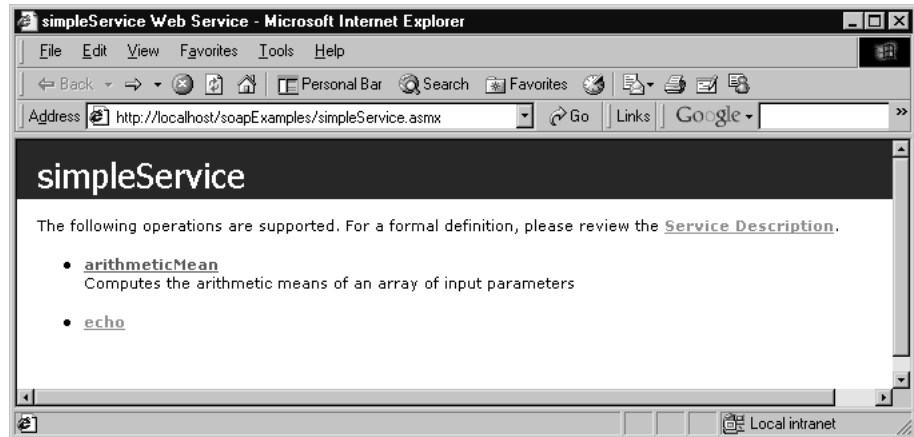
Note that you've added additional metadata to the method (see Figure 4.22):

- Specify that the *SOAPAction* HTTP header should be the method name, overriding the default, which is the method name, preceded by the namespace of the Web Service class (line 1).
- Specify the namespaces used by SOAP in requests to and responses from this Web method (lines 2 and 4). Namespaces specified at the Web method level overrule namespaces specified at the Web class level. Here, stick with the one you already defined on the class level.
- Set the XML element names used in the SOAP envelope to wrap the method data. As you have seen in the first example of the *echo* Web

method, and you don't change this here, by default the method name is used for SOAP requests (line 3), whereas the method name, appended with the string *Response*, is used for SOAP responses (line 5).

- Add a description of the Web method (lines 6 and 7). This shows up, for instance, on the Web Service overview page, as shown in Figure 4.23.

**Figure 4.23** Web Method Descriptions



You can start testing the new method by calling it using a simple HTTP *GET*. The individual array input elements are simply appended at the end of the URL—in the example, the numbers 1, 2, and 7:

```
http://localhost/soapExamples/simpleService.asmx/arithmeticMean?
arrayInput=1&arrayInput=2&arrayInput=7
```

You get the following result:

```
<?xml version="1.0" encoding="utf-8" ?>
<float xmlns="urn:schemas-syngress-com-soap">3.33333325</float>
```

The expected result for the arithmetic mean of 1, 2, and 7 is of course 3.33333333, and not 3.33333325, which shows that you should apparently be more careful when dealing with floating point arithmetic. Calling the method using SOAP, you can go to `http://localhost/soapExamples/simpleService.asmx?op=arithmeticMean` to figure out the correct syntax of the SOAP request envelope. You can then create a simple Visual Basic script similar to the one in Figure 4.17 (see the file `arithmeticMean.vbs` on the Solutions Web site for the book). In Figures 4.24 and Figure 4.25, you can see the SOAP-encoded data being exchanged during a client call to the *arithmeticMean* Web method.

**Figure 4.24** SOAP Request to *arithmeticMean*


---

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <arithmeticMean xmlns="urn:schemas-syngress-com-soap">
      <arrayInput>
        <int>1</int>
        <int>2</int>
        <int>7</int>
      </arrayInput>
    </arithmeticMean>
  </soap:Body>
</soap:Envelope>

```

---

**Figure 4.25** SOAP Response from *arithmeticMean*


---

```

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <arithmeticMeanResponse xmlns="urn:schemas-syngress-com-soap">
      <arithmeticMeanResult>3.33333325</arithmeticMeanResult>
    </arithmeticMeanResponse>
  </soap:Body>
</soap:Envelope>

```

---

## Error Handling

What happens if something goes wrong? An important part of debugging an application is realizing what can go wrong in the first place. In the case of Web Services, you may frequently encounter three kinds of errors. If you construct the

SOAP envelope by hand as opposed to using, say, Web References, your first stab at it will quite likely have some typos—this is the case of a malformed SOAP request. Another frequent error source is that some arguments passed to your Web method are not of the correct type. Finally, something can go wrong during execution of code on the server, and you will need to know how such a server exception is propagated back to the client, in order for you to take appropriate action. The following sections look at those three error scenarios in detail.

## Malformed SOAP Request

Call again the *arithmeticMean* Web method as you did earlier (see Figure 4.24). But this time, change the SOAP envelope in such a way that the XML is no longer valid XML (this shouldn't be too hard). Let's look what happens if you remove the start tag of the last *int* element, as shown in Figure 4.26, line 11.

**Figure 4.26** A Malformed SOAP Request

```
01: <?xml version="1.0" encoding="utf-8"?>
02: <soap:Envelope
03:   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
04:   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
05:   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
06:   <soap:Body>
07:     <arithmeticMean xmlns="urn:schemas-syngress-com-soap">
08:       <arrayInput>
09:         <int>1</int>
10:         <int>2</int>
11:           7</int>
12:       </arrayInput>
13:     </arithmeticMean>
14:   </soap:Body>
15: </soap:Envelope>
```

You then get a SOAP response that looks like the one shown in Figure 4.27.

**Figure 4.27** SOAP Response Indicating a Malformed Request

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
```

Continued



**Figure 4.27 Continued**


---

```

<soap:Body>
  <soap:Fault>
    <faultcode>soap:Client</faultcode>
    <faultstring>System.Web.Services.Protocols.SoapException:
      Server was unable to read request. ---&gt; System.Exception:
      There is an error in XML document (7, 21). ---&gt;
      System.Xml.XmlException: The 'arrayInput' start tag on line
      '5' does not match the end tag of 'int'. Line 8, position 16.
      at System.Xml.XmlTextReader.ParseTag()
      at System.Xml.XmlTextReader.ParseBeginTagExpandCharEntities()
      at System.Xml.XmlTextReader.Read()
      at System.Xml.XmlReader.Skip()
      at System.Xml.Serialization.XmlSerializationReader.
        UnknownNode(Object o)
      at n2499d7d93ffa468fbd8861780677ee41.XmlSerializationReader1.
        Read5_arithmeticMean()
      at System.Xml.Serialization.XmlSerializer.Deserialize
        (XmlReader xmlReader)
      at System.Web.Services.Protocols.SoapServerProtocol.
        ReadParameters()
      at System.Web.Services.Protocols.SoapServerProtocol.
        ReadParameters()
      at System.Web.Services.Protocols.WebServiceHandler.Invoke()
      at System.Web.Services.Protocols.WebServiceHandler.
        CoreProcessRequest()
    </faultstring>
  </detail/>
</soap:Fault>
</soap:Body>
</soap:Envelope>

```

---

What has happened is that the SOAP deserializer on the server noticed that the XML was not valid, threw an exception, and returned a *SOAP Fault*. A SOAP fault is what's returned to the client if an error occurred during program execution on the server. You can check programmatically for a SOAP Fault on the client in two ways:

- SOAP Faults return an HTTP error code 500 (Server error).
- SOAP Faults include the XML element `<Fault>` in the SOAP return envelope.

Inside the `<Fault>` element are four standard sections:

- `<faultcode>` Denotes if the error is a client or server error. In the example case of malformed XML, this is a client error. In fact, if you start the debugger in Visual Studio.NET and step through the code as you did in the earlier section on debugging using a client script, you will see that the *arithmeticMean* Web method is never even reached—program execution stops and control is returned to the client during the SOAP deserialization process, before the Web Service class is ever instantiated.
- `<faultstring>` Includes additional information about the error. By default, this contains the call stack at the time the error occurred.
- `<detail>` Where you as an application developer can put additional information about the error. Here, it is empty.
- `<faultactor>` An additional element defined by the SOAP specifications, but not returned by Microsoft .NET in this example.

## Wrong Argument Types

What if you try to pass a float argument to the Web method, that is, if your SOAP request contains the following element?

```
<int>1.1</int>
```

Similar to the malformed XML example earlier, a SOAP Fault is returned by the SOAP deserializer indicating a client fault. This is very powerful—it means that you will rarely have to worry about argument checking, because the .NET runtime environment will do this for you. (You still have to write code on the client to handle this situation appropriately, of course.)

## Exceptions in Server Code

Most often, exceptions will occur during program execution in your Web Service class and objects created by that class on the server. The *arithmeticMean* class, for instance, generates an exception whenever the argument array passed to it is empty.

In Figure 4.28, you see such a SOAP request to *arithmeticMean*: the *arrayInput* argument array containing the integers of which you want to compute the arithmetic mean is empty, and because you did not write your Web method in a robust way, a server error is returned (as shown in Figure 4.29).

### Figure 4.28 SOAP Request to *arithmeticMean*

---

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <arithmeticMean xmlns="urn:schemas-syngress-com-soap">
      <arrayInput/>
    </arithmeticMean>
  </soap:Body>
</soap:Envelope>
```

---

### Figure 4.29 SOAP request from arithmeticMean

---

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>System.Web.Services.Protocols.SoapException:
        Server was unable to process request.
        ---&gt; System.Exception: No input data...
        at soapExamples.simpleService.arithmeticMean
        (Int32[] arrayInput) in
        c:\inetpub\wwwroot\soapexamples\simpleService.asmx.cs:line 31
    </faultstring>
    <detail/>
  </soap:Fault>
</soap:Body>
</soap:Envelope>
```

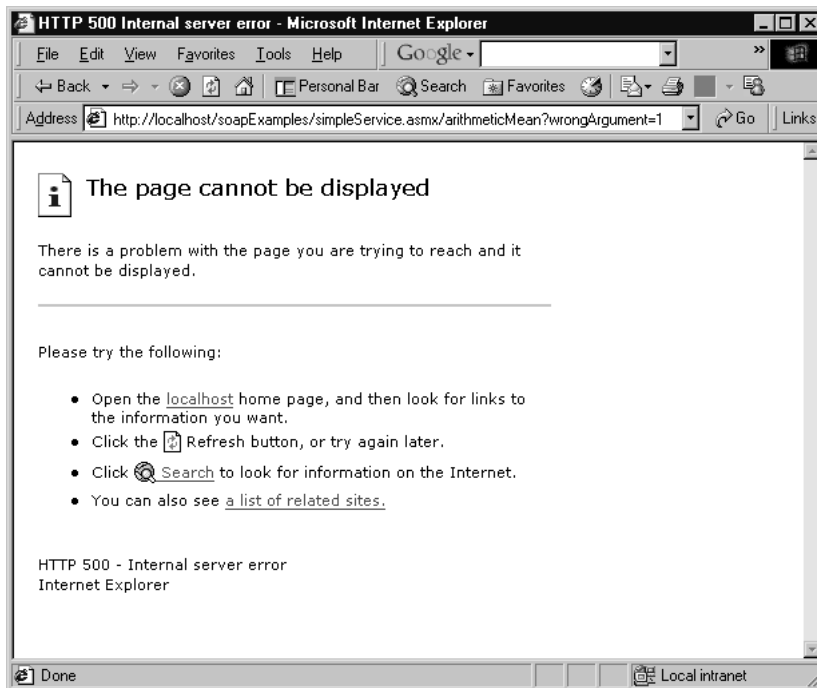
---

Again, notice how powerful Microsoft .NET is. In the Web class code (Figure 4.22, lines 9 and 10), you just threw a new *System.Exception*, with a custom error message (“No input data...”). .NET then did all the hard work and converted the system error into a SOAP Fault (see Figure 4.29) and even added the error message into the `<faultstring>` element, even though the formatting is maybe less than perfect. You also see that this time this is a server error (`<faultcode>soap:Server</faultcode>`), as expected.

It turns out that you have fine-grained control over SOAP Faults, and error handling in general.

Finally, note that if you call a Web method through a simple HTTP *GET* (or *POST*) request using a Web browser, depending on the exact request, all you may get could be the bleak browser error page (see Figure 4.30)—another reason to use SOAP from the very beginning!

**Figure 4.30** A Not Very Informative Error Page



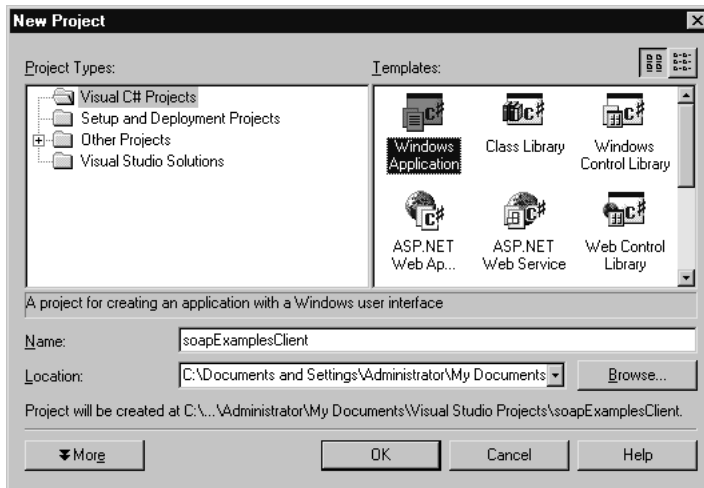
## Writing a SOAP Client Application

Maybe you're a little bit tired by now—manually writing Visual Basic scripts to test your Web Service—and would rather do some pointing and clicking. This is,

in fact, possible using Visual Studio.NET, although you lose some control over what's going on by going this route.

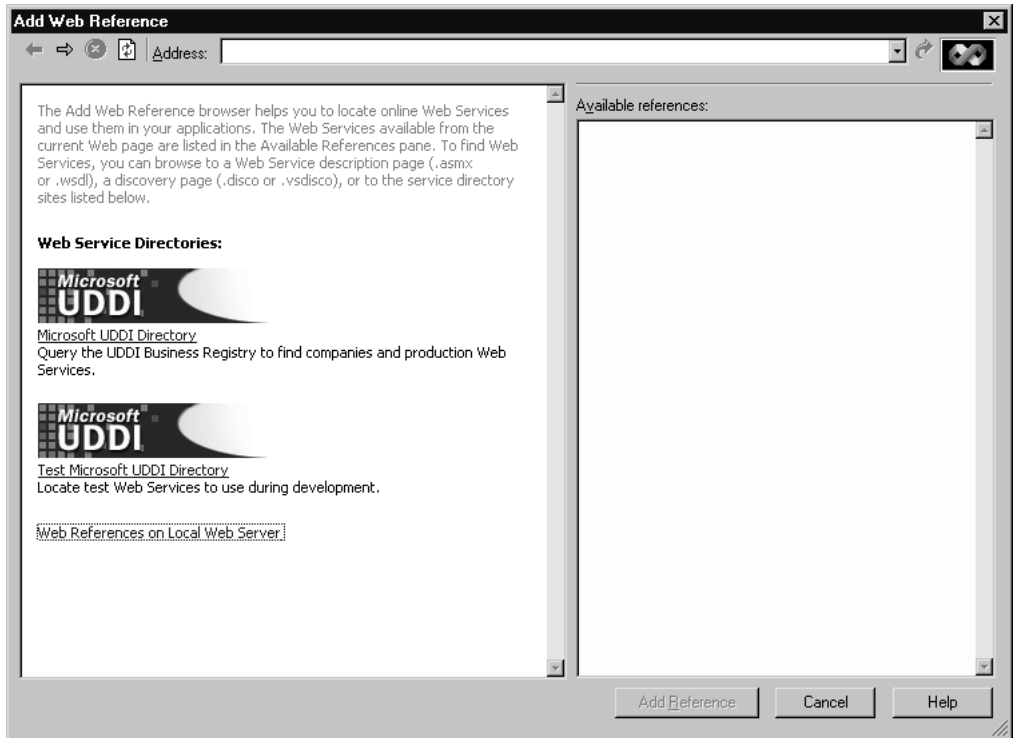
Let's then go ahead and create a Windows Forms–based client application for the *echo* Web method of the *simpleService* Web Service. Close the Visual Studio.NET solution you may be working on and create a new C# Windows application by selecting **File | New | Project**, choosing the entry **Windows Application** under the Visual C# Projects folder, and entering **soapExamplesClient** as the Name of the project as shown in Figure 4.31.

**Figure 4.31** Setting Up a New C# Windows Forms Application

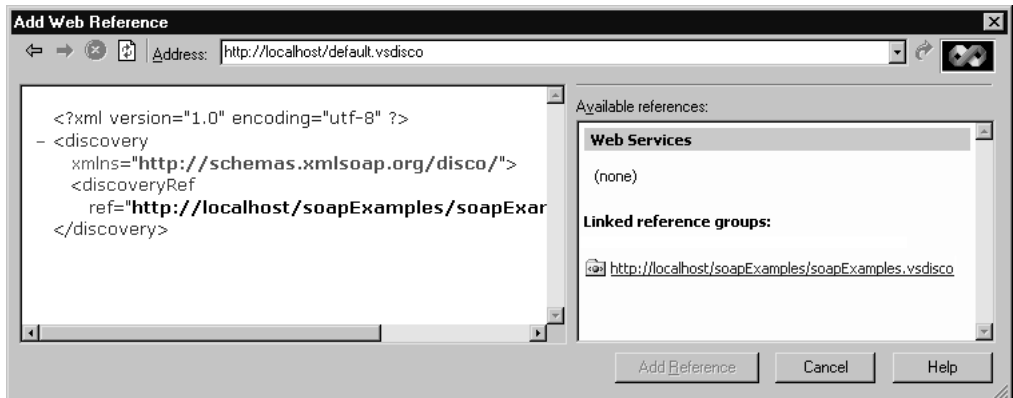


This will set up the necessary project files, and add a new Windows Form called `form1.cs`. Interestingly, Windows Forms applications do not separate design from code, and you will see references to form elements pop up in your C# code file, even though Visual Studio.NET goes through some efforts trying to “hide” those from you.

You need to teach the client to “know” about your Web Service. Go to the Solution Explorer, right-click the **soapExamplesClient** project, and select **Add Web Reference**. From here you could, for example, query a UDDI registry. Pretend that you didn't know what services are available on your machine, and use the DISCO discovering mechanism exposed under Web References On Local Web Server in the lower-left part of the Add Web Reference window (see Figure 4.32). For information on UDDI and DISCO, refer to Chapter 5 “WSDL and UDDI”.

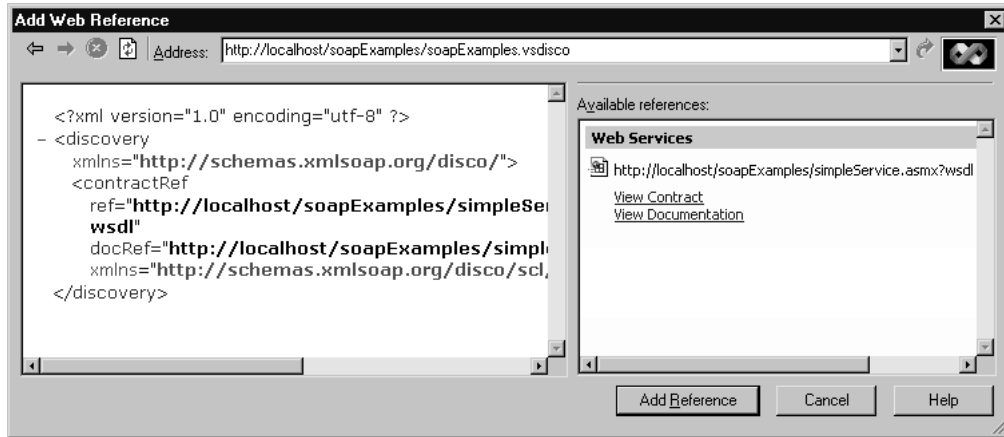
**Figure 4.32** The Add Web Reference Window

After a period of reflection, the DISCO file for your server will appear on the left panel, and the Web Service shows up as Linked Reference Group on the right panel (see Figure 4.33).

**Figure 4.33** Showing Available Linked Reference Groups through the DISCO Mechanism

Click on the DISCO file, and get to the next window (see Figure 4.34).

**Figure 4.34** Showing Available Web Services through the DISCO Mechanism



You can see the location of the corresponding WSDL file conveniently displayed both within the DISCO file on the left and the listing of Web Services on the right. You can now click **Add Reference** and let Visual Studio.NET contact the Web Service to gather all relevant data about this service through the WSDL mechanism.

Note that if DISCO fails you, as it has us a few times, just copy and paste the WSDL location (`http://localhost/soapExamples/simpleService.asmx?wsdl`) directly into the Address input box of the dialog, which is probably the preferred method anyway.

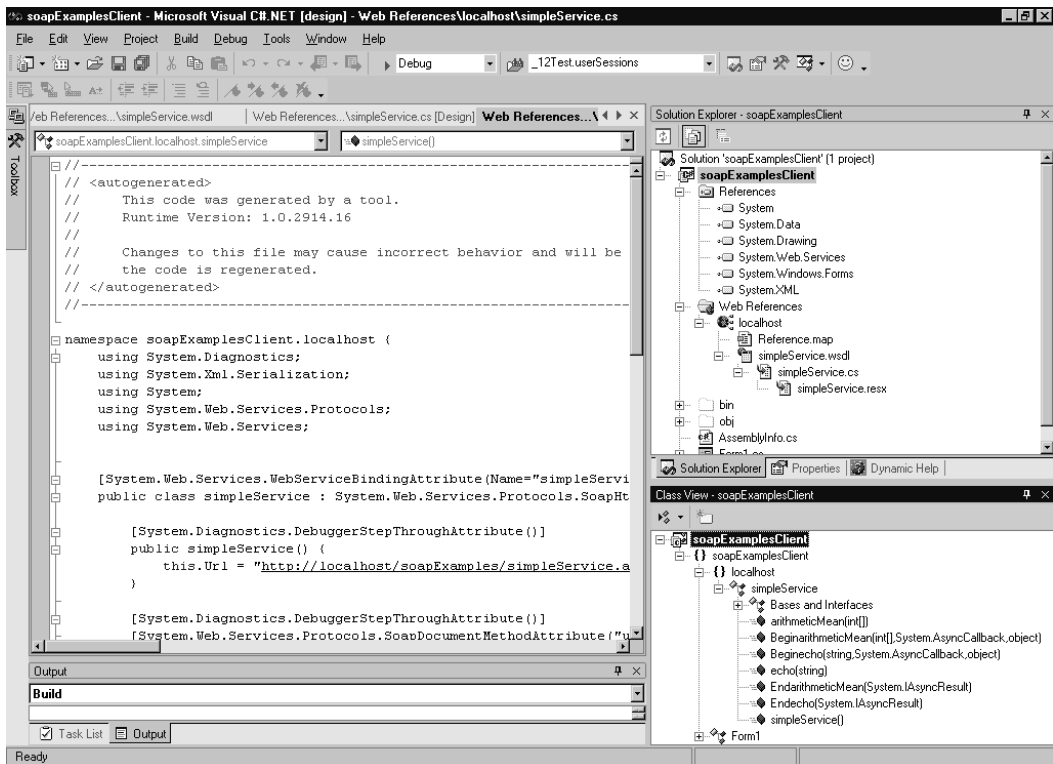
Let's see what Visual Studio.NET has done for you: go to the Solution Explorer, click the **Show All Files** icon to get into expert mode, expand all folders under the Web References folder, select **simpleService.cs**, and click on the **View Code** icon (Microsoft does not make this easy!). What you see is something like Figure 4.35.

What has happened? Visual Studio.NET has generated a *proxy* class for the *simpleService* Web class of the *soapExamples* Web Service. This proxy allows you to do a number of things:

- It has methods to call all methods your referenced Web Service exposes both through synchronous and asynchronous SOAP requests.
- All of the SOAP wire communication, including serializing and deserializing data, is done through the proxy, freeing you from a lot of manual coding.

- It allows you to work with remote Web Services the way you would with local objects, including full IntelliSense support.

**Figure 4.35** A Web Service Proxy



Concretely, it creates the *localhost.simpleService* class, which has the following public methods:

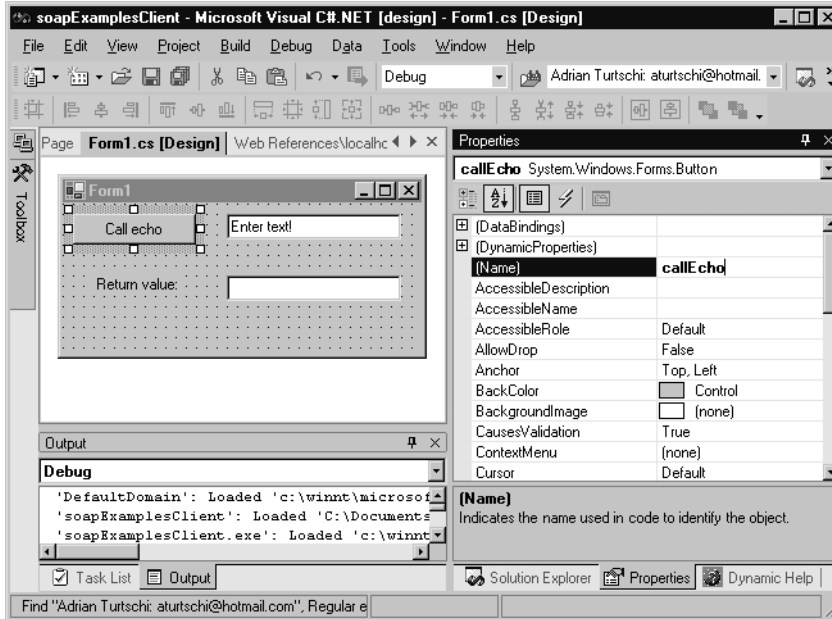
- *echo()* and *arithmeticMean()* to call the corresponding Web methods directly through issuing a (synchronous) SOAP request.
- *Beginecho()* and *BeginarithmeticMean()*, which call the corresponding Web methods through an asynchronous SOAP request. These methods have as an input parameter a reference to a *System.AsyncCallback* delegate which in turn references the callback method to be called when the asynchronous SOAP request has completed.
- Finally, *Endecho()* and *EndarithmeticMean()* are used to return the value of SOAP response after completion of an asynchronous SOAP request.



Note that *simpleService* inherits from the *System.Web.Services.Protocols.SoapHttpClientProtocol* class, where all the heavy lifting occurs to make SOAP calls possible.

So, let's design a form for the *echo* Web method, like the one shown in Figure 4.36.

**Figure 4.36** Creating a Web Service Client Form (Form1.cs of *soapExamplesClient*)



You need to add essentially two lines of code to call the *echo* Web method, as shown in Figure 4.37. The code for Figure 4.37 is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 4.37** Calling the *echo* Web Method (in Form1.cs of *soapExamplesClient*)

```
private void callEcho_Click(object sender, System.EventArgs e) {
    localhost.simpleService myWebSvc =
        new localhost.simpleService();
    try {
        this.soapReturnEcho.Text =
            myWebSvc.echo(this.enterText.Text);
    } catch (Exception ex) {
```

Continued

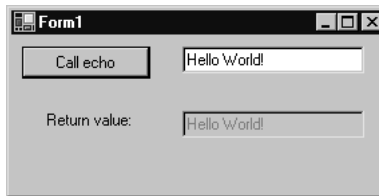
**Figure 4.37 Continued**

```

    // add error handling here...
}
}

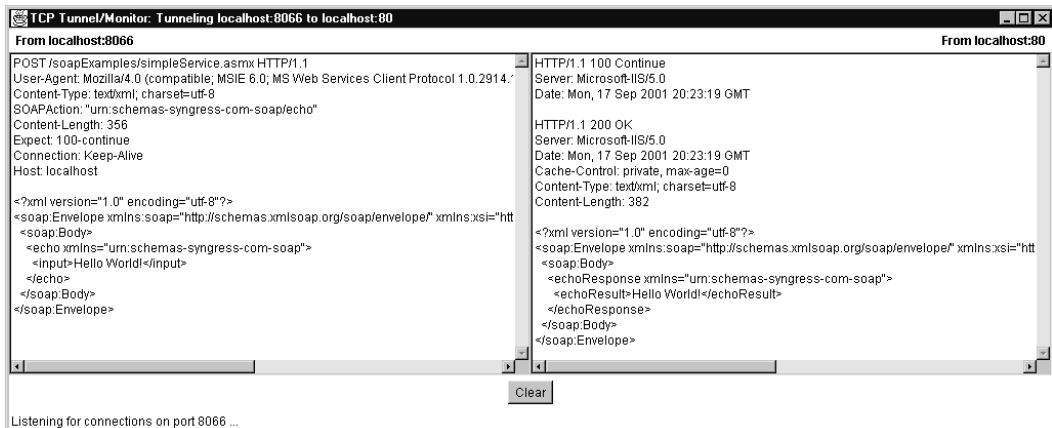
```

Let Microsoft .NET handle everything else. Running the application, if everything went well, will give you the picture shown in Figure 4.38.

**Figure 4.38 A Happy Web Service Client**

If you want to run this application outside Visual Studio.NET, you will find it at the following location: %USERPROFILE% \Visual Studio Projects\soapExamplesClient\bin\Debug\.

If you were to analyze HTTP traffic between your Web Service client and server applications using a network monitoring or network tunneling tool, you would see the exact same SOAP envelopes exchanged that you encountered in the earlier section “Testing a Web Service Using a Client Script.” For example, TcpTunnelGui, which is an excellent network tunneling tool that ships as part of the Apache SOAP implementation, nicely shows the SOAP exchange as depicted in Figure 4.39.

**Figure 4.39 Tunneling the echo Web Service to Inspect the SOAP Traffic**

You can find the complete code for this project on the Solutions Web Site for this book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the directory `soapExamplesClient/`.

## Passing Objects

The *SoapFormatter* class in the *System.Runtime.Serialization.Formatters.Soap* namespace is responsible for serializing and deserializing data according to the SOAP protocol. It is capable of sending and receiving whole objects, in addition to handling simple and complex data types, which you have already seen earlier in this chapter.

As an example, let's construct a simple Web Service that sends performance counter data to a Web client. The *System.Diagnostics* namespace contains the *PerformanceCounter* class, which is perfect for your purposes. You then simply write a Web method that takes as arguments the category, counter, and instance names necessary to instantiate a performance counter object, which you then send as a serialized object over SOAP to potential client applications. Note that valid argument values can be gathered from the Performance Monitor tool that's part of Windows 2000.

In Figure 4.40, you see the few lines of code needed to implement such a Web method. Simply add the code to your existing *soapExamples* project. These lines of code can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 4.40** *getCounterInfo* Web Method (simpleService.asmx.cs)

```
[SoapDocumentMethodAttribute(Action="getCounterInfo",
    RequestNamespace="urn:schemas-syngress-com-soap",
    RequestElementName="getCounterInfo",
    ResponseNamespace="urn:schemas-syngress-com-soap",
    ResponseElementName="getCounterInfoResponse")]
[WebMethod(Description="Returns performance counter information")]
public System.Diagnostics.PerformanceCounter getCounterInfo(
    string categoryName, string counterName, string instanceName) {

    System.Diagnostics.PerformanceCounter perfCounter
        = new System.Diagnostics.PerformanceCounter();

    perfCounter.CategoryName = categoryName;
    perfCounter.CounterName = counterName;
```

Continued

## Figure 4.40 Continued

---

```
perfCounter.InstanceName = instanceName;

if (perfCounter.CounterType < 0) {
    // counter is not a valid counter
    throw new Exception("Counter Data Invalid!");
}

return perfCounter ;
}
```

---

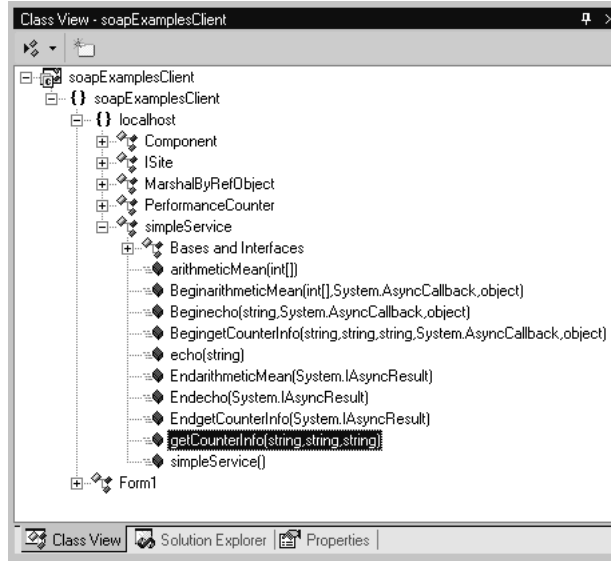
As shown, you initiate a new *PerformanceCounter* object using the argument data, check if you have a valid *PerformanceCounter*, and then simply return that object to the calling client. The .NET Framework will then do all the work for you, serializing the object through using a standard format.

If your Web Service client is itself a Microsoft .NET application, you are truly in luck, because the client can then receive the Web Service response as a *PerformanceCounter* object, and not as just an XML document containing SOAP data.

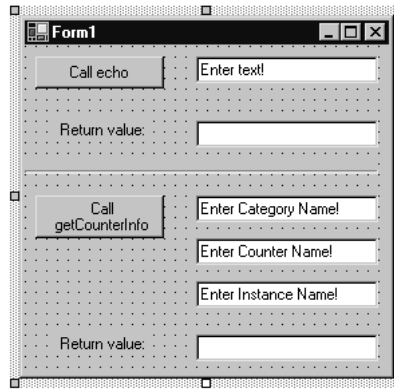
Here's how you need to modify the client code:

1. Open again your *soapExamplesClient* client application in Visual Studio.NET.
2. Right-click the **localhost** Web Reference in the Solution Explorer, and select **Update Web Reference**, which will add code to call the **getCounterInfo** Web method you just created to the client proxy (see Figure 4.41).
3. Change the Windows Form a little bit to accommodate the *getCounterInfo* Web method, as in Figure 4.42.
4. Add the necessary code to call the *getCounterInfo* Web method (see Figure 4.43).

**Figure 4.41** Proxy Code Added for New Performance Counter Web Method (*soapExamplesClient*)



**Figure 4.42** Adding Elements on the Windows Form for the *getCounterInfo* method (Form1.cs in *soapExamplesClient*)



**Figure 4.43** Calling the *getCounterInfo* Web Method (Form1.cs in *soapExamplesClient*)

```
private void callGetCounterInfo_Click(
    object sender, System.EventArgs e) {

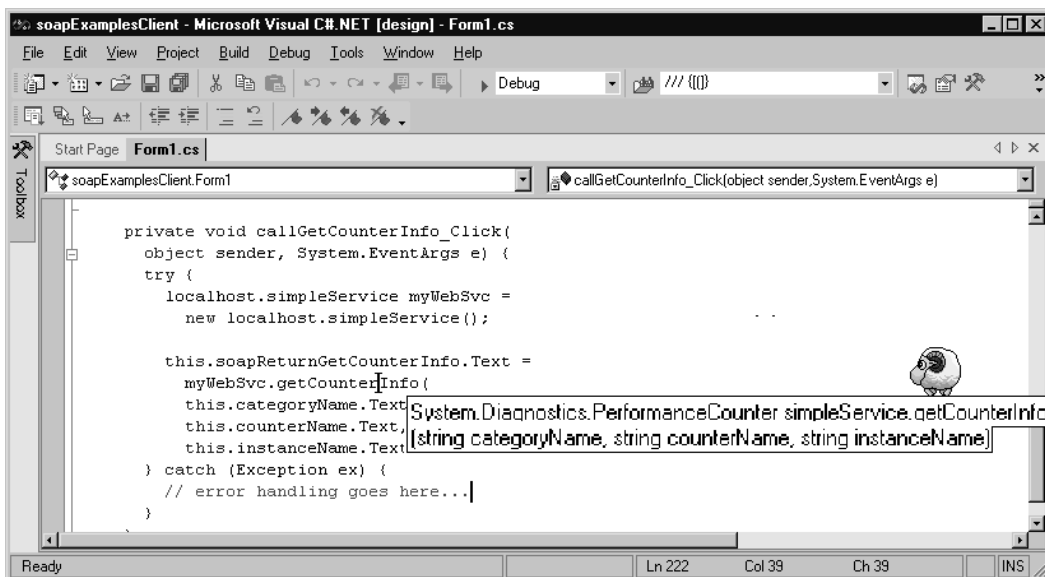
    localhost.simpleService myWebSvc =
```

Continued

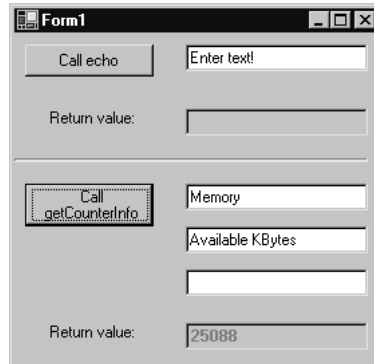
**Figure 4.43 Continued**

```
new localhost.simpleService();
try {
    this.soapReturnGetCounterInfo.Text =
        myWebSvc.getCounterInfo(
            this.categoryName.Text,
            this.counterName.Text,
            this.instanceName.Text).RawValue.ToString();
} catch (Exception ex) {
}
}
```

Note that the *getCounterInfo* Web method returns an object of type *PerformanceCounter*, as IntelliSense correctly tells us (see Figure 4.44).

**Figure 4.44 Microsoft's IntelliSense in Action**

5. After compiling the application, you are now able to expose, say, the size of available physical memory to the world, as depicted in Figure 4.45. Obviously, you should probably now secure this Web Service (see the “Security” section later in this chapter).

**Figure 4.45** Exposing Performance Information through a Web Service

If the Web Service client does not run on the Microsoft .NET platform, however, more work is needed. In this case, as a client application developer, you can either define a class matching the return type and extend the SOAP deserializer to handle that class type correctly, or as a last resort, you can always manually parse the SOAP return envelope for the data you are interested in.

To illustrate what's going on behind the scenes, let's look at the SOAP envelope passed back to the client in the Web Service response (see Figure 4.46).

**Figure 4.46** SOAP Response from *getCounterInfo* Passing Back Serialized Object Data

---

```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <getCounterInfoResponse xmlns="urn:schemas-syngress-com-soap">
      <getCounterInfoResult>
        <Site xsi:nil="true"/>
        <CategoryName>Memory</CategoryName>
        <CounterName>Available KBytes</CounterName>
        <RawValue>25080</RawValue>
      </getCounterInfoResult>
    </getCounterInfoResponse>
  </soap:Body>
</soap:Envelope>
```

---

As you see, the various properties of the *PerformanceCounter* class are serialized as XML elements, with their values being converted to a string format and added as text nodes. If you are sending your objects instantiated from your own classes, you can achieve finer control over how they are being serialized by using the *XmlAttributeAttribute* and *XmlElementAttribute* classes found in the *System.Xml.Serialization* namespace. In the same namespace, you also find classes that let you manipulate the XML namespaces used during the serialization process.

The opposite is also possible: If you already have an XML schema that you would like SOAP to use for data transfer, you can then take advantage of the XML Schema Definition Tool *xsd.exe*, found in `%ProgramFiles%\Microsoft.NET\FrameworkSDK\Bin\`, to generate the corresponding .NET classes to support that schema.

## Passing Relational Data (DataSets)

An interesting special case of passing objects over SOAP is passing back data coming from a relational database, such as *DataSets*. The .NET SOAP serializer, which is the piece of code that puts your data in XML format to be sent back inside a SOAP return envelope, can indeed serialize *DataSets* out of the box.

Let's have a look what happens under the hood, by writing a simple Web method that queries Microsoft's Northwind database for all data in the Shippers table and returns a serialized *DataSet* (note that you cannot serialize a *DataTable* using the default serializer). The code is in Figure 4.47, and also on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in directory `chapter1/rsTest/`.



**Figure 4.47** Code to Return a *DataSet* From The Northwind Database (`rsTest.asmx.cs`)

```
using System;
using System.ComponentModel;
using System.Data;
using System.Data.SqlClient;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;

namespace rsTest
{
    [WebServiceAttribute(Namespace="urn:schemas-syngress-com-soap" ) ]
```

Continued



**Figure 4.47 Continued**


---

```

public class rsTest : System.Web.Services.WebService
{
    public rsTest() {
    }

    [SoapDocumentMethodAttribute(Action="returnRS",
        RequestNamespace="urn:schemas-syngress-com-soap:rsTest",
        RequestElementName="returnRS",
        ResponseNamespace="urn:schemas-syngress-com-soap:rsTest",
        ResponseElementName="returnRSResponse")]
    [WebMethod]
    public DataSet returnRS() {
        try {
            string sqlConnectionString =
                "server=(local)\\NetSDK;database=Northwind;User ID=SA;Password=";
            SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(
                "SELECT * FROM shippers", sqlConnectionString);
            DataSet shippers = new DataSet();
            sqlDataAdapter.Fill(shippers, "shippers");
            return shippers;
        }
        catch (Exception e) {
            throw e;
        }
    }
}

```

---

When you now call the Web method *returnRS*, you get the SOAP envelope as in Figure 4.48, which looks complicated indeed! If you study the XML returned in detail, you will notice that the XML contains an XML Schema definition section for the *DataSet* returned, followed by the actual data, which consists of three shipping company records.

**Figure 4.48** SOAP Encoded *DataSet* Returned from Northwind Database

```

<?xml version="1.0" encoding="utf-8"?>
<DataSet xmlns="urn:schemas-syngress-com-soap">
  <xsd:schema id="NewDataSet" targetNamespace=""
    xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xsd:element name="NewDataSet" msdata:IsDataSet="true">
      <xsd:complexType>
        <xsd:choice maxOccurs="unbounded">
          <xsd:element name="shippers">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="ShipperID"
                  type="xsd:int" minOccurs="0" />
                <xsd:element name="CompanyName"
                  type="xsd:string" minOccurs="0" />
                <xsd:element name="Phone"
                  type="xsd:string" minOccurs="0" />
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
  <diffgr:diffgram
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
    <NewDataSet xmlns="">
      <shippers diffgr:id="shippers1" msdata:rowOrder="0">
        <ShipperID>1</ShipperID>
        <CompanyName>Speedy Express</CompanyName>
        <Phone>(503) 555-9831</Phone>
      </shippers>
      <shippers diffgr:id="shippers2" msdata:rowOrder="1">
        <ShipperID>2</ShipperID>

```

Continued

**Figure 4.48 Continued**


---

```

    <CompanyName>United Package</CompanyName>
    <Phone>(503) 555-3199</Phone>
  </shippers>
  <shippers diffgr:id="shippers3" msdata:rowOrder="2">
    <ShipperID>3</ShipperID>
    <CompanyName>Federal Shipping</CompanyName>
    <Phone>(503) 555-9931</Phone>
  </shippers>
</NewDataSet>
</diffgr:diffgram>
</DataSet>

```

---

If your client is running Microsoft .NET software, you're in luck: The client will automatically reassemble the SOAP response into a *DataSet* that you can then use to continue processing. However, there are potential (business!) clients on the Internet who do not and never will run on a Microsoft platform. For those, the XML in Figure 4.48 is hard to parse. Theoretically, this should be possible, because the XML does contain the XML Schema definition needed to understand and reassemble the data, but in practice, few people would want to deal with such a monstrosity.

Our advice, then, is to shy away from passing data coming from a database as Microsoft *DataSets*, unless you really, really know that the only clients ever to consume your Web Services will be Microsoft clients, running, preferably, on the .NET platform.

## Passing XML Documents

So far we have focused on using Web Services as an RPC (remote procedure call) mechanism. Although the data being exchanged through SOAP has of course been in the form of XML documents all along, it was the data being exchanged and not the XML document as such that we were interested in so far.

There are cases, however, when you will just want to exchange XML documents between a client and a server; these XML documents could be invoices, tagged magazine articles, your own custom data encoding scheme, and so on. Often, these XML documents being exchanged will have an associated schema against which they will be validated.

The example shown in Figure 4.49 is a simple service that accepts an XML document and returns the same XML document, adding only an XML attribute *dateProcessed* to the XML root element, indicating when the XML was processed. It is part of the *simpleService* Web Service. The example in Figure 4.49 can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 4.49** *xmlTester* Web Method (simpleService.asmx.cs)

```
01: [SoapDocumentMethodAttribute(Action="xmlTester",
02:     RequestNamespace="urn:schemas-syngress-com-soap",
03:     ResponseNamespace="urn:schemas-syngress-com-soap",
04:     ParameterStyle = SoapParameterStyle.Bare)]
05: [WebMethod(Description="XML echo service that " +
06:     "adds a dateProcessed attribute.")]
07: [return: XmlAnyElement]
08: public XmlElement xmlTester(
09:     [XmlAnyElement]XmlElement inputXML){
10:
11:     inputXML.SetAttribute("dateProcessed",
12:         System.DateTime.Now.ToUniversalTime().ToString("r"));
13:     return inputXML;
14: }
```

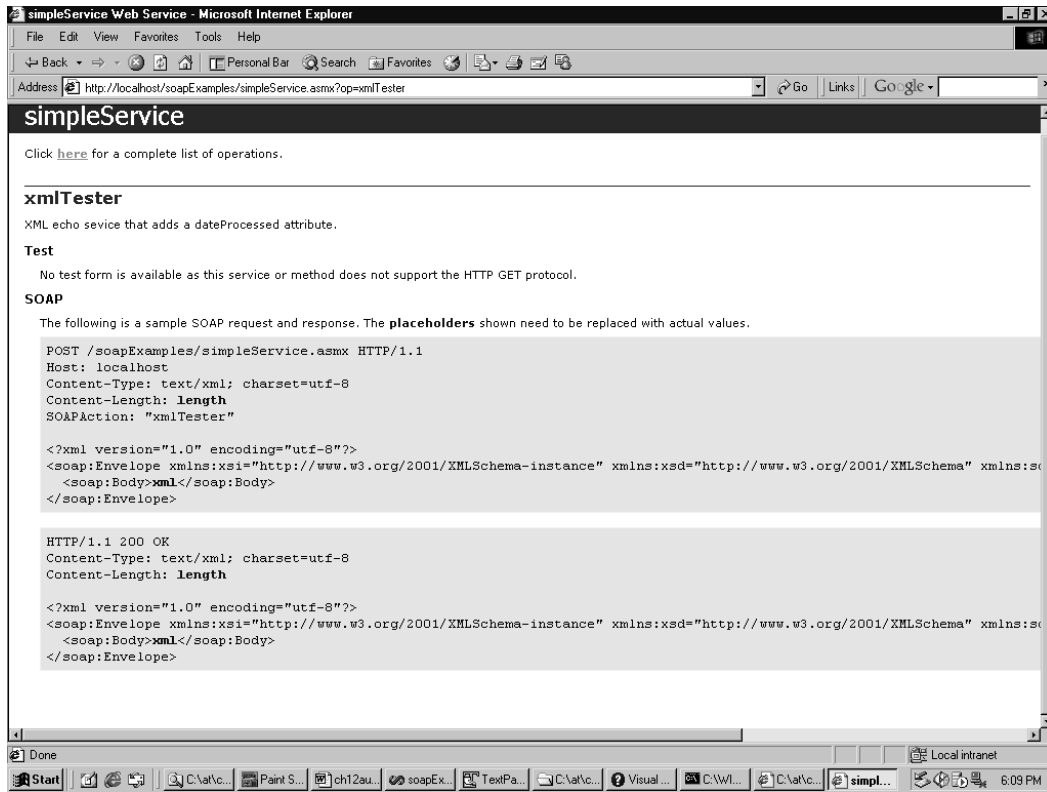
Note you've added the instruction

```
ParameterStyle = SoapParameterStyle.Bare
```

to the *SoapDocumentMethodAttribute* section (Figure 4.49, line 4), specifying that the XML document that is the argument for the *xmlTester* Web method should appear directly beneath the *Body* element of the SOAP request envelope, and that you don't want an intermediate XML element in the SOAP response either.

When you run *xmlTester* through Visual Studio.NET, you will see that this Web method can be called only through SOAP (see Figure 4.50), which makes sense because you can't pass an XML document through a simple HTTP *GET* or HTTP *POST*.

You can test this service by writing a Visual Basic script similar to the ones you created earlier in this chapter (see Figure 4.51). When running this script, you can observe the SOAP data exchange taking place as shown in Figures 4.52 and 4.53. Note the additional attribute *dateProcessed* in Figure 4.53, shown in bold, that was added through the Web *xmlTester* method.

Figure 4.50 The Overview Page For The *xmlTester* Web MethodFigure 4.51 VBS Script to Test the *xmlTester* Web Method (xmlTester.vbs)

```

myWebService = "http://localhost/soapExamples/simpleService.asmx"
myMethod = "xmlTester"

' ** create the SOAP envelope with the request
s = ""
s = s & "<?xml version=""1.0"" encoding=""utf-8""?>" & vbCrLf
s = s & "<soap:Envelope "
s = s & "   xmlns:xsi=""http://www.w3.org/2001/XMLSchema-instance"" "
s = s & "   xmlns:xsd=""http://www.w3.org/2001/XMLSchema"" "
s = s & "   xmlns:soap=""http://schemas.xmlsoap.org/soap/envelope/">"
s = s & vbCrLf
s = s & "   <soap:Body>" & vbCrLf
s = s & "       <rootElement>" & vbCrLf
s = s & "           <someNode someAttribute=""random"">" & vbCrLf

```

Continued

**Figure 4.51 Continued**


---

```

s = s & "          <someOtherNode>some data</someOtherNode>" & vbCrLf
s = s & "          </someNode>" & vbCrLf
s = s & "          </rootElement>" & vbCrLf
s = s & " </soap:Body>" & vbCrLf
s = s & "</soap:Envelope>" & vbCrLf

msgbox(s)

set requestHTTP = CreateObject("Microsoft.XMLHTTP")

msgbox("xmlhttp object created")

requestHTTP.open "POST", myWebService, false
requestHTTP.setRequestHeader "Content-Type", "text/xml"
requestHTTP.setRequestHeader "SOAPAction", myMethod
requestHTTP.Send s

msgbox("request sent")

set responseDocument = requestHTTP.responseXML

msgbox("http return status code: " & requestHTTP.status)
msgbox(responseDocument.xml)

```

---

**Figure 4.52 SOAP Request to *xmlTester* Web Method**


---

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <rootElement>
      <someNode someAttribute="random">
        <someOtherNode>some data</someOtherNode>

```

---

Continued

**Figure 4.52 Continued**


---

```

    </someNode>
  </rootElement>
</soap:Body>
</soap:Envelope>

```

---

**Figure 4.53 SOAP Response from *xmlTester* Web Method**


---

```

<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <rootElement dateProcessed="Tue, 18 Sep 2001 22:15:55 GMT">
      <someNode someAttribute="random">
        <someOtherNode>some data</someOtherNode>
      </someNode>
    </rootElement>
  </soap:Body>
</soap:Envelope>

```

---

Obviously, this is only the very tip of the iceberg. The ability to send generic XML documents back and forth is a powerful feature of SOAP. In passing, we mention that a related standard called *SOAP Messages With Attachments* ([www.w3.org/TR/SOAP-attachments](http://www.w3.org/TR/SOAP-attachments)) defines a way to pass generic files (binary or text) using SOAP as MIME-encoded attachments.

## SOAP Headers

Similar to the way the HTTP protocol has a header section that contains general information about the request and a body section that contains specific application data relevant to the request, the SOAP protocol specifies that the SOAP envelope has both a header and a body section. So far, you have only seen examples of SOAP requests (and responses) that had *Body* elements, but no *Header* elements. That's because a SOAP *Body* element is required, whereas a SOAP *Header* element is not. In fact, SOAP headers were designed to give SOAP an extension mechanism.

The SOAP *Header* element appears right underneath the SOAP *Envelope* element, and you're free to define your header name and header value, and what it means to have such a SOAP header present. As an example, you could encode transaction information in a SOAP header. In the "Maintaining State" section to follow, we show you a possible usage of SOAP headers as a mechanism to establish a notion of a client session, and we discuss what classes in the .NET Framework you have to use to handle SOAP headers.

## Advanced Web Services

Web Services were designed to be, above all, simple—simple to implement, and simple to use. Simplicity has its price, however, and there are a variety of features that you won't find in Web Services—features that are part of older, more established data exchange protocols, such as COM/DCOM or CORBA. Such features include state management, security, and transaction processing.

You need to realize that programming on the Internet is different than programming on a private network. Expecting the two to be the same would be wrong. You don't have the same level of control on the Internet that you have on a local area network, and it is clear that data communication on the Internet will mean having less direct control, and allowing for more things to go wrong. You should therefore not expect to be able to implement a complex real-time transactional system involving ten transaction partners using SOAP—at least not today.

Let's look at two problem areas you are likely to encounter when developing real-world Web Services. First, the question of whether to maintain state or not, and if yes, how, and secondly how to handle security.

## Maintaining State

Our suggestion is to not try to introduce state in Web Service applications, at least for the time being. If you consider where state has traditionally been introduced in Web applications, the most prominent area is probably in e-commerce with the usage of so-called shopping carts. Clearly, you should not try to write a Web Service shopping cart application. Another area is security. We discuss security later in the chapter, but good alternatives exist to having explicitly stateful applications in that area as well. In all other areas, introducing state is almost always a bad idea. Considering that Web Services were designed to let distributed systems talk to each other in a *loosely coupled* way, state just doesn't seem to fit the picture quite right from an architectural point of view. Still, you have a variety of options to add state, which we discuss next.



Let's first briefly review the options you have in architecting stateful Web applications.

HTTP, the protocol underlying Web applications, is an inherently stateless protocol. A client issues a request against a server, which in turn issues a response. Every client request is seen by the server as a new request, not connected to any previous request. Technically, the client issues an HTTP request by opening a TCP/IP socket connection to the server, issues a request using the HTTP protocol, gets some data from the server, and then closes the socket. The next HTTP request will be issued using a new TCP/IP socket connection, making it impossible for the server to understand, on the protocol level, that the second request may really be the continuation of the first request. Note that the keep-alive function in HTTP does not change this picture, because it is geared mainly towards making the retrieval of Web pages containing many individual page elements more efficient, but it does not guarantee in any way that a network connection to the server is maintained over any longer period of time.

Introducing state means that you add logic on the server to be able to relate a previous request from a particular client to a subsequent request from the same client. This is being done by introducing information that identifies a particular client to the HTTP request and response data, and developing *application level* code that makes sense of that additional data. Saying that a client establishes a *session* with a server just means that you have application logic that connects several client requests to a logical session using that additional information, even though, because of the nature of the HTTP protocol, there is no physical equivalent to a session (i.e., no ongoing network connection over the lifetime of a client-server interaction).

Looking at the HTTP protocol, there are three places where you may add state information identifying a client:

- The URL against which the request is issued (the first line in an HTTP request)
- The header part of an HTTP request (including cookies)
- The body part of an HTTP request

And the two latter possibilities hold for HTTP responses as well.

We look at some examples in the following sections. You can find the code about maintaining state in the directory `sessionTest/` on the Solutions Web site ([www.syngress.com/solutions](http://www.syngress.com/solutions)) for the book.

## State Information in the URL (URL Mangling)

You can maintain state information by adding a unique client session identifier to the URL. Microsoft's Passport service uses this method to assign and maintain client session authentication information. ASP.NET natively supports this method through a configuration entry in the `config.web` file. The advantage of this method is that it is very scalable, supports Web farms and Web gardens, can be configured to survive IIS restarts without losing session information, and that you have the option of saving client information on an external SQL Server database. Technically, what happens is that a Web application that is configured to map state information to URLs will redirect a new incoming client request using an HTTP 302 status code (Found) to a new URL that contains a session identifier. Here's how it works:

1. Set the **cookieless** attribute of the **session** element in the `web.config` ASP.NET configuration file to **True**.
2. Create a new Web method with an attribute **EnableSession** set to **True**, and use the `System.Web.HttpContext.Current.Session` object (or `Web.Service.Session`, which amounts to the same object):

```
[WebMethod(EnableSession=true)]
public string sessionTest__URL() {
    if (Session["HitCounter"] == null) {
        Session["HitCounter"] = 1;
    } else {
        Session["HitCounter"] = ((int) Session["HitCounter"]) + 1;
    }
    return (Session["HitCounter"].ToString());
}
```

Let's look what happens on the HTTP protocol level if a client calls this method twice. You can look at the HTTP data exchange by using a TCP tunneling tool. Here we have used `TcpTunnelGui`, which ships as part of the Apache Project's SOAP implementation, but you can, of course, easily write your own TCP tunnel program using the .NET Framework (do it—it's a great exercise!).

You can call the Web Service through a simple HTTP `GET` request (we ignore some of the irrelevant HTTP headers). In the first call, the client issues an HTTP `GET`:

```
GET /sessionTest/sessionTest.asmx/sessionTest__URL HTTP/1.1
```

```
Host: localhost
Connection: Keep-Alive
```

Server issues an HTTP 302 (Moved) to a URL that contains the session identifier:

```
HTTP/1.1 302 Found
Server: Microsoft-IIS/5.0
Date: Wed, 12 Sep 2001 22:14:21 GMT
Location: /sessionTest/(bf33go2yvicwfhbragscdwvu)/
    sessionTest.asmx/sessionTest__URL
Cache-Control: private
Content-Type: text/html; charset=utf-8
Content-Length: 176

<html><head><title>Object moved</title></head><body>
<h2>Object moved to
<a href='/sessionTest/(bf33go2yvicwfhbragscdwvu)/
    sessionTest.asmx/sessionTest__URL'>
here</a>.</h2></body></html>
```

Client reissues an HTTP *GET* for the new URL:

```
GET /sessionTest/(bf33go2yvicwfhbragscdwvu)/
    sessionTest.asmx/sessionTest__URL HTTP/1.1
Host: localhost
Connection: Keep-Alive
```

Server send back the SOAP response:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Wed, 12 Sep 2001 22:14:21 GMT
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 96

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="urn:schemas-syngress-com-soap">1</string>
```

In the second call, the client issues an HTTP *GET* (using the modified URL):

```
GET /sessionTest/(bf33go2yvicwfhbragscdwvu)/
  sessionTest.asmx/sessionTest__URL HTTP/1.1
Host: localhost
Connection: Keep-Alive
```

The server responds, incrementing the session hit counter:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Wed, 12 Sep 2001 22:14:30 GMT
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 96

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="urn:schemas-syngress-com-soap">2</string>
```

So far, so good. The problem with implementing session state for Web Services this way is that you need to teach your Web Service client application two things:

- They need to follow HTTP 302 messages.
- When issuing a follow-up request, they should either use relative URLs, or they should remember changed URLs through HTTP 302 codes.

Both constraints are hard to implement, and somewhat contrary to the underpinnings of the Web Services philosophy. Basically, you require your Web Service clients to be very smart, as smart, indeed, as a Web browser is. None of the current Web Service clients is currently capable of supporting this functionality, and that includes the .NET Web Service proxy.

## State Information in the Http Header (Cookies)

You can add state information in additional HTTP headers. This is used in two common scenarios:

- **Authentication** The various authentication schemes, such as Basic Authentication, Windows NTLM-based authentication, Kerberos-based authentication, and others, work by sending an additional Authentication header element between client and server. Typically, the client sends credential information to the server, which then verifies the information

received, may ask for additional information, and finally answers by returning a session key (which is still sent in the Authentication header field), that is then used by all subsequent client requests to access protected server resources.

- **Cookies** Cookies are pieces of data that are persisted on the client computer. They are stored and received using an additional HTTP header element called *Cookie*.

ASP.NET has improved session handling using cookies; similarly to the “cookieless” session management explained in the preceding section, it now supports cookie-based sessions that scale well, support Web farms and Web gardens, and it can save client information away in a remote database out-of-the-box.

Let’s look at an example using cookies to store state information:

1. Set the **cookieless** attribute of the **session** element in the web.config ASP.NET configuration file to **False**.
2. Create a new Web method with an attribute **EnableSession** set to **True**, and use the *System.Web.HttpContext.Current.Session* object (or use the *Web.Service.Session* object):

```
[WebMethod(EnableSession=true)]
public string sessionTest__httpHeader() {
    if (Session["HitCounter"] == null) {
        Session["HitCounter"] = 1;
    } else {
        Session["HitCounter"] = ((int) Session["HitCounter"]) + 1;
    }
    return (Session["HitCounter"].ToString());
}
```

Let’s look what happens on the HTTP protocol level if a client calls this method twice. You can call the Web Service through a simple HTTP GET request (we ignore some of the irrelevant HTTP headers). In the first call, the client issues an HTTP *GET*:

```
GET /sessionTest/sessionTest.asmx/sessionTest__httpHeader HTTP/1.1
Host: localhost
Connection: Keep-Alive
```

The server sends back the SOAP response, including a *Cookie* header requesting the client to set a session cookie:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 13 Sep 2001 17:58:09 GMT
Transfer-Encoding: chunked
Set-Cookie: ASP.NET_SessionId=znbmfmqcufov4p45s204wp45; path=/
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="urn:schemas-syngress-com-soap">1</string>
```

In the second call, the client issues an HTTP *GET*, and sends the session *Cookie* header received from the server in the previous call:

```
GET /sessionTest/sessionTest.asmx/sessionTest__httpHeader HTTP/1.1
Host: localhost
Connection: Keep-Alive
Cookie: ASP.NET_SessionId=znbmfmqcufov4p45s204wp45
```

The server responds, incrementing the session hit counter (the *Cookie* header is not sent again, because the server retrieved the *Cookie* header in the HTTP request from the client, so it knows that the client honored its cookie request from the first response):

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/5.0
Date: Thu, 13 Sep 2001 17:58:20 GMT
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 96

<?xml version="1.0" encoding="utf-8"?>
<string xmlns="urn:schemas-syngress-com-soap">2</string>
```

However, if you want to encode session state information into cookies, you need to insist that all your Web Service clients are capable of handling cookies correctly. Only very few potential consumers will probably be willing to add that

functionality to their client applications because, again, cookies really belong into the domain of Web browsers, and seem strange in a Web Service client application.

On the other hand, you could certainly add session state information in a custom HTTP header (maybe called *webState?*). This would require manually adding code to both the Web Service server to clients to correctly handle that additional header element. Even worse, WSDL, the Web Service description format, has no provisions to support such new, required HTTP headers.

## State Information in the Http Body (SOAP Header)

The last possibility, finally, is to embed state information into the HTTP body itself. This method really only makes sense if you use SOAP to call your Web Service (as opposed to issuing simple HTTP *GET* or *POST* requests).

SOAP indeed does have the option of adding custom SOAP headers into the SOAP envelope. Note that a SOAP header is *not* the same as an HTTP header; it is a header relative to the SOAP message, that is it appears within the HTTP body, inside the SOAP envelope.

There is currently no support for keeping client state information in SOAP headers in ASP.NET, so you need to do everything yourself.

Let's try then to re-create a simple hit counter using SOAP headers. You need to implement the following:

- Name your SOAP header element: call it *webState*.
- Create a class that can handle your SOAP header on the server
- Create a class on the server that records and maintains all client sessions, using a static hash table.

Let's look at the server code (see Figure 4.54).

**Figure 4.54** Implementing a Hit Counter Using SOAP Headers

---

```

01: using System;
02: using System.Collections;
03: using System.ComponentModel;
04: using System.Data;
05: using System.Diagnostics;
06: using System.Web;
07: using System.Web.Services;
08: using System.Web.Services.Protocols;
09: using System.Runtime.InteropServices;

```

---

Continued

**Figure 4.54 Continued**

```
10:
11: namespace sessionTest {
12:     [WebServiceAttribute(
13:         Namespace="urn:schemas-syngress-com-soap")]
14:     public class sessionTest : System.Web.Services.WebService {
15:         public sessionTest() {
16:         }
17:
18:         protected override void Dispose( bool disposing ) {
19:         }
20:
21:         public class soapHeader : SoapHeader {
22:             public string webState;
23:         }
24:
25:         public soapHeader mySoapHeader;
26:         public static Hashtable userSessions = new Hashtable();
27:
28:         [SoapDocumentMethodAttribute(Action="sessionTest__soapHeader",
29:             RequestNamespace=
30:                 "urn:schemas-syngress-com-soap:sessionTestst",
31:             RequestElementName="sessionTest__soapHeader",
32:             ResponseNamespace=
33:                 "urn:schemas-syngress-com-soap:sessionTestst",
34:             ResponseElementName="sessionTest__soapHeaderResponse")]
35:         [SoapHeader("mySoapHeader",Direction=SoapHeaderDirection.InOut,
36:             Required=true)]
37:         [WebMethod]
38:         public string sessionTest__soapHeader() {
39:             // declare user session hit counter
40:             int hitCounter;
41:             // declare session identifier
42:             string sessionID;
43:
44:             if ((mySoapHeader.webState == null) ||
```

Continued



**Figure 4.54 Continued**


---

```

45:         (mySoapHeader.webState.Trim().Length < 1)){
46:             // create a new random session identifier
47:             sessionID = System.Guid.NewGuid().ToString().ToUpper();
48:             hitCounter = 1;
49:             // create a new user session, and set hit counter to one
50:             userSessions.Add(sessionID, hitCounter);
51:             // return session identifier to user
52:             mySoapHeader.webState = sessionID;
53:         } else {
54:             // valid user session?
55:             sessionID = mySoapHeader.webState.ToString().Trim();
56:             if(userSessions[sessionID] != null) {
57:                 // get session hit counter
58:                 hitCounter = (int)userSessions[sessionID];
59:                 // save away incremented session hit counter
60:                 userSessions[sessionID] = ++hitCounter;
61:             } else {
62:                 // session identifier passed was invalid
63:                 // throw error
64:                 throw new Exception("Invalid session identifier passed!");
65:             }
66:         }
67:         // return session counter
68:         return hitCounter.ToString();
69:     }
70:
71: }
72: }

```

---

Note the following important elements in the code shown in Figure 4.54:

- It includes a class *soapHeader* (line 21–23), which extends *System.Web.Services.Protocols.SoapHeader*, with a public string variable called *webState* (line 22), which is the SOAP header that should contain your client state identifier. The code calls the corresponding Web Service class instance variable *mySoapHeader* (line 25).

- The code includes a static hash table called *userSessions*, which will contain the collection of all client sessions (line 26).
- It includes the Web method *sessionTest\_\_soapHeader* (line 38) with the attribute *SoapHeader*, (lines 35–36), where you specify that you *require* the *webState* SOAP header, and that this SOAP header is bidirectional. This means that if a client does not send you this SOAP header, the .NET Framework will send a SOAP fault to the client, and you don't need to code for that possibility yourself.
- Because you want to tell your clients what session identifier to use in subsequent requests, you return the new session identifier in the same *webState* SOAP header (line 68).

On the client side, because you require the presence of the *webState* SOAP header, you need to initialize this header before issuing the SOAP request. That is, if you write a client using Web references, your call to the *sessionTest\_\_soapHeader* Web method will look like this:

```
testClient.localhost.sessionTest myClient =
    new sessionTestClient.localhost.sessionTest();
myClient.soapHeaderValue = new testClient.localhost.soapHeader();
string result = myClient.sessionTest__soapHeader();
```

The following code is a sample client server interaction using the SOAP protocol (ignoring HTTP headers). In the first call, the client issues an SOAP request, leaving the *webState* SOAP header empty:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <soapHeader xmlns="urn:schemas-syngress-com-soap">
      <webState></webState>
    </soapHeader>
  </soap:Header>
  <soap:Body>
    <sessionTest__soapHeader
      xmlns="urn:schemas-syngress-com-soap:sessionTest">
    </sessionTest__soapHeader>
```

```

</soap:Body>
</soap:Envelope>

```

The server sends back the SOAP response, including the *webState* SOAP header element with the new session identifier:

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Header>
    <soapHeader xmlns="urn:schemas-syngress-com-soap">
      <webState>{45D345B6-BE1F-434F-BFD7-D628C756A432}</webState>
    </soapHeader>
  </soap:Header>
  <soap:Body>
    <sessionTest__soapHeaderResponse
      xmlns="urn:schemas-syngress-com-soap:sessionTestst">
      <sessionTest__soapHeaderResult>1</sessionTest__soapHeaderResult>
    </sessionTest__soapHeaderResponse>
  </soap:Body>
</soap:Envelope>

```

In the second call, the client issues another SOAP request, and sends the session identifier in the *webState* SOAP header received from the server in the previous response:

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Header>
    <soapHeader xmlns="urn:schemas-syngress-com-soap">
      <webState>{45D345B6-BE1F-434F-BFD7-D628C756A432}</webState>
    </soapHeader>
  </soap:Header>
  <soap:Body>
    <sessionTest__soapHeader
      xmlns="urn:schemas-syngress-com-soap:sessionTest">

```

```

    </sessionTest__soapHeader>
  </soap:Body>
</soap:Envelope>

```

The server responds, incrementing the session hit counter:

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Header>
    <soapHeader xmlns="urn:schemas-syngress-com-soap">
      <webState>{45D345B6-BE1F-434F-BFD7-D628C756A432}</webState>
    </soapHeader>
  </soap:Header>
  <soap:Body>
    <sessionTest__soapHeaderResponse
      xmlns="urn:schemas-syngress-com-soap:sessionTestst">
      <sessionTest__soapHeaderResult>2</sessionTest__soapHeaderResult>
    </sessionTest__soapHeaderResponse>
  </soap:Body>
</soap:Envelope>

```

If you look at the WSDL description of this Web Service, shown in Figure 4.55, notice that it requests the client to send a *webState* SOAP header, and that this header is required. However, as always, the WSDL file does not contain semantic information helping a client to send a correct request. In other words, although it does instruct clients to include this SOAP header, it does not tell them what it means, or how to properly use it. This is a task that you, as a developer, have to do.

Also, note that the WSDL file does not contain HTTP *GET* and HTTP *POST* bindings for this Web Service. This is because those two methods of calling Web Services do not work when SOAP headers are required.

**Figure 4.55** WSDL Description of the *sessionTest\_\_soapHeader* Web Method

```

<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"

```

Continued

**Figure 4.55 Continued**


---

```

xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="urn:schemas-syngress-com-soap:sessionTest"
xmlns:s1="urn:schemas-syngress-com-soap"
targetNamespace="urn:schemas-syngress-com-soap"
xmlns="http://schemas.xmlsoap.org/wsdl/" >

<types>
  <s:schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="urn:schemas-syngress-com-soap:sessionTest" >
    <s:element name="sessionTest__soapHeader" >
      <s:complexType />
    </s:element>
    <s:element name="sessionTest__soapHeaderResponse" >
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1"
            name="sessionTest__soapHeaderResult"
            nillable="true" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
  </s:schema>
  <s:schema attributeFormDefault="qualified"
    elementFormDefault="qualified"
    targetNamespace="urn:schemas-syngress-com-soap" >
    <s:element name="soapHeader" type="s1:soapHeader" />
    <s:complexType name="soapHeader" >
      <s:sequence>
        <s:element minOccurs="1" maxOccurs="1" name="webState"
          nillable="true" type="s:string" />
      </s:sequence>
    </s:complexType>
    <s:element name="string" nillable="true" type="s:string" />

```

---

Continued

**Figure 4.55 Continued**


---

```

    </s:schema>
</types>
<message name="sessionTest__soapHeaderSoapIn">
  <part name="parameters" element="s0:sessionTest__soapHeader" />
</message>
<message name="sessionTest__soapHeaderSoapOut">
  <part name="parameters"
    element="s0:sessionTest__soapHeaderResponse" />
</message>
<message name="sessionTest__soapHeadersoapHeader">
  <part name="soapHeader" element="s1:soapHeader" />
</message>
<portType name="_sessionTestSoap">
  <operation name="sessionTest__soapHeader">
    <input message="s1:sessionTest__soapHeaderSoapIn" />
    <output message="s1:sessionTest__soapHeaderSoapOut" />
  </operation>
</portType>
<binding name="_sessionTestSoap" type="s1:_sessionTestSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
    style="document" />
  <operation name="sessionTest__soapHeader">
    <soap:operation soapAction="sessionTest__soapHeader"
      style="document" />
    <input>
      <soap:body use="literal" />
      <soap:header n1:required="true"
        message="s1:sessionTest__soapHeadersoapHeader" part="soapHeader"
        use="literal" xmlns:n1="http://schemas.xmlsoap.org/wsdl/" />
    </input>
    <output>
      <soap:body use="literal" />
      <soap:header n1:required="true"
        message="s1:sessionTest__soapHeadersoapHeader" part="soapHeader"
        use="literal" xmlns:n1="http://schemas.xmlsoap.org/wsdl/" />
    </output>
  </operation>
</binding>

```

---

Continued

**Figure 4.55 Continued**


---

```

        </output>
    </operation>
</binding>
<service name="_sessionTest">
    <port name="_sessionTestSoap" binding="s1:_sessionTestSoap">
        <soap:address location="
            http://localhost/sessionTest/sessionTest.asmx" />
    </port>
</service>
</definitions>

```

---

Again, we recommend you think twice (ten times?) before programming stateful Web Services. If you decide to go ahead introducing state, we would advise doing it through SOAP headers, because it seems to be the most natural option you have, and because it is reflected in the WSDL description of your Web Service. WSDL is described in more detail in Chapter 5 “WSDL and UDDI”.

The preceding example should give you a good starting point. However, as you no doubt noticed, the example still needs a bit of work, in particular:

- Although you can add new user sessions, you should have code that is capable of deleting user session information after a certain amount of time (otherwise your memory will eventually fill up to capacity).
- It would be nice to be able to persist user information in a database like MS SQL, the way ASP.NET can do it, and then add a trigger to go off after a specified amount of time cleaning the expired sessions.
- You should add functionality to support Web farms and Web gardens (which, again, ASP.NET does support).

## Security

The SOAP specification does not touch security. You can look at this as a plus, because it keeps the standard small and implementable. RPC protocols that do define security, such as CORBA and COM/DCOM are far more complicated, harder to implement, and don’t work well on the Internet.

On the other hand, as a developer, you obviously shouldn’t ignore security altogether. In the end, you have two possibilities:

- Leverage the security features made available by IIS and ASP.NET.
- Do it yourself.

If you go with the first option, you can secure your Web Services by using the security features of IIS, such as Basic Authentication (probably over SSL), NTLM, or Kerberos-based authentication if you are on an intranet, or authentication-based on Public Key Cryptography (PKC) using client certificates. The latter is particularly interesting for Windows 2000 developers because Active Directory allows you to automatically map client certificates to user accounts if your certificates are issued by a Windows 2000 Certificate Server that's a member of your enterprise domain forest. Note that for this to work, your clients don't need to run on a Windows platform.

Additionally, you can use features provided by ASP.NET on top of what you can do on the HTTP protocol level. ASP.NET allows you to use Microsoft Passport to authenticate users, although you will have to pay licensing fees if you want to go down this route.

ASP.NET also allows you to grant and deny users of your services every imaginable kind of rights once they have been authenticated (this is called *authorization*).

Yet another interesting option is to use SOAP Digital Signature. Also based on PKC, it enables you to digitally sign the body of a SOAP envelope and to include the signature information in a special SOAP header. This does not actually encrypt the SOAP message, but it does guarantee its integrity, that is, you know that nobody has changed its content as it traveled from one machine to another. See [www.w3.org/TR/SOAP-dsig/](http://www.w3.org/TR/SOAP-dsig/) for more information.

Security in the context of Web Services is still very much an evolving area and is currently far from well understood. You can find more information in an article that recently appeared in *XML-Journal* ("Securing and Authenticating SOAP Based Web Services", by M. Moore and A. Turtshi, *XML-Journal*, volume 2, issue 9).



## Summary

Web Services is a new technology designed primarily to facilitate communications between enterprises on the Internet. Web Services are supported by all major software vendors, and are based on Internet standards:

- HTTP as the network protocol (among others)
- XML to encode data
- SOAP as the wire transport protocol

Microsoft's .NET Framework is based on Web Services, and Visual Studio.NET is an excellent platform to develop Web Services. Web Services are different from previous technologies used to create distributed systems, such as COM/DCOM, in that they:

- Use open standards
- Were designed from the ground up to work on the Internet, including working well with corporate firewalls
- Use a “simple” protocol not requiring multiple round trips to the server
- Purposefully don't address advanced features such as security or transaction support as part of the protocol specification

We showed you a variety of examples of Web Services exchanging simple and complex types of data. In addition to using SOAP based Web Services as an RPC (Remote Procedure Call) mechanism, you can use SOAP to exchange any type of XML documents.

We explained the basic structure of the SOAP protocol: SOAP exchanges an XML document called a SOAP Envelope, which has two parts:

- The SOAP Header, which is designed to be extended to include application-specific metadata, such as security- or session-related identifiers
- The SOAP Body, which contains the necessary information to find a class and method on the server to handle the Web Service request, in addition to parameter data that may be necessary to process such a request

The SOAP specification defines a number of XML encoding schemes for different data types, such as strings, integers, floats, arrays, enumerations, and so on. SOAP also includes a mechanisms for error handling.

We showed you how to call Web Services using standalone Visual Basic scripts, client-side script in a Web browser, and through creating Windows Forms-based applications. Visual Studio.NET includes tools that create client proxies for (remote) Web Services for you, greatly simplifying the effort of developing Web Service client applications.

Finally, we talked about two advanced topics that are not directly part of the Web Services standards, but that are nevertheless important for developers, namely security and state management. We recommend to use standard security mechanisms such as SSL and public key cryptography, and to forgo state management until Web Service clients are more robust.

## Solutions Fast Track

### The Case for Web Services

- ☑ Web Services are a new Internet standard, supported by all major vendors, to facilitate data exchange across system boundaries.
- ☑ Standards include a wire protocol (SOAP), a way to describe services (WSDL), and a way to publish services (UDDI).
- ☑ Web Services are classes that extend *System.Web.Services.WebService*.
- ☑ A method becomes a Web method by decorating it with *[System.Web.Services.WebMethod]*.
- ☑ Visual Studio.NET includes a powerful debugger.
- ☑ Once you are in debug mode, external programs calling your Web Service *will* go through the debugger.
- ☑ Writing a Visual Basic script to call your Web Service through SOAP is a fast, easy way to test your application.
- ☑ Visual Studio.NET tells you the correct format of the SOAP request envelope when you open the Web Service overview page (<http://serverName/webServiceProjectName/webServiceName?op=webMethodName>).

## Working with Web Services

- ☑ SOAP can encode arrays, enumerations, and so on. You are rarely directly exposed to the complexities of the underlying protocols because Visual Studio.NET does most of the work for you.
- ☑ Error handling is seamless. Microsoft .NET lets you work with SOAP errors the way you work with any other exceptions.
- ☑ Adding a Web reference lets you use remote Web Services the way you would use local objects, including IntelliSense support, hiding all complexities of SOAP from you.
- ☑ Visual Studio.NET will automatically add client proxy code into your solution.
- ☑ You add a Web reference by pointing to the WSDL description of the Web Service.
- ☑ You can find WSDL files through DISCO or UDDI.
- ☑ SOAP lets you pass instantiated objects between clients and servers. If both the client and the server application run on the .NET platform, the communication is seamless.
- ☑ You can pass any kind of XML through SOAP. This is particularly relevant for interenterprise and third-party integration applications.
- ☑ Visual Studio.NET integrates nicely with UDDI. You can find third-party Web Services and add them to your solutions without ever leaving the development environment.

## Advanced Web Services

- ☑ SOAP itself does not contain a state management mechanism.
- ☑ Web Services should be stateless, even more so than traditional Web applications.
- ☑ If you really do need state information, you may want to look into using SOAP headers.
- ☑ The SOAP protocol does not address security.

- ☑ Use the mechanisms provided by the underlying network protocols, such as encrypting your network channel (HTTPS) and using Public Key Cryptography (certificates).

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** Can I consume Web Services in .NET that have been written in other languages?

**A:** That’s the idea! Web Services define a standard to pass data between heterogeneous systems over the Internet. If you are writing a Web Service client in .NET, you don’t have to worry what language the Web Service you are consuming has been written in, or on what platform it is running.

**Q:** Can Web Services pass binary data efficiently?

**A:** Yes and no. Web Services are based on XML, and thus the emphasis is maybe more on textual data. You can add binary data as CDATA sections in your XML documents you are sending. However, probably a better way is to add binary data as MIME-encoded attachments to your SOAP calls (see the proposed SOAP Messages With Attachments standard at [www.w3.org/TR/SOAP-attachments](http://www.w3.org/TR/SOAP-attachments)). Note, though, that .NET Web Services do not currently support attachments out of the box. If you are sending large amounts of binary data, you may want to look into compressing the data you are sending.

**Q:** Where can I find a list of SOAP implementations?

**A:** Paul Kulchenko maintains a list on his Perl::Lite site at [www.soaplite.com/#Toolkits](http://www.soaplite.com/#Toolkits).

**Q:** Where can I find more information about how the various implementations of SOAP-based Web Services interoperate?

**A:** XMethods maintains the SOAPBuilders Interoperability Lab at [www.xmethods.net/ilab/](http://www.xmethods.net/ilab/). You can also find an excellent overview article discussing the various aspects of interoperability at [www-106.ibm.com/developerworks/webservices/library/ws-asio/?dwzone=webservices](http://www-106.ibm.com/developerworks/webservices/library/ws-asio/?dwzone=webservices).

## WSDL and UDDI

### Solutions in this chapter:

- Web Service Standards
- Describing Web Services—WSDL
- Discovering Web Services—DISCO
- Publishing Web Services—UDDI
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

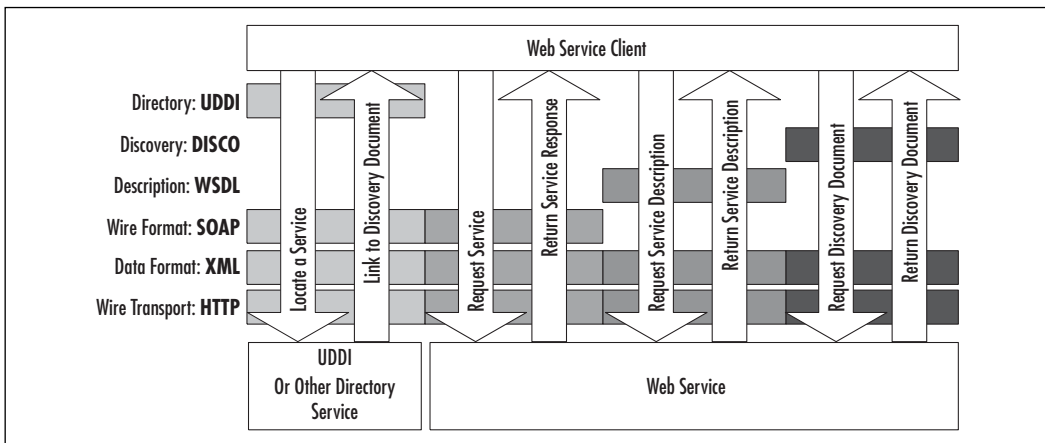
# Introduction

Web Services are useful only if clients can find out what services are available in the first place, where to locate them, and how exactly those services can be called. A number of initiatives are under way driven by the major vendors in the Web Service area to address those application development and business needs. Two of the more important ones, both of which are supported by the Microsoft .NET Framework and fully integrated into Visual Studio.NET , are the following:

- **Web Service Description Language (WSDL)** An XML format to describe how a particular Web Service can be called, what arguments it takes, and so on.
- **Universal Description, Discovery, and Integration (UDDI)** A directory to publish business entities and the Web Services they offer, and where you can find those services. UDDI is implemented as a Web Service itself.

Additionally, there's DISCO, a mechanism based on XML developed by Microsoft to dynamically discover Web Services running on a particular machine. Putting everything together, a picture of the world of Web Services starts to evolve that may look like Figure 5.1.

**Figure 5.1** Web Service Standards



**WARNING**

---

A variety of groups with Microsoft have implemented the SOAP standard. Apart from the .NET Web Services group, these include, among others, .NET Remoting, Windows XP Message Queue, SOAP Toolkit (Web Services based on COM), and BizTalk Server.

Apparently, these groups all have their own code bases, and the various SOAP implementations differ in their level of support of the standard. For instance, the .NET Remoting group implemented “jagged” and sparse arrays, whereas the .NET Web Services did not. Another difference is the support of MIME-encoded attachments. Be aware then when you’re thinking about reusing SOAP code or code designs from one Microsoft product to another that you may have to carefully investigate the details of what exactly is implemented in the various products.

---

## Web Service Standards

In this section, we cover in detail the various Web Services standards introduced in the previous section:

- SOAP, the wire transport protocol
- WSDL to describe Web Services
- DISCO to discover
- UDDI to publish Web Services

You will also write your very first Web Service using the tools provided by Microsoft Visual Studio.NET. By the end of this section, you will have enough knowledge to go ahead and create your own Web Services. The remainder of this chapter then addresses more advanced topics, such as error handling and state management.

## Describing Web Services—WSDL

Because you have programmed the *soapExamples* Web Service that includes the *echo* Web method yourself, from Chapter 4, you “know” how to access it. Well, at least you remember that the *echo* method takes an input parameter, of type *string*, which you called *input*, and returns as its output another string. And although you



may not quite remember how to correctly call this Web method, particularly the gory details of that SOAP envelope, you can always just point your browser to the welcome page (refer back to Figure 4.14) to get more information.

In the world of classic COM, to use an analogy if you are familiar with that framework, classes are described using their interfaces, which in turn were exposed through type libraries. Type libraries are binary files that are created by compiling a file, written in the Interface Definition Language (IDL), that describes the interface of a COM component. It is by enquiring a component type library that a COM client learns how to call a COM server.

In the world of Web Services, the role of a type library is taken by the WSDL description of a Web Service. Not very surprisingly, WSDL is an XML language. Unlike in COM, it does not need to get compiled, which is a very big advantage indeed.

In Microsoft .NET, you can generate the WSDL Web Service description in three ways:

- You can get the WSDL description *dynamically* by calling the Web Service URL appended by the WSDL parameter; in this case, simply `http://localhost/soapExamples/simpleService.asmx?WSDL`. This is the preferred method, because it always gives you an up-to-date description of the service.
- You can (statically) generate the WSDL description by using the `disco.exe` tool found at `%ProgramFiles%\Microsoft.NET\FrameworkSDK\Bin\`. It takes the URL of your Web Service as an argument and writes the information into an XML file. For this example, type **`disco http://localhost/soapExamples/simpleService.asmx`** on a command line.
- Finally, you can programmatically create WSDL files by using the corresponding classes in the `System.Web.Services.Description` namespace. Note that the documentation sometimes erroneously refers to SDL, an older Web Service description technology that is no longer supported, but rest assured that these classes really do deal with WSDL only.

WSDL is a complex standard that is still undergoing changes, and discussing it in detail is beyond the scope of this book; you can find more information about WSDL, including the actual WSDL specification, which is currently stands at version 1.1, at [www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl).

However, you can get a cursory understanding of the structure of WSDL by looking at the WSDL description of the *echo* Web method, which you can access by going to <http://localhost/soapExamples/simpleService.asmx?WSDL> (see Figure 5.2).

**Figure 5.2** WSDL Description for the Echo Web Method

```
<?xml version="1.0" encoding="utf-8"?>
<definitions
  targetNamespace="urn:schemas-syngress-com-soap"
  xmlns:s="http://www.w3.org/2001/XMLSchema"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
  xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:s0="urn:schemas-syngress-com-soap"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema attributeFormDefault="qualified"
      elementFormDefault="qualified"
      targetNamespace="urn:schemas-syngress-com-soap">
      <s:element name="echo">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="input"
              nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="echoResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="echoResult"
              nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
    </s:schema>
  </types>

```

Continued

**Figure 5.2 Continued**


---

```

    <s:element name="string" nillable="true" type="s:string" />
  </s:schema>
</types>
<message name="echoSoapIn">
  <part name="parameters" element="s0:echo" />
</message>
<message name="echoSoapOut">
  <part name="parameters" element="s0:echoResponse" />
</message>
<message name="echoHttpGetIn">
  <part name="input" type="s:string" />
</message>
<message name="echoHttpGetOut">
  <part name="Body" element="s0:string" />
</message>
<message name="echoHttpPostIn">
  <part name="input" type="s:string" />
</message>
<message name="echoHttpPostOut">
  <part name="Body" element="s0:string" />
</message>
<portType name="simpleServiceSoap">
  <operation name="echo">
    <input message="s0:echoSoapIn" />
    <output message="s0:echoSoapOut" />
  </operation>
</portType>
<portType name="simpleServiceHttpGet">
  <operation name="echo">
    <input message="s0:echoHttpGetIn" />
    <output message="s0:echoHttpGetOut" />
  </operation>
</portType>
<portType name="simpleServiceHttpPost">
  <operation name="echo">
    <input message="s0:echoHttpPostIn" />

```

---

**Continued**

**Figure 5.2 Continued**


---

```

        <output message="s0:echoHttpPostOut" />
    </operation>
</portType>
<binding name="simpleServiceSoap" type="s0:simpleServiceSoap">
    <soap:binding
        transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
    <operation name="echo">
        <soap:operation soapAction="urn:schemas-syngress-com-soap/echo"
            style="document" />
        <input>
            <soap:body use="literal" />
        </input>
        <output>
            <soap:body use="literal" />
        </output>
    </operation>
</binding>
<binding name="simpleServiceHttpGet"
    type="s0:simpleServiceHttpGet">
    <http:binding verb="GET" />
    <operation name="echo">
        <http:operation location="/echo" />
        <input>
            <http:urlEncoded />
        </input>
        <output>
            <mime:mimeXml part="Body" />
        </output>
    </operation>
</binding>
<binding name="simpleServiceHttpPost"
    type="s0:simpleServiceHttpPost">
    <http:binding verb="POST" />
    <operation name="echo">

```

---

Continued

**Figure 5.2 Continued**


---

```

    <http:operation location="/echo" />
    <input>
      <mime:content type="application/x-www-form-urlencoded" />
    </input>
    <output>
      <mime:mimeXml part="Body" />
    </output>
  </operation>
</binding>
<service name="simpleService">
  <port name="simpleServiceSoap" binding="s0:simpleServiceSoap">
    <soap:address
      location="http://localhost/soapExamples/simpleService.asmx" />
  </port>
  <port name="simpleServiceHttpGet"
    binding="s0:simpleServiceHttpGet">
    <http:address
      location="http://localhost/soapExamples/simpleService.asmx" />
  </port>
  <port name="simpleServiceHttpPost"
    binding="s0:simpleServiceHttpPost">
    <http:address
      location="http://localhost/soapExamples/simpleService.asmx" />
  </port>
</service>
</definitions>

```

---

You can see from Figure 5.22 that WSDL has five parts, wrapped in the `<definitions>` XML element:

- The `<types>` section defines all data types used by the service. In this case, you have two types, both of *string* type: the *input* parameter, which is the argument passed to the *echo* Web method, and *echoResponse*, which is the output from *echo* that's returned to the caller.
- The `<message>` section, which defines input and output parameters of the Web Service. It refers back to the data types defined in the `<types>`

section of the preceding code. In this example are six individual `<message>` sections. As you have seen earlier, for simple Web Services, .NET defines three access methods—*HTTP GET*, *HTTP POST*, and SOAP. The echo method uses the request-response message pattern, and you see therefore two `<message>` sections for each of the three access methods: one declaring the input parameter, the other one declaring the output parameter.

- The `<portType>` section ties the access methods to the messages declared in the `<message>` section. Because you have three access methods, you see three corresponding `<portType>` sections.
- The `<bindings>` section declares the protocols used to access the echo Web method—*HTTP GET*, *HTTP POST*, and SOAP. It also defines the encoding used to send data over the wire; for *HTTP GET* and *POST*, you simply use URL encoding, whereas for SOAP you use the encoding mechanism provided by the SOAP standard. This section also defines the value that has to be used in the *SOAPAction* HTTP header.
- Everything is now tied together in the `<service>` section: You see your Web Service, *simpleService*, appear, with its only method, *echo*, that has three bindings attached to it, as explained earlier in this list, that can all be accessed at the URL `http://localhost/soapExamples/simpleService.asmx`.

## Discovering Web Services—DISCO

DISCO, which presumably stands for “discovery”, is a mechanism developed by Microsoft for clients to dynamically locate Web Services. More precisely, DISCO guides clients to the WSDL files describing the call syntax of Web Services. DISCO is not supported by anybody outside Microsoft, and it is unclear what future, if any, DISCO has. In practice, DISCO has largely been replaced by UDDI.

DISCO has two parts. Files with the `.vsdisco` extension contain information where to *dynamically search* for Web Services on the local server. Files with the `.disco` extension, in turn, contain information about *already found* Web Services on the local server, particularly where the corresponding WSDL information is located. You will now immediately realize the problem with DISCO: It is an insular solution in that you need to know both the name of the server and the DISCO location on that server *before* you can query for Web Services.

Microsoft Visual Studio.NET automatically adds and maintains a file with extension `.vsdisco` to Web Service projects. It also puts a VSDISCO file into the root directory of the Web server. These VSDISCO files look like the one shown in Figure 5.3.

**Figure 5.3** A Typical DISCO Discovery File

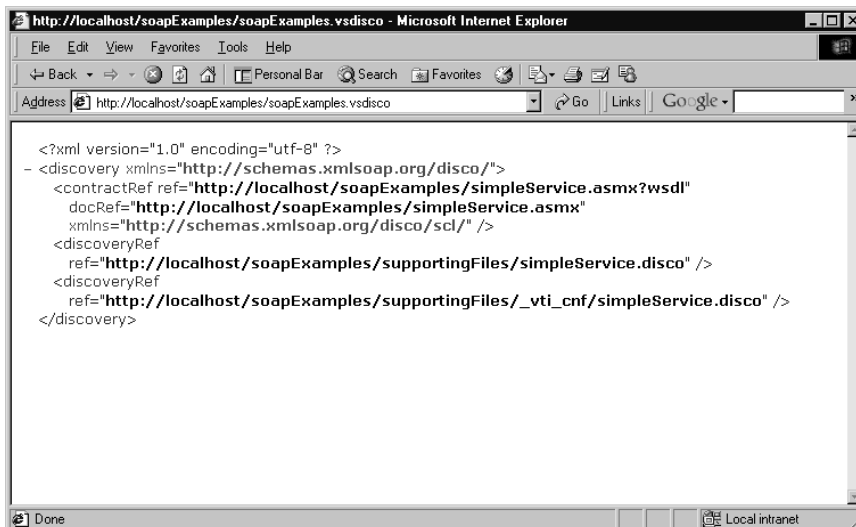
---

```
<?xml version="1.0" ?>
<dynamicDiscovery
  xmlns="urn:schemas-dynamicdiscovery:disco.2000-03-17">
<exclude path="_vti_cnf" />
<exclude path="_vti_pvt" />
<exclude path="_vti_log" />
<exclude path="_vti_script" />
<exclude path="_vti_txt" />
<exclude path="Web References" />
</dynamicDiscovery>
```

---

When you point a Web browser to such a VSDISCO file, Microsoft .NET starts to dynamically query the server for Web Services in the corresponding virtual directory (and below). If you go to the URL `http://localhost/soapExamples/soapExamples.vsdisco`, for example, IIS responds after a while by sending a DISCO file back to you that looks like the one shown in Figure 5.4.

**Figure 5.4** DISCO Information for the *soapExamples* Web Service



You can also statically generate a DISCO file using the `disco.exe` tool found at `%ProgramFiles%\Microsoft.NET\FrameworkSDK\Bin\`. This is the same tool that also outputs the WSDL description. It takes the URL of your Web Service as an argument and writes the information into a file with a `.disco` extension. Unfortunately, this DISCO file contains slightly different information, but it also directs you to the WSDL description of the service, which is really all that matters (see Figure 5.5).

**Figure 5.5** DISCO Discovery File Containing a Reference to WSDL Description

---

```
<?xml version="1.0" encoding="utf-8"?>
<discovery
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/disco/" >
  <contractRef
    ref="http://localhost/soapExamples/simpleService.asmx?wsdl"
    docRef="http://localhost/soapExamples/simpleService.asmx"
    xmlns="http://schemas.xmlsoap.org/disco/scl/" />
  <soap
    address="http://localhost/soapExamples/simpleService.asmx"
    xmlns:q1="urn:schemas-syngress-com-soap"
    binding="q1:simpleServiceSoap"
    xmlns="http://schemas.xmlsoap.org/disco/soap/" />
</discovery>
```

---

## Publishing Web Services—UDDI

Fortunately, a more comprehensive way to locate Web Services exists, and that's the Universal Description, Discovery, and Integration (UDDI) initiative, supported by IBM, Microsoft, and a host of other vendors in the field of Web Services.

UDDI is a Web Service *itself*, and it allows businesses and individuals to publish information about themselves and the Web Services they are offering. It is conceived as a global directory service, open to everybody, simple to use, and comprehensive in its scope. You can find the UDDI home page at [www.uddi.org](http://www.uddi.org).



The three major sponsors of UDDI operate distributed, replicated UDDI services. The access points are as follows:

- **Microsoft** <http://uddi.microsoft.com>
- **IBM** [www.ibm.com/services/uddi](http://www.ibm.com/services/uddi)
- **HP** <http://uddi.hp.com>

Visual Studio.NET supports UDDI through the possibility to query the UDDI directory and add references to Web Services into client applications. You will see an example of that in the next section.

If you want to programmatically interface with UDDI, you can get the Microsoft UDDI SDK, which consists of a series of both COM and .NET classes to interact with the UDDI registry; you can download it from [www.microsoft.com/downloads/release.asp?ReleaseID=30880](http://www.microsoft.com/downloads/release.asp?ReleaseID=30880). Notice, though, that because UDDI is itself a Web Service, you can certainly do everything yourself and interface with it by simply issuing SOAP requests and parsing the SOAP responses from the UDDI server for the information you are looking for.

The industry has put high hopes in UDDI. The functionality of the UDDI registry is still somewhat limited, and the specifications are evolving, but the fact it is so widely supported should encourage you to register yourself, your company, and the Web Services you offer. Best of all, it's free.

## Working with UDDI

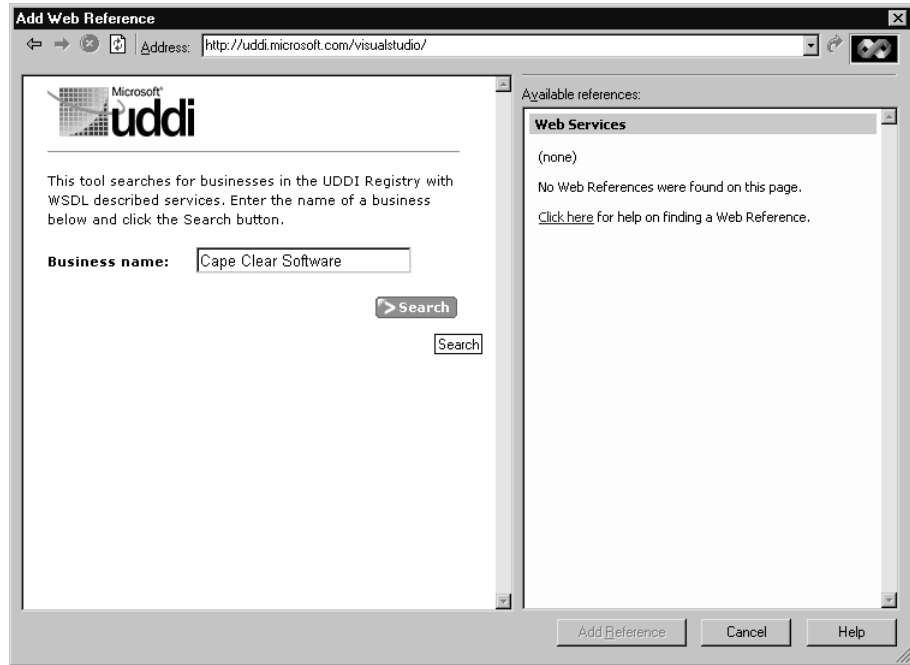
The UDDI registry of Web Services is still in its infancy, and quite frankly, there are not a lot of useful Web Services out there at the time of writing this book. But there are some, and as UDDI seems to be the direction the industry is heading, let's write a simple client application that calls a publicly available third-party Web Service that exposes data about climate conditions of international airports. You can find the complete code for this client application in the directory `uddiClient/` on the Solutions Web Site ([www.syngress.com/solutions](http://www.syngress.com/solutions)) for the book.

You can start by creating a new Windows Forms–based application called *uddiClient*. Query the UDDI registry as follows:

1. Go to the Solution Explorer, right-click the **uddiClient** project, and select **Add Web Reference**.
2. Click **Microsoft UDDI Directory** on the left side of the dialog.
3. Visual Studio.NET will take you to <http://uddi.microsoft.com/>, and ask you to enter the name of the business publishing the service. Enter

**Cape Clear Software**, an early adopter of Web Service technologies (see Figure 5.6).

**Figure 5.6** Searching for a Business in the UDDI Directory



4. UDDI will return a page indicating that it has found Web Services published by Cape Clear Software (see Figure 5.7), among them the Airport Weather Check service. Expand that Web Service, and click the **tModel** hyperlink. Note that if you are interested in the internal structure of UDDI, you will usually find the information relevant for you as a developer under the tModel entries.
5. The tModel contains a link to the WSDL, which will show up on the left panel of the dialog; the right panel tells you that you have one available (Web) reference (see Figure 5.8).
6. Click **Add Reference**. This will create the necessary local client proxy classes to call the *AirportWeather* Web Service.

Figure 5.7 Selecting a Web Service in UDDI

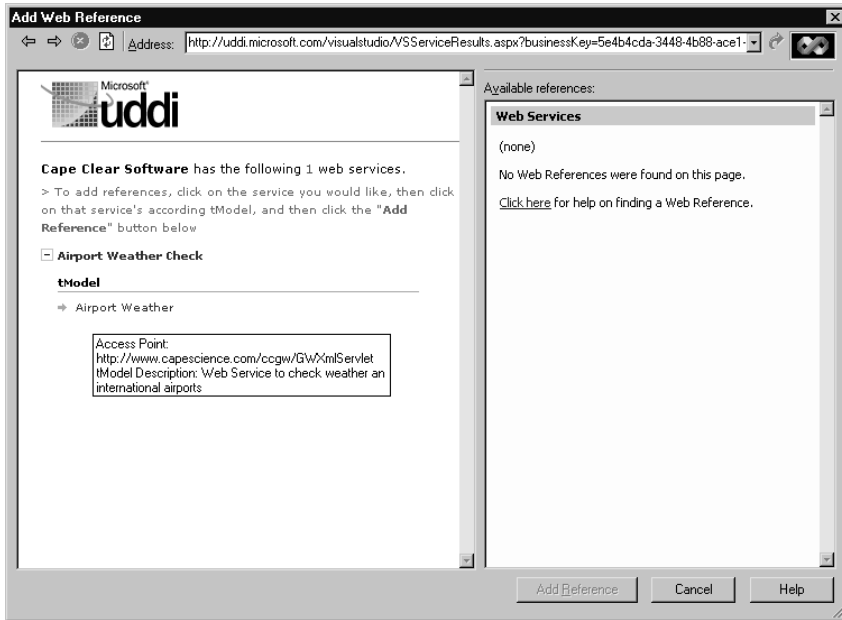
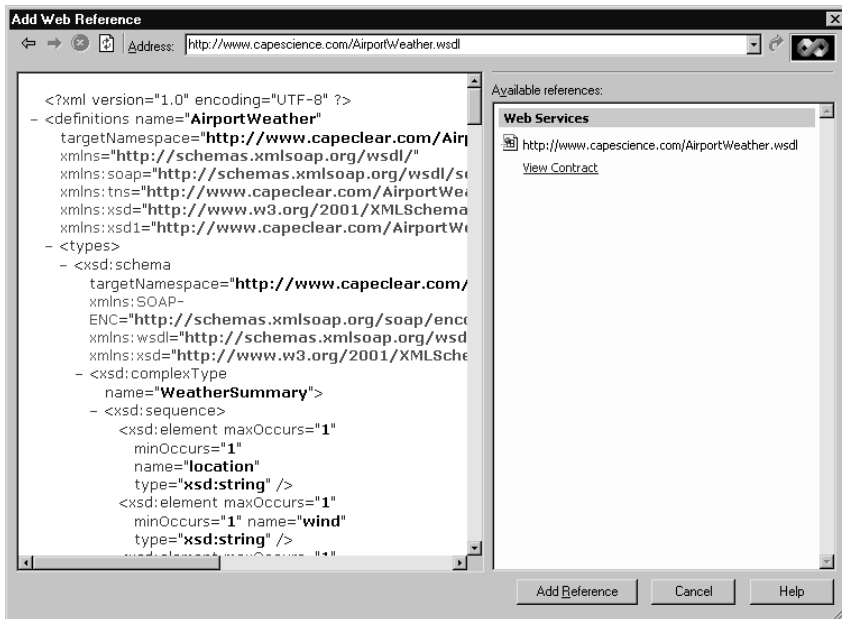


Figure 5.8 Displaying the WSDL Description of a Third-Party Web Service in UDDI

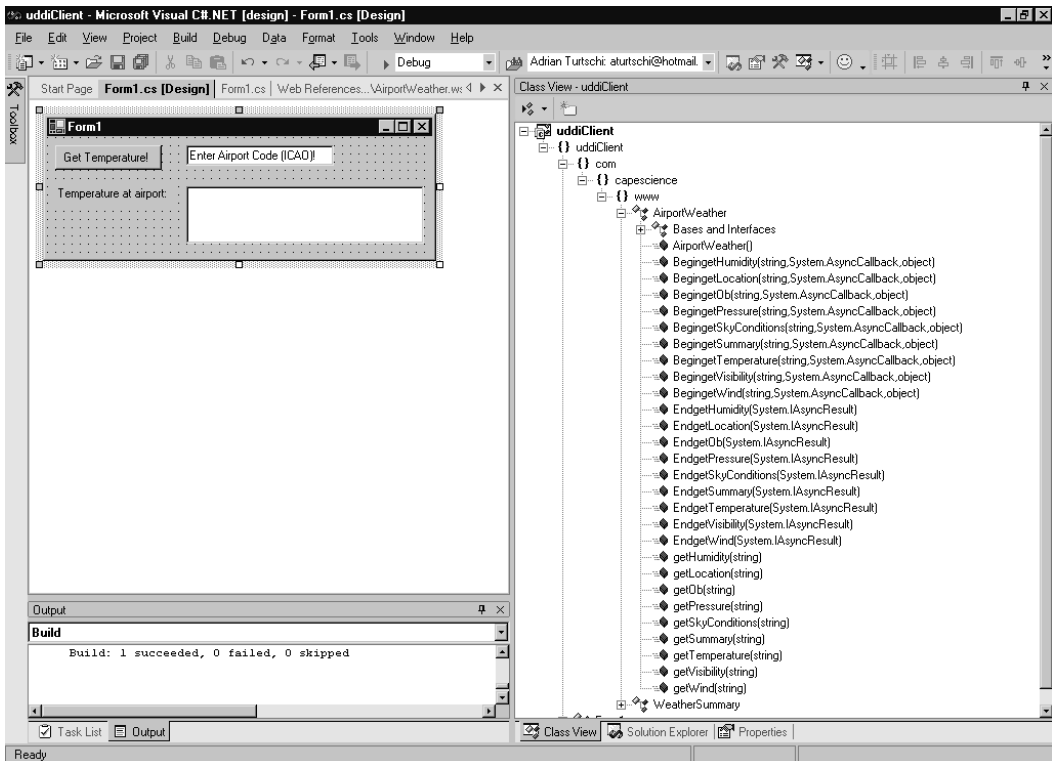


**WARNING**

UDDI support is a recent addition to Visual Studio.NET. In our experience, the UDDI Wizard lacks robustness and tends to crash a lot, forcing Visual Studio.NET to restart. You may want to consider using the `Wsd.exe` command-line tool instead.

If you check what has happened in Visual Studio Class View, you see that a new proxy class `com.capescience.www.AirportWeather` has been added, with a number of methods returning weather-related information of international airports (see Figure 5.9).

**Figure 5.9** Proxy Classes for the AirportWeather Web Service



You are just interested in temperature information, maybe, so you can set up a little Windows form to test the service (see Figure 5.10). The code to call the

Web Service is shown in Figure 5.10. The code shown in Figure 5.10 can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 5.10** Calling the *getTemperature* Web Method (Form1.cs of *uddiClient*)

---

```
private void getTemperature_Click(
    object sender, System.EventArgs e) {
    try {
        com.capescience.www.AirportWeather airportWeather =
            new com.capescience.www.AirportWeather();
        airportTemperature.Text =
            airportWeather.getTemperature(enterAirportCode.Text);
    } catch(Exception ex) {
        // error handling goes here...
    }
}
```

---

One question you may be asking is how do we know the *semantics* of this Web method? After all, the code block invoking the *getTemperature* method looks as in Figure 5.11, that is, the argument to the method is named, rather unfortunately, *arg0*. The code in Figure 5.11 can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 5.11** The *getTemperature* Web Method Definition (AirportWeather.cs of *uddiClient*)

---

```
public string getTemperature(string arg0) {
    object[] results = this.Invoke("getTemperature", new object[] {
        arg0});
    return ((string)(results[0]));
}
```

---

Consulting the WSDL description (see file *AirportWeather.wsdl*) of this method also doesn't help, because the authors did not include any *<description>* XML elements. The answer, then, is to either contact the business that published this Web Service (UDDI does include such information), or hope that a Web page exists out there describing what the Web Service does and what the

parameters *mean*. Luckily, in the case of *AirportWeather*, such a Web page really exists at [www.capescience.com/webservices/airportweather/index.html](http://www.capescience.com/webservices/airportweather/index.html).

You can now test your application by requesting the current temperature at New York's JFK airport, as shown in Figure 5.12. Unfortunately, the authors of this Web Service want you to use the ICAO rather than the more familiar IATA airport codes, but you can get your favorite airport's code at [www.ar-group.com/Airport-Locator.asp](http://www.ar-group.com/Airport-Locator.asp).

**Figure 5.12** The AirportWeather Web Service in Action



We note in passing that there's another slight problem with the Web method, in that it returns a string that contains all the relevant information, but that is difficult to parse if all you really want is the temperature information. Returning a complex XML structure might have been a better design decision.

Finally, let's look at the data exchanged on the level of the SOAP protocol, as seen through a TCP tunneling tool: Figure 5.13 shows the SOAP request to find the current temperature at JFK Airport; Figure 5.14 shows the SOAP response with the relevant data in bold (72F). The code for Figures 5.13 and 5.14 can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 5.13** SOAP Request to Get the Temperature at JFK

```
POST /ccgw/GWXmlServlet HTTP/1.1
User-Agent: Mozilla/4.0
    (compatible; MSIE 6.0; MS Web Services Client Protocol 1.0.2914.16)
Content-Type: text/xml; charset=utf-8
SOAPAction: "capeconnect:AirportWeather:com.capeclear.
    weatherstation.Station#getTemperature"
Content-Length: 630
Expect: 100-continue
Connection: Keep-Alive
Host: localhost
```

Continued

**Figure 5.13 Continued**


---

```

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:tns="http://tempuri.org/"
  xmlns:types="http://tempuri.org/encodedTypes"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body
    soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <q1:getTemperature xmlns:q1="capeconnect:AirportWeather:com.
      capeclear.weatherstation.Station">
      <arg0 xsi:type="xsd:string">KJFK</arg0>
    </q1:getTemperature>
  </soap:Body>
</soap:Envelope>

```

---

**Figure 5.14 SOAP Response with the Temperature at JFK**


---

```

HTTP/1.0 200 OK
Content-Type: text/xml; charset=UTF-8
Content-Length: 601
SOAPAction: "capeconnect:AirportWeather:com.capeclear.
  weatherstation.Station#getTemperature"
Servlet-Engine: CapeConnect/2.1 (Orcas/4.3; Tomcat Web Server/3.2.1)

```

```

<?xml version="1.0"?>
<SOAP-ENV:Envelope
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <ccl:getTemperatureResponse xmlns:ccl="capeconnect:
      AirportWeather:com.capeclear.weatherstation.Station">

```

---

**Continued**

**Figure 5.14 Continued**

---

```
<return xsi:type="xsd:string">The Temperature at New York,  
  Kennedy International Airport, NY, United States is  
  72.0 F (22.2 C)  
</return>  
</ccl:getTemperatureResponse>  
</SOAP-ENV:Body>  
</SOAP-ENV:Envelope>
```

---



## Summary

Web Services is a new technology designed primarily to facilitate communications between enterprises on the Internet. Web Services are supported by all major software vendors, and are based on Internet standards:

- HTTP as the network protocol (among others)
- XML to encode data
- SOAP as the wire transport protocol
- WSDL to describe Web Service syntax
- UDDI to publish Web Service information

Microsoft's .NET Framework is based on Web Services, and Visual Studio.NET is an excellent platform to develop Web Services. Web Services are different from previous technologies used to create distributed systems, such as COM/DCOM, in that:

- They use open standards.
- They were designed from the ground up to work on the Internet, including working well with corporate firewalls.
- They use a “simple” protocol not requiring multiple round trips to the server.
- They purposefully don't address advanced features such as security or transaction support as part of the protocol specification.

We showed you a variety of examples of Web Services exchanging simple and complex types of data. In addition to using SOAP based Web Services as an RPC (Remote Procedure Call) mechanism, you can use SOAP to exchange any type of XML documents. We explained the basic structure of the SOAP protocol: SOAP exchanges an XML document called a SOAP Envelope, which has two parts:

- The SOAP Header, which is designed to be extended to include application-specific metadata, such as security- or session-related identifiers.
- The SOAP Body, which contains the necessary information to find a class and method on the server to handle the Web Service request, in addition to parameter data that may be necessary to process such a request.

The SOAP specification defines a number of XML encoding schemes for different data types, such as strings, integers, floats, arrays, enumerations, and so on. SOAP also includes a mechanisms for error handling.

We showed you how to call Web Services using standalone Visual Basic scripts, client-side script in a Web browser, and through creating Windows Forms-based applications. Visual Studio.NET includes tools that create client proxies for (remote) Web Services for you, greatly simplifying the effort of developing Web Service client applications.

## Solutions Fast Track

### Web Service Standards

- ☑ Web Services are classes that extend *System.Web.Services.WebService*.
- ☑ A method becomes a Web method by decorating it with *[System.Web.Services.WebMethod]*.
- ☑ Visual Studio.NET includes a powerful debugger.
- ☑ Once you are in debug mode, external programs calling your Web Service *will* go through the debugger.
- ☑ Writing a Visual Basic script to call your Web Service through SOAP is a fast, easy way to test your application.
- ☑ Visual Studio.NET tells you the correct format of the SOAP request envelope when you open the Web Service overview page (<http://serverName/webServiceProjectName/webServiceName?op=webMethodName>).

### Describing Web Services—WSDL

- ☑ In the world of Web Services, the role of a type library is taken by the WSDL description of a Web Service.
- ☑ WSDL is a complex standard that is still undergoing changes, and discussing it in detail is beyond the scope of this book; you can find more information about WSDL, including the actual WSDL specification, which is currently stands at version 1.1, at [www.w3.org/TR/wsdl](http://www.w3.org/TR/wsdl)

## Discovering Web Services—DISCO

- ☑ DISCO guides clients to the WSDL files describing the call syntax of Web Services.
- ☑ Microsoft Visual Studio.NET automatically adds and maintains a file with extension `.vsdisco` to Web Service projects.
- ☑ You can also statically generate a DISCO file using the `disco.exe` tool found at `%ProgramFiles%\Microsoft.NET\FrameworkSDK\Bin\`. This is the same tool that also outputs the WSDL description.

## Publishing Web Services—UDDI

- ☑ UDDI is a Web Service *itself*, and it allows businesses and individuals to publish information about themselves and the Web Services they are offering.
- ☑ The UDDI registry of Web Services is still in its infancy, and quite frankly, there are not a lot of useful Web Services out there at the time of writing this book.
- ☑ If you want to programmatically interface with UDDI, you can get the Microsoft UDDI SDK, which consists of a series of both COM and .NET classes to interact with the UDDI registry; you can download it from [www.microsoft.com/downloads/release.asp?ReleaseID=30880](http://www.microsoft.com/downloads/release.asp?ReleaseID=30880).

# Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** Is registration to UDDI free?

**A:** Yes, at the moment it is.

**Q:** Where can I find more information about the business case for Web Services, and how Web Services compare with other distributed technologies such as COM/DCOM, CORBA, and EJBs?

**A:** A good starting point is Orchestra Network’s white paper at [www.orchestranetworks.com/us/solutions/0105\\_whitepaper.cfm](http://www.orchestranetworks.com/us/solutions/0105_whitepaper.cfm).

**Q:** Where can I find more examples of Web Services?

**A:** Visit Visual Studio.NET’s CodeSwap site at [www.vscodeswap.com/](http://www.vscodeswap.com/). XMethods has a large repository of publicly available Web Services at [www.xmethods.net/](http://www.xmethods.net/).



## Building an ASP.NET/ADO.NET Shopping Cart with Web Services

### Solutions in this chapter:

- Setting Up the Database
  - Creating the Web Services
  - Using WSDL Web References
  - Building the Site
  - Site Administration
  - Customer Administration
  - Creating an ADOCatalog
  - Building an XMLCart
  - Creating the User Interface
- 
- ☑ Summary
  - ☑ Solutions Fast Track
  - ☑ Frequently Asked Questions

## Introduction

Now that we've gotten XML under our belt, let's start working with ADO.NET. A good way to really see what ADO can do is within the frame of a shopping cart application. In this chapter, we will create a shopping cart application for a fictitious online bookseller called "Book Shop."

To enable online shoppers to purchase books from our site, our shopping cart application must be able to: Authenticate users, show current contents of the cart, and enable add, update, and checkout operations.

We will also need to create a catalog that our shoppers can browse through to add items to their cart. Users should also be able to query books by category and view a range of books at a time. In order to achieve these goals, we will create the following:

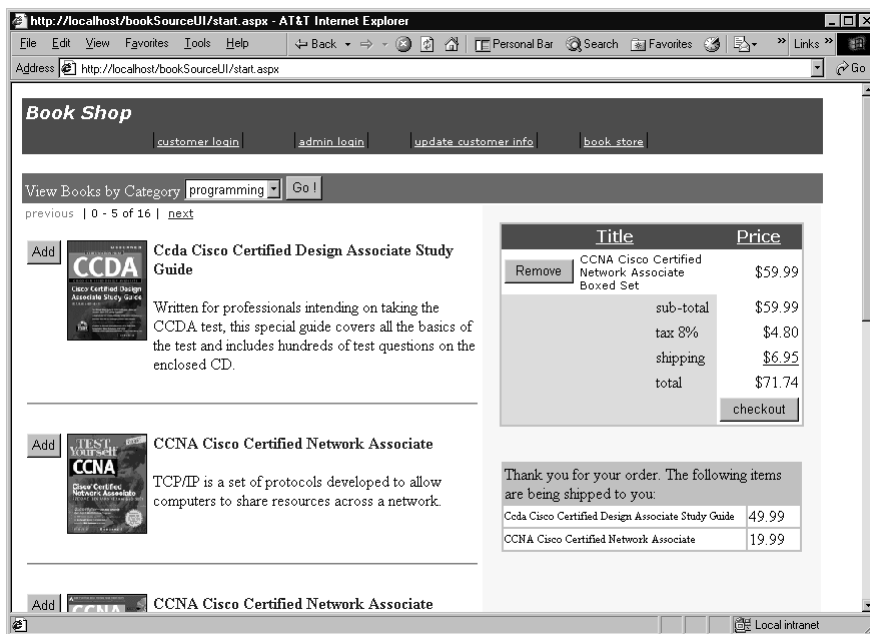
- A database to store all book details
- Stored procedures (MS SQL 2000) or parameterized queries (MS Access 2000) for all add, update, delete, and retrieve operations
- Web Services that will handle all database interactions
- Web Services Description Language (WSDL) Web references to our Web Services
- Server-side classes that will connect the Web Services with our user interface (UI)
- Web interface for displaying both our catalog and cart

We will also need to create admin interfaces to handle add, update, delete, and retrieve operations for our customers (site users) and site administrators. The interface that will be created in our example can be seen in Figure 6.1.

## Setting Up the Database

First, we will design the database for our shopping cart. We will start out by designing an MS Access 2000 database which we will then upsize to a SQL Server 2000 database.

Figure 6.1 The “Book Shop” User Interface

**NOTE**

To set up the database in this example, you will need to know some basic fundamentals of database design. A good source is Syngress Publishing's *Designing SQL Server 2000 Databases for .NET Enterprise Servers* (ISBN 1-928994-19-9).

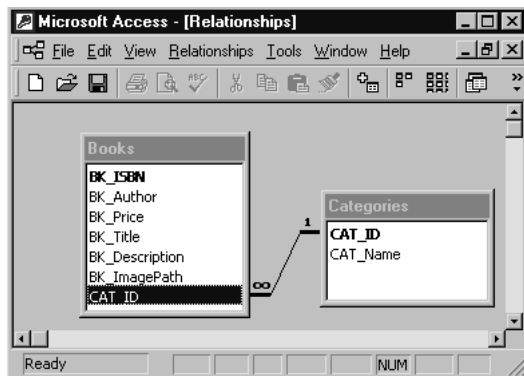
We are creating what is called a *relational database*. A relational database is a series of tables that represent entities related to one another. Let's look at a simple example to help illustrate this point: our database. See Figure 6.2.

Table *Books* is an entity that represents all the attributes of a book. Table *Categories* is an entity that represents all the attributes for a specific category. A relationship between the two tables is created by the use of *primary* and *foreign keys*. Table *Categories* has an attribute named *CAT\_ID*, which is the primary key for the table. This means simply that *CAT\_ID* uniquely identifies every row in the table. This will ensure we won't get duplicate rows of data. The same concept is true for the table *Books*. We can create the relationship between the two tables by putting the attribute *CAT\_ID* into the table “Books.” By doing so, we have



created a foreign key in the table *Books* which references the table *Categories*. We have now created a *one-to-many* relationship between the table *Books* and the table *Categories*.

**Figure 6.2** Table Relationship



There are three different types of table relationships:

- **One-to-one** Exactly one row corresponds with a matching row of the related table.
- **One-to-many** One row corresponds to many rows of the related table.
- **Many-to-many** Many rows correspond to many rows of the related table.

## WARNING

A many-to-many relationship between tables is not a recommended practice. When this type of relationship is created in the design of your database, use a splitter table in-between the two tables that have the affected relationship. This will create two one-to-many relationships and ensure data integrity for your database.

We will now create the *entities* for our shopping cart application. Entities enable us to map the real world. Since we are making a shopping cart, we need some basic objects to start off with. First of all, we need product. We have chosen to use *Books* as the product for the shopping cart but this could be anything. Next, we need an object that will be using the shopping cart, *Customers*. As in the

previous paragraph, we have more than one category of product, or in our case *Books*, so we have another object to map which is *Categories*. The last piece to finish off the whole design is a way to track what is bought, *BookOrders*. Now we need to go over each entity to explain why we have selected the attributes included in each.

## Setting Up the Table *Books*

The *Books* table will contain the following attributes:

- ***BK\_ISBN*** This will also be our Primary key for the table since an ISBN is already a global unique identifier.
- ***BK\_Author*** This contains the author's full name.
- ***BK\_Price*** The price of the book.
- ***BK\_Title*** The book title.
- ***BK\_Description*** A brief description of the book.
- ***BK\_ImagePath*** The path to where we will store the image.
- ***CAT\_ID*** Our foreign key attribute to table "Categories."

## Setting Up the Table *Categories*

The *Categories* table will contain the following attributes:

- ***CAT\_ID*** The primary key for the table which will be an auto generated number; I will cover this in the next two sections.
- ***CAT\_Name*** The name of the category.

## Setting Up the Table *Customer*

The *Customer* table will contain the following attributes:

- ***CT\_ID*** The primary key for the table, an auto generated number.
- ***CT\_FirstName*** Customer first name.
- ***CT\_LastName*** Customer last name.
- ***CT\_Email*** Customer e-mail.
- ***CT\_Password*** Customer password.

## Setting Up the Table *Orders*

The *Orders* table will contain the following attributes:

- ***OR\_ID*** The primary key for the table, an auto generated number.
- ***CT\_ID*** This is our foreign key attribute to table “Customers.”
- ***OR\_Date*** The date of the order.
- ***OR\_ShippedDate*** The date the order ships.

## Setting Up the Table *BookOrders*

The *BookOrders* table is the split table for the handling of our relationship between the tables *Books* and *Orders*. This table includes the following attributes:

- ***OR\_ID*** This is our foreign key attribute to table “Orders.” This is also part of the composite Primary key for the table.
- ***BK\_ISBN*** This is our foreign key attribute to table “Books.” This is the other part of the composite primary key.
- ***BKOR\_Quantity*** The total of number of books.
- ***BKOR\_Price*** The total amount of the order.

### NOTE

It is good practice to come up with a naming convention for your database. The naming convention can be anything of your choosing, just make sure you’re consistent throughout your database. A naming convention is a uniformed way to document your code. In our example, *OR\_* denotes the table *Orders*.

Now, let’s implement this database in Microsoft Access.

## Creating an Access Database

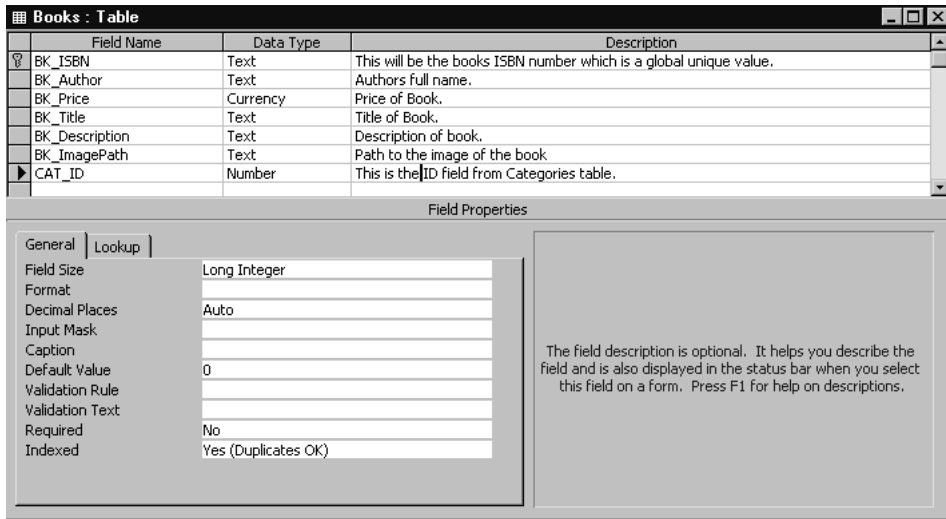
To create a database in Microsoft Access, simply navigate to **Programs | Microsoft Access**. The main window will pull up, prompting you to either pick a database from the list of current databases, create a blank database, or use the wizard. See Figure 6.3.

Figure 6.3 Setting Up the Access Database



We want to select the **Blank Database** option and not the wizard. Select **OK**, then give the database the name **shopDb**. Next, select the **Tables** object. From here, we choose the option **Create table in design view**. We can now transfer the attributes for the tables into the interface (see Figure 6.4).

Figure 6.4 Creating Tables in Design View



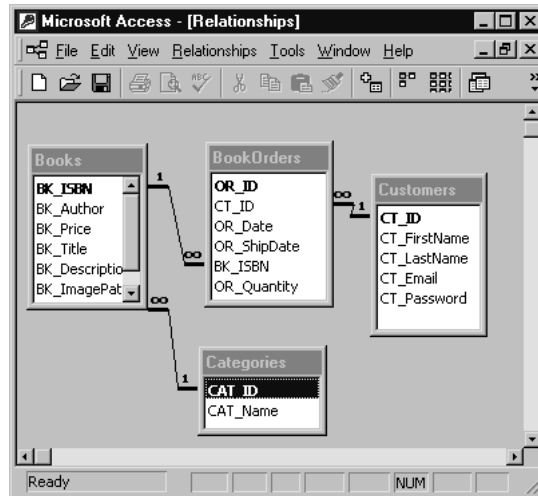
Now we can transfer almost everything that's been done into the interface. One thing we have not discussed is *datatypes*.

The following is a list of datatypes we will implement in the database:

- **Text** Text or combinations of text and numbers: Maximum size 255 characters.
- **Currency** Used for monetary functions, prevents rounding off of total: Size 8 bytes.
- **AutoNumber** Unique number automatically inserted when a record is added: Size 4 bytes.
- **Number** Numeric data to be used for mathematical calculations: Size 1, 2, 4, or 8 bytes.
- **Date/Time** Stores date/time: Size 8 bytes.
- **Yes/No** Boolean value, 0 or 1: Size 1bit.
- **Memo** Used for storing large amounts of text: Maximum size 64,000 characters.
- **OLE Object** Can store Word docs, Excel files, and so on: Maximum size 1 gigabyte.

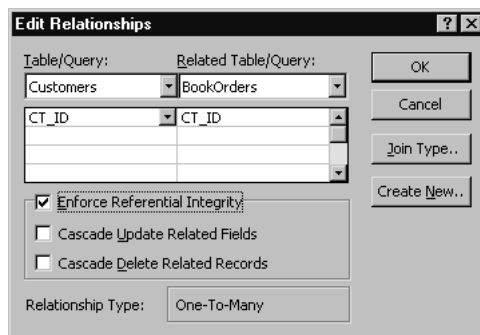
Continue this process for the rest of the tables. If you want, you can download the file shopDb.mdb from the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) and view the complete database. Let's look at the complete diagram generated by Access after we finish filling in our tables (shown in Figure 6.5).

**Figure 6.5** A Database Diagram



To generate the preceding diagram, go to the **Tools** menu and select the **Relationships** option. You will be prompted for what tables to add. Select the tables you have created and hit **OK**. To create the relationships between the tables, left-click the **attribute** you want to make a relationship with and drag it over to the **table** that has the matching attribute, release the mouse and you will be prompted with a set of options for the relationship. See Figure 6.6.

**Figure 6.6** Defining Relationships in Access



The default is to have the **Enforce Referential Integrity** option selected. This is good enough for our example; the other two options will enable cascading deletes and updates.

## WARNING

When defining relationships, make sure the column is of the same datatype as the one you are trying to make a relationship with, otherwise Access will throw an error.

We will do what is called *de-normalize* the database for the Access version to make things flow between the Data tier of our application and the two different databases. Since our shopping cart uses all OleDb connections to the database regardless of source, the stored procedures created in the SQL Db are the same for the Access version, but we have some limitations when it comes to Access. We cannot easily return the submitted record ID from the table like we can in SQL using the global variable `@@identity`, so we must solve this by eliminating the Orders table in the schema for Access and adding those rows to the *BookOrders* table. This will result in customers having multiple order entries, but keeps all

data handling code the same for both databases. If you were to program this application, you would select one or the other and optimize accordingly—we are going to straddle the fence here and show both in the same logic.

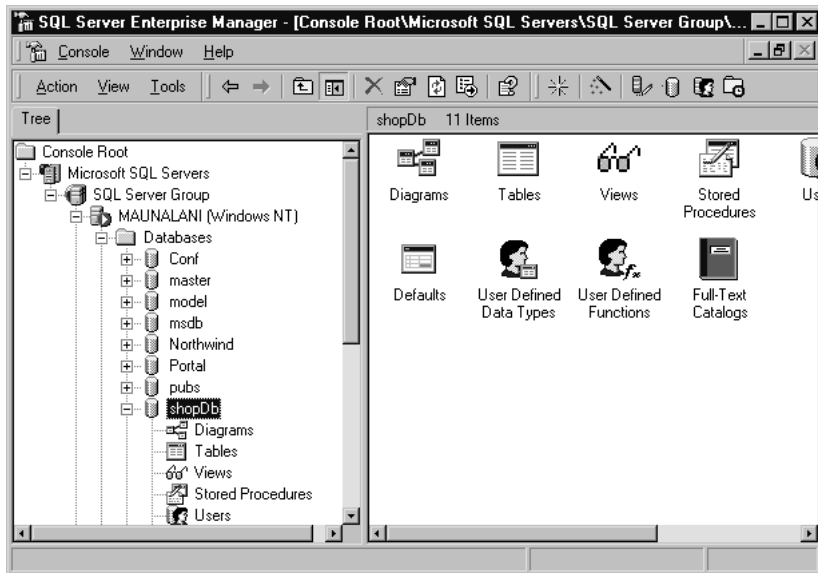
Now that we have our database schema done, we can upsize the database using the Access Upsizing Wizard and make a SQL server version. Go to **Tools | Database Utilities | Upsizing Wizard**. Follow the wizard and choose all the defaults.

## SQL Server Database

Now that we have our schema upsize into SQL, we can easily create the rest of our database components. We primarily need a set of stored procedures that will run all of our operations against the database. This will enable us not to have to use ad hoc queries in our code for our Data Tier interaction.

One thing we need to do first is ensure that all our primary keys were transcribed into the upsize version. Let's open up the **Enterprise Manager of MSSQL 2000 (EM)**. Navigate to your program files and select the **SQL Server** group, then select **EM**. From EM, we can quickly navigate to our database (*shopDB*). See Figure 6.7.

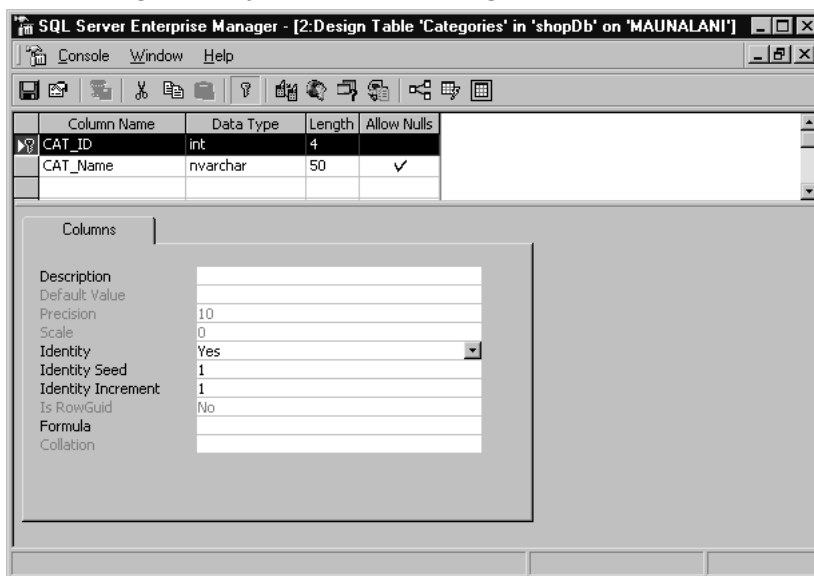
**Figure 6.7** The SQL EM Interface



Select **Tables** and you'll notice our tables from Access are now here. Right-click a table and select **Design Table**. From here, we can check to see if our

tables made the move without ill effects. If everything looks correct, check the rest of the tables—you'll see the Access datatype *autonumber* does not come over to SQL Server as an *int* identity column datatype, which it needs to be. So, for the tables that have *autonumber*, you will have to change it to the *int* datatype with identity, and give them a seed and increment value. See Figure 6.8.

**Figure 6.8** Setting Identity to Yes and Giving Seed and Increment Value



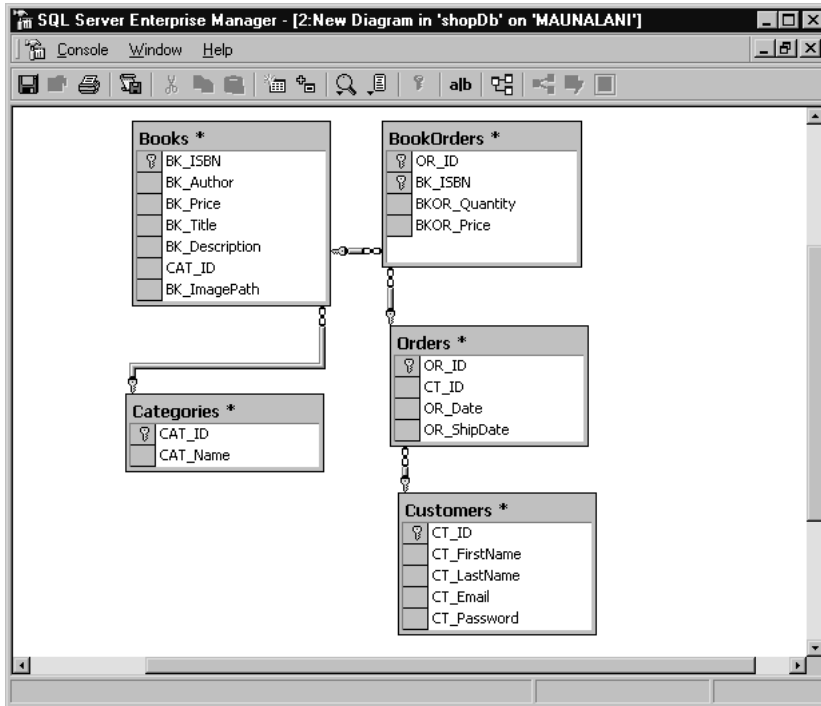
You must also uncheck **Allow Nulls**. This is because the field we are working with is a primary key and we cannot have a null value for a primary key field. Also, we are using the option identity in this instance, which requires that null not be allowed.

We will also separate the table *BookOrders* into its original design since SQL Server can easily give us a value for the identity field returned. After we have done all of this, we can create a new diagram in SQL and apply our new relationships. In the EM view, right-click diagrams and select **New Diagram**. The wizard will prompt you for the tables you want to select for the database diagram. Add only the tables we have created, leave out all the system tables. We will now view our new diagram generated by SQL Server (see Figure 6.9).

We can create relationships in the same manner as before. Click the column you want to make a relationship with and drag and drop it into to the appropriate column and table. We will go with the selected defaults. We have a normalized database now completed in SQL Server. We will now create the stored procedures we'll need for the rest of the application.



Figure 6.9 A SQL Server Diagram



## Creating the Stored Procedures

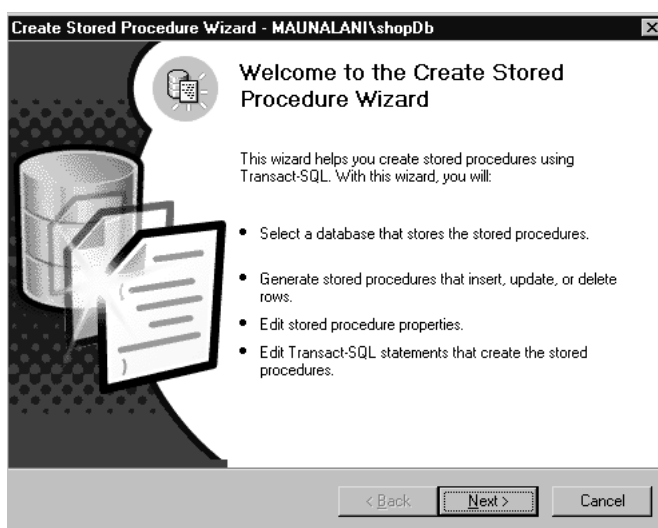
We'll now create the following list of stored procedures:

- *AdminAddBook*
- *AdminAddCustomer*
- *AdminAddCat*
- *AdminDeleteCat*
- *AdminDeleteCustomer*
- *AdminDeleteBook*
- *AdminUpdateBook*
- *AdminUpdateCat*
- *AdminUpdateCustomer*
- *AllCustById*
- *GetAllBooks*

- *GetAllCat*
- *LoginCustomers*
- *OrderBook*

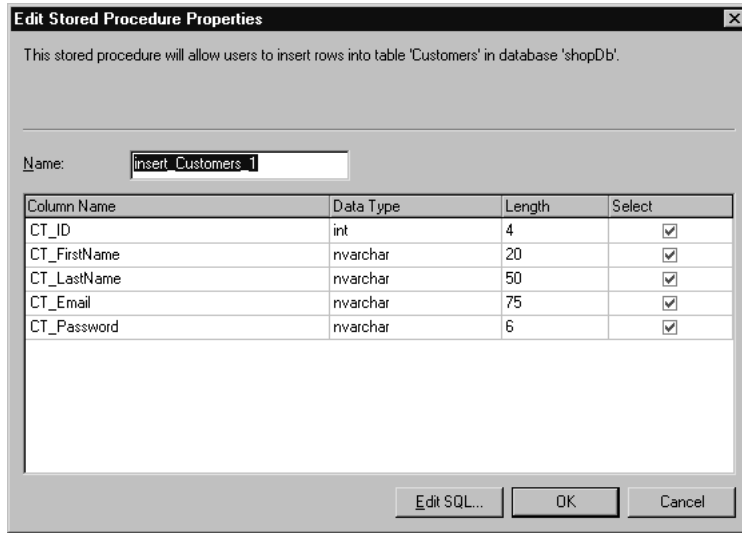
Don't be intimidated. We'll use the SQL Server Wizard to create most of these procedures. Now we need to begin creating all our stored procedures. Go to the **Tools** menu and select **Wizards**. From there, a new window will pop up with a listing of items. Double-click the first item, **Database**, then select **Create Stored Procedure Wizard**. You should see the screen shown in Figure 6.10.

**Figure 6.10** The Create Stored Procedure Wizard



Click **Next** and select the database, which is **shopDb**. The next window will show all the tables on the left and the subsequent procedures that can be created on the right. Mark the check box labeled **insert** in the row of options listed for the **Customers** table. Click **Next**. The window that appears will give you the choice to Edit the SQL syntax—select this option. We need to give the procedure a name, which in this case will be *AdminAddCustomer*. See Figure 6.11.

In Figure 6.11, we see that all columns are selected for insert; however, we do not need one for *CT\_ID* because the identity field generates that. Uncheck that option and rename the stored procedure **AdminAddCustomer**. Select **Edit SQL**. Let's look at the code generated by this; it's shown in Figure 6.12 and found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) as *ShopDB.sql*.

**Figure 6.11** The Stored Procedure Wizard's Properties Dialog Box**Figure 6.12** ShopDB.sql

```

USE [shopDb]
GO
CREATE PROCEDURE [AdminAddCustomer]
    (@CT_FirstName    [nvarchar](20),
    @CT_LastName     [nvarchar](50),
    @CT_Email        [nvarchar](75),
    @CT_Password     [nvarchar](6))

AS INSERT INTO [shopDb].[dbo].[Customers]
    ([CT_FirstName],
    [CT_LastName],
    [CT_Email],
    [CT_Password])

VALUES
    (@CT_FirstName,
    @CT_LastName,
    @CT_Email,
    @CT_Password)

```

Here we have the SQL syntax to insert a row of new data. In the code view window, SQL Server likes to put numbers on all the variables. You can delete this so the code looks cleaner and will be easy to use when we write the Web service that will hit this proc and execute it. Create the rest of the Admin stored procedures in this same manner.

Now that we have completed a majority of the stored procedures needed for our database through the use of the wizards, we have to create more complex stored procedures using the Query Analyzer. Open up **Query Analyzer** from the **Tools** menu of **EM**. Connect the server you are running. In the drop-down menu, select the database **shopDb**. The next stored procedure that you need to build is *AllCustById*. We will write a simple select statement with one parameter. Let's look at some code which can be executed in Query Analyzer:

```
CREATE PROC AllCustById
@CT_ID int
AS
SELECT *
FROM customers
WHERE CT_ID = @CT_ID
GO
```

The next procedure in the list after *AllCustById* is *GetAllBooks*. No need for parameters—just give up the data.

```
CREATE PROCEDURE GetAllBooks

AS

SELECT BK_ISBN isbn,
       category.CAT_Name "name",
       category.CAT_ID "id",
       BK_ImagePath imgSrc,
       BK_author author,
       BK_Price price,
       BK_Title title,
       BK_Description "description"
FROM Books book inner Join Categories category
on book.CAT_ID = category.CAT_ID

ORDER BY "name"
```

**NOTE**


---

In the code in this section, we are using aliasing so the column headers returned will have easy-to-use names. The *DataSet* will use the column names as XML element names when the data is converted to XML.

---

Now we need to get a selection of categories from the database for our drop-down menus:

```
CREATE PROC GetAllCat
AS
SELECT * FROM Categories
```

This will populate with all category names and associated IDs.

Now we need to create a proc that will query the database and return a Customer's ID. This is our Login stored procedure:

```
CREATE proc LoginCustomers
    @CT_Email nvarchar(75),
    @CT_Password nvarchar(6)

as

SELECT [CT_ID]
FROM Customers
WHERE CT_Email = @CT_Email And CT_Password = @CT_Password
```

This will return a value of either the Customers ID or -1, which we can check for on the page load.

Now we need to handle the ordering of a book. We can load and run the *OrderBook* procedure to do that:

```
CREATE Procedure OrderBook
(
    @CT_ID int,
    @BK_ISBN int,
    @BKOR_Quantity int,
    @BKOR_Price money
)
AS
```

```
declare @OR_Date datetime
declare @OR_ShipDate datetime
declare @OR_ID int

select @OR_Date = getdate()
select @OR_ShipDate = getdate()

begin tran NewBook

INSERT INTO Orders
(
    CT_ID,
    OR_Date,
    OR_ShipDate
)
VALUES
(
    @CT_ID,
    @OR_Date,
    @OR_ShipDate
)

SELECT @OR_ID = @@Identity

INSERT INTO BookOrders
(
    OR_ID,
    BK_ISBN,
    BKOR_Quantity,
    BKOR_Price
)
VALUES
(
    @OR_ID,
    @BK_ISBN,
    @BKOR_Quantity,
```

```

        @BKOR_Price
    )
commit tran NewBook

```

We are using *begin tran* and *end tran*. This simply means that if there is an error during any part of the previous query the transaction will be aborted and rolled back. That's it for the stored procedures. Now to make these all work in the Access database, you need to trim out some stuff from the preceding code.

As a rule of thumb, you can grab all the code after the key word *AS*. This is then pasted into Access query SQL mode and saved as the same file name. Open up the shopDB.mdb file (which you downloaded earlier from the Syngress Solutions Web site for the book) and see the differences in the code.

## Creating the Web Services

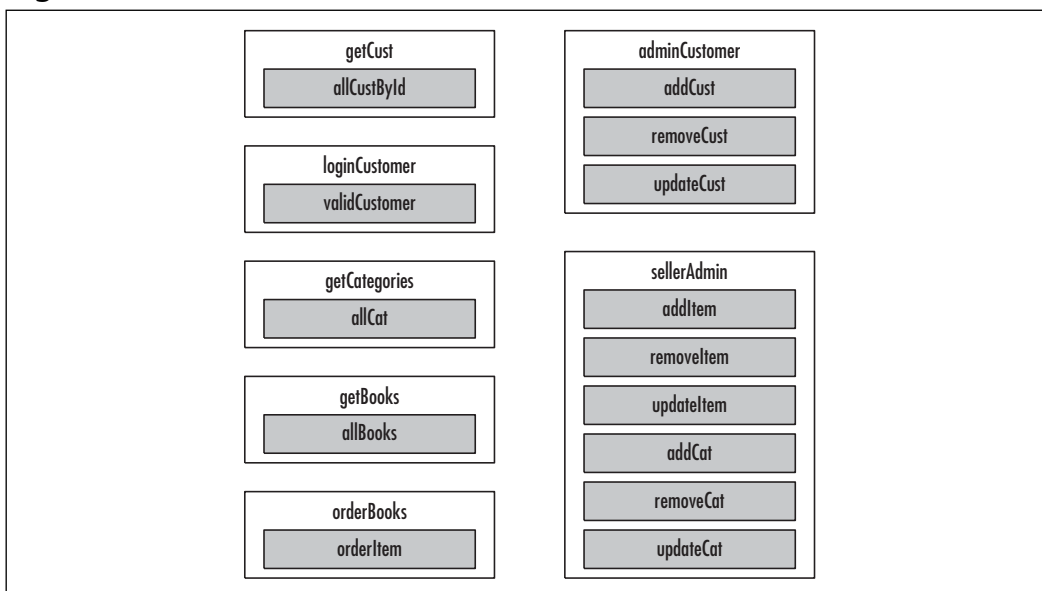
This section will provide an overview of the Web Services needed for the site, and describe the processes of creating the data connection, creating a Web Service, and, finally, testing the Web Service.

### Overview of the Book Shop Web Services

We will be using Web Service methods to wrap our database logic (stored procedures for SQL, or parameterized queries for Access). This will provide separation of the data tier from the UI. This will also enable our data to be accessed from multiple clients including Java-servlets, JSP, PHP, desktop application with Hypertext Transfer Protocol (HTTP) connections, and, of course, ASP.NET applications.

We will be creating the following Web Services (see Figure 6.13):

- *sellerAdmin*
- *adminCustomer*
- *getCustomer*
- *loginCustomer*
- *getBooks*
- *getCategories*
- *orderBooks*

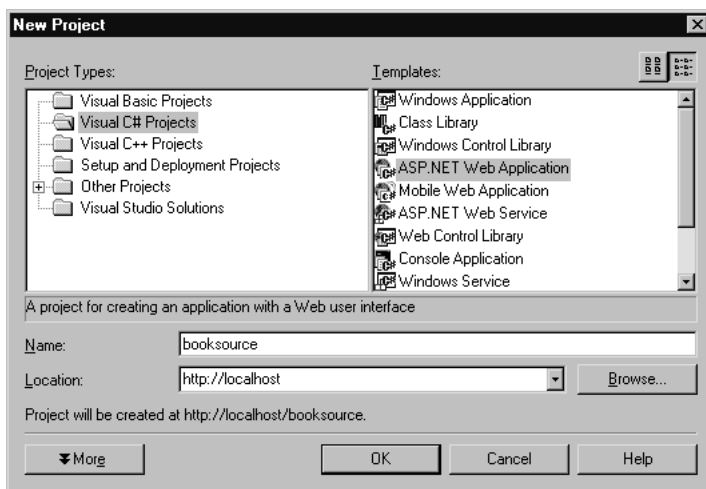
**Figure 6.13** An Overview of Web Services and Their Methods

Earlier in this chapter (see the section “Setting Up the Database”), you created stored procedures for use with an SQL database, as well as the equivalent parameterized queries for use with an Access database, to make the interface to the data source consistent; this allows us to write ADO.NET code that can be used against both SQL and Access.

You will also use the OleDb data connection object since most databases have an OleDb provider. This will enable your code not only to work with SQL and Access but with any database that has an OleDb interface. So, the application will work with an SQL database and the application will work with an Access database. The only code that will need to be changed with this approach is the connection string.

Let’s create a new project to host all of the Web Services. Open **Visual Studio .NET** (VS.NET), and select **New Project**. We want to create a C# ASP.NET Web Service application named *booksource* (see Figure 6.14); next, we will create the data connection.



Figure 6.14 Creating the *booksource* Web Service

## Creating the Data Connection

Data Connections can be created in several ways. Let's look at how the VS.NET Wizard does this. For this example, you'll create a connection to an Access database. The steps for MS SQL will be slightly different.

1. Open the **Server Explorer**, and select **View | Server Explorer** from the menu.
2. Right-click **Data Connection**, then select **Add connection**.
3. Select the **Provider** tab.
4. Select the appropriate provider. For access, select **Jet 4.0 OLEDB Provider**.
5. Click **Next**.
6. Select the database name by clicking the **Browse...** button and navigating to your database.
7. Click **Test Connection**. You should get a pop-up window that says **Connection succeeded**.
8. Click **OK**.
9. Click **OK**. You now have a data connection.

While in design mode, you can drag and drop this connection onto your .asmx page. This will add the following to the code-behind page as the first line in the *service* public class:

```
private System.Data.OleDb.OleDbConnection OleDbConnection1;
```

Connection string information will also be added to the *InitializeComponent()* method. Alternatively, you can still create a connection string by creating a .udl file on the desktop, double-clicking it and following the dialogs. With this method, you will have to insert the code ourselves, as follows:

1. In C#, add **Using System.Data.OleDb** to the top “using” section.
2. Then add the following inside the *service* class:

```
private OleDbConnection myConnection = new OleDbConnection();
```

3. Add the following to a method (*page\_onload*, or a method of your own creation):

```
myConnection.ConnectionString =  
[the string obtained from the udl file]
```

We will take a closer look at adding a connection when we create the *sellerAdmin* service in the next section.



## Creating a Web Service

All of the code for the Web Services in this chapter can be found on the Solutions Web site for the book at [www.syngress.com/solutions](http://www.syngress.com/solutions). (See: *adminCustomer.asmx.cs*, *sellerAdmin.asmx.cs*, *getBooks.asmx.cs*, *getCategories.asmx.cs*, *getCustomer.asmx.cs*, *loginCustomer.asmx.cs*, *orderBooks.asmx.cs*, and *sellerAdmin.asmx.cs*.)

Let's take a closer look at adding a connection by creating the *sellerAdmin* Service. To create this service follow these steps:

1. Create the *Connection* object.
2. Set the *Connection* string.
3. Create the *Command* object.
4. Create the *Parameter* objects and assign their values.

5. Execute the procedure. You will be using the *AdminAddBook* stored procedure. It takes the following parameters: *BK\_ISBN*, *BK\_Author*, *BK\_Price*, *BK\_Title*, *BK\_Description*, *CAT\_ID*, *BK\_ImagePath*.
6. Return string indicating success or failure of the operation.

Now let's get started. To accomplish Step 1 (creating the *Connection* object), first create a new C# Web Service and name it **sellerAdmin.asmx**. Add this directive to the top "using" section:

```
Using System.Data.OleDb;
```

Scroll down to below the method named *Dispose*(*bool disposing*). Add the following:

```
protected OleDbConnection sellerAdminConn = new OleDbConnection();
```

This accomplishes the creation of the *Connection* object. Now, for Step 2 (setting the *Connection* string), add the following:

```
protected void init()
{
    this.sellerAdminConn.ConnectionString =
    @"Provider=SQLOLEDB.1;
    Persist Security Info=False;
    User ID=[user id]; password=[password]; Initial Catalog=[Database Name];
    Data Source=[Server Name]"
}
```

Note that the use of the "@" before the *Connection* string is required. This accomplishes Step 2.

For Step 3, (creating the *Command* object), first create a new method called *addItem*. It should have parameters corresponding to the stored procedures parameters, and should return a string indicating success or failure of the operation:

```
public string addItem(string ISBN, string author, double price, string
    title, string description, string imagePath, int CAT_ID)
```

Now create a *Command* object that references the *AdminAddBook* stored procedure:

```
OleDbCommand addItem =
new OleDbCommand("AdminAddBook", this.sellerAdminConn);
addItem.CommandType = CommandType.StoredProcedure;
```

This accomplishes Step 3.

For Step 4 (creating the *Parameter* objects and assigning their value), we will create *Parameter* objects for *ISBN*, *author*, *price*, *title*, *description*, *imagePath*, and *CAT\_ID*, then set their values. Here is the code for *isbn*:

```
OleDbParameter addISBN =
addItem.Parameters.Add( "@BK_ISBN",OleDbType.Char,15);
addISBN.Value = ISBN;
```

Note that *@BK\_ISBN* is the name of the parameter we are assigning a value to; *OleDbType.Char* is its datatype (it should be compatible with the field in the database); and “15” refers to the character size as defined for the field in the database.

The code to create *Parameter* objects for each of the method parameters is nearly identical, and can be found on the on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the file *sellerAdmin.asmx.cs*. This accomplishes Step 4.

Now, for Step 5 (executing the procedure), open the connection and execute the query. Since the stored procedure performs an insert operation it will return an *int* containing the number of rows affected. Therefore, you will use the command *ExecuteNonQuery*.

```
this.sellerAdminConn.Open();
int queryResult = QueryObject.ExecuteNonQuery();
```

This accomplishes Step 5. Now close the connection and return the result of executing the stored procedure (this is Step 6). Note that the method returns the following string: “success” or the generated error message.

```
this.sellerAdminConn.Close();
if ( queryResult != 0)
{
    return "Success";
}
else
{
    return "error: QueryResult= " + queryResult;
}
```

This accomplishes Step 6. Since all of the Web methods have similar logic, you can combine some of this code into a method that each Web Method calls:



```
protected string ExecuteQuery( OleDbCommand QueryObject)
{
    this.sellerAdminConn.Open();
    int queryResult = QueryObject.ExecuteNonQuery();
    if ( queryResult != 0)
    {
        this.sellerAdminConn.Close();
        return "Success";
    }
    else
    {
        return "error: QueryResult= " + queryResult;
    }
}
```

Add one more thing to the method to make it accessible as a Web method:

```
[ WebMethod(Description="Adds a new book to the books table",
            EnableSession=false)]
```

Putting it all together, you should get the following:

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Data.OleDb;

namespace bookSource
{
    public class sellerAdmin : System.Web.Services.WebService
    {
        public sellerAdmin()
        {
            InitializeComponent();
        }
    }
}
```

```
protected override void Dispose( bool disposing )
{
}

protected OleDbConnection sellerAdminConn =
    new OleDbConnection();

protected void init()
{
    this.sellerAdminConn.ConnectionString =
        @"Provider=SQLOLEDB.1;
        Persist Security Info=False;
        User ID=[user id];
        password=[password];
        Initial Catalog=[Database Name];
        Data Source=[Server Name]";
}

protected string ExecuteQuery( OleDbCommand QueryObject)
{
    this.sellerAdminConn.Open();
    int queryResult = QueryObject.ExecuteNonQuery();
    if ( queryResult != 0)
    {
        this.sellerAdminConn.Close();
        return "Success";
    }
    else
    {
        return "error: QueryResult= " + queryResult;
    }
}

[ WebMethod(Description="Adds a new book to the books
            table", EnableSession=false)]
public string addItem(string ISBN,string author,
            double price, string title,string description,
```

```

        string imagePath, int CAT_ID)
    {
        try
        {
            this.init();
            OleDbCommand addItem =
                new OleDbCommand(
                    "AdminAddBook",
                    this.sellerAdminConn);
            addItem.CommandType =
                CommandType.StoredProcedure;

            OleDbParameter addISBN =
                addItem.Parameters.Add(
                    "@BK_ISBN",OleDbType.Char,15);
            addISBN.Value = ISBN;

            OleDbParameter addAuthor = addItem.Parameters.Add(
                "@BK_Author",OleDbType.Char,80);
            addAuthor.Value = author;

            OleDbParameter addPrice = addItem.Parameters.Add(
                "@BK_Price",OleDbType.Currency,8);
            addPrice.Value = price;

            OleDbParameter addTitle = addItem.Parameters.Add(
                "@BK_Title",OleDbType.Char,75);
            addTitle.Value = title;

            OleDbParameter addDescription =addItem.Parameters.Add(
                "@BK_Description",OleDbType.Char,255);
            addDescription.Value = description;

            OleDbParameter addImage = addItem.Parameters.Add(
                "@BK_ImagePath",OleDbType.Char,50);
            addImage.Value = imagePath;

```

```
        OleDbParameter addCatId = addItem.Parameters.Add(
            @"CAT_ID",OleDbType.Integer,4);
        addCatId.Value = CAT_ID;

        return this.ExecuteNonQuery( addItem );
    }
    catch(Exception e)
    {
        return e.ToString();
    }
}
.
.
.
```

In this section, you created the *sellerAdmin* Web Service and the *addItem* Web Service Method. In the next section, we will look at how to test the Web Service and its methods.

## Testing a Web Service in ASP.Net

We can test our service by performing the following steps:

1. In VS.NET right-click the file **sellerAdmin.asmx**, and select **Set as start page**.
2. Press **F5** to run it. This will take a few seconds to compile and run.
3. When the browser loads, you should see something like Figure 6.15.
4. To test the service *addItem*, click the **addItem** link. An input form will be displayed, prompting you for values for its parameters. See Figure 6.16.
5. Fill in the appropriate textboxes and click **Invoke**.
6. Since this service returns a datatype *string*, we should see something like Figure 6.17.



Figure 6.15 Web Service Listing

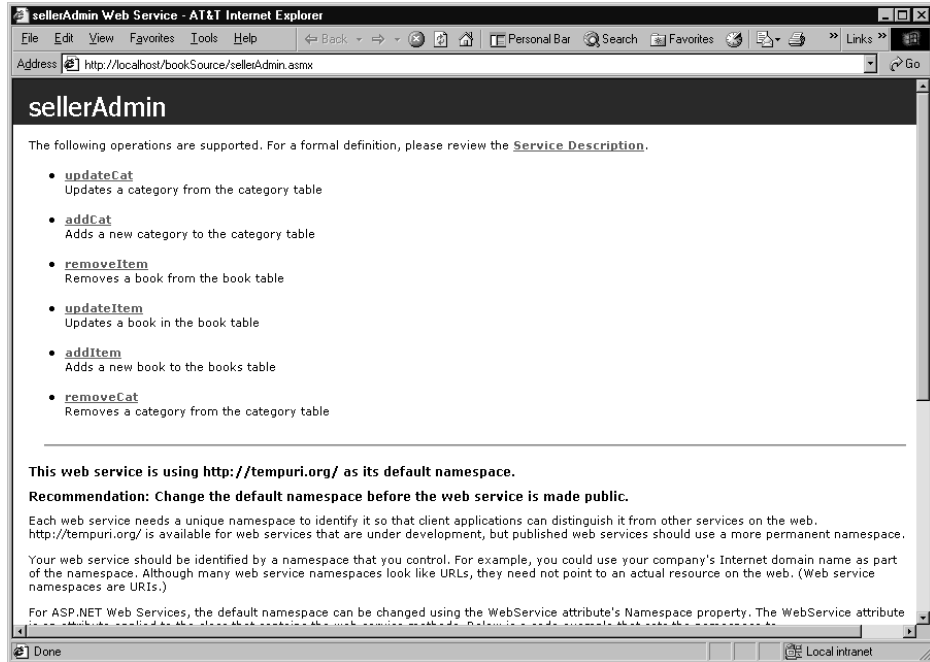
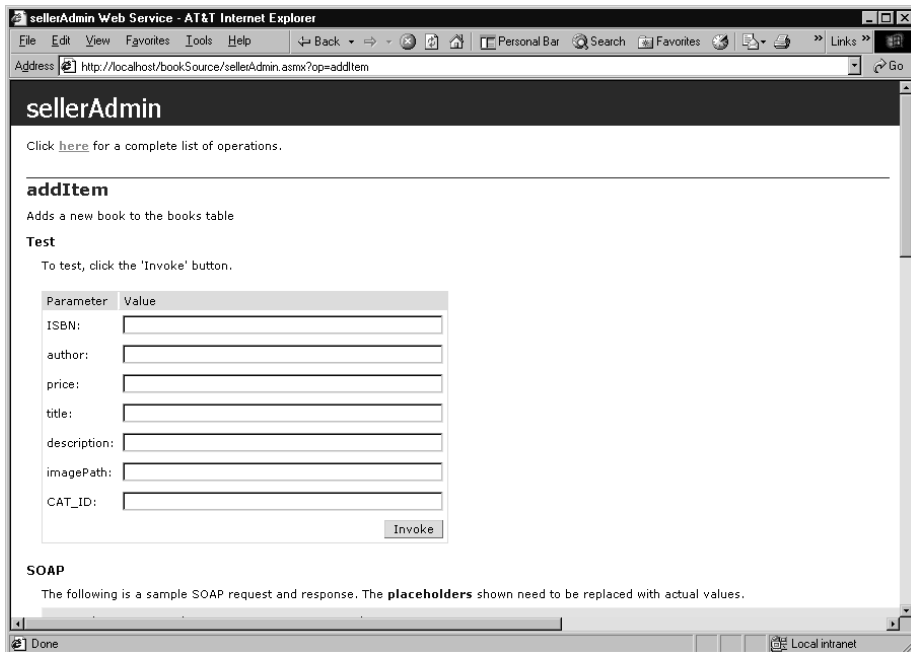
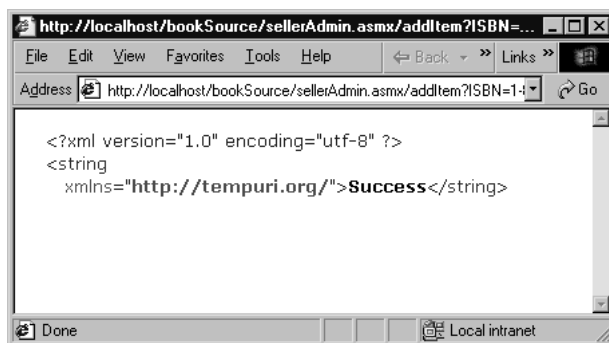


Figure 6.16 Testing a Web Service



**Figure 6.17** Results of Invoking the *addItem* Web Service

This shows that the method has completed successfully and returned the corresponding output. These steps can be repeated for each of the remaining methods: *removeItem*, *updateItem*, *addCat*, *removeCat*, and *updateCat*. Each of these methods is coupled with a corresponding stored procedure (MSSQL) or parameterized query (MS Access).

The following is a function prototype overview of the process-flow or steps involved in creating each of these Web methods. See if you can create and test these Web methods on your own, then compare them to the source code available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). The *sellerAdmin* Web service and all of its methods can also be found on the Solutions Web site in the file *sellerAdmin.asmx.cs*.

- ***removeItem*** (*int isbn*) Removes a book item from the database
  1. Call *init()*.
  2. Create *Command* object accessing the *AdminRemoveBook* stored procedure.
  3. Create the *Parameter* object and assign its value.
  4. Execute the procedure. Call *ExecuteQuery( commandObj )*.
  5. Return string indicating success or failure of the operation.
- ***updateItem*** (*string ISBN, string author, double price, string title, string description, string imagePath, int CAT\_ID*) Updates a book item's information
  1. Call *init()*.
  2. Create *Command* object accessing the *AdminUpdateBook* stored procedure.
  3. Create the *Parameter* objects and assign their values.

4. Execute the procedure. Call *ExecuteQuery( commandObj )*.
  5. Return string indicating success or failure of the operation.
- ***addCat*** (*string CAT\_Name*) Adds a category name to the database
    1. Call *init()*.
    2. Create *Command* object accessing the *AdminAddCat* stored procedure.
    3. Create the *Parameter* object and assign its value.
    4. Execute the procedure. Call *ExecuteQuery( commandObj )*.
    5. Return string indicating success or failure of the operation.
  - ***updateCat*** (*int CAT\_ID, string CAT\_Name*) Updates category details
    1. Call *init()*.
    2. Create *Command* object accessing the *AdminUpdateCat* stored procedure.
    3. Create the *Parameter* objects and assign their values.
    4. Execute the procedure. Call *ExecuteQuery( commandObj )*.
    5. Return string indicating success or failure of the operation.
  - ***removeCat*** (*int CAT\_ID*) Removes a category from the database
    1. Call *init()*.
    2. Create *Command* object accessing the *AdminUpdateCat* stored procedure.
    3. Create the *Parameter* object and assign its value.
    4. Execute the procedure. Call *ExecuteQuery( commandObj )*.
    5. Return string indicating success or failure of the operation.

## NOTE

This application contains several different Web Services. The code for these Web Services can be found on the Solutions Web site for the book at [www.syngress.com/solutions](http://www.syngress.com/solutions). In the directory for Chapter 6 refer to the following files *adminCustomer.aspx.cs*, *sellerAdmin.aspx.cs*, *getBooks.aspx.cs*, *getCategories.aspx.cs*, *getCustomer.aspx.cs*, *loginCustomer.aspx.cs*, *orderBooks.aspx.cs*, and *sellerAdmin.aspx.cs*.

Now that you know the Web Service and its methods are working correctly, the next step will be to create our UI for the Web application and generate proxy classes for it to retrieve data from our Web Services. In the next section, we will see how VS.NET works with WSDL and Universal Description, Discovery, and Integration (UDDI) to enable our ASP.NET Web Application to connect to and retrieve data from our *booksource* Web Service project.

## Using WSDL Web References

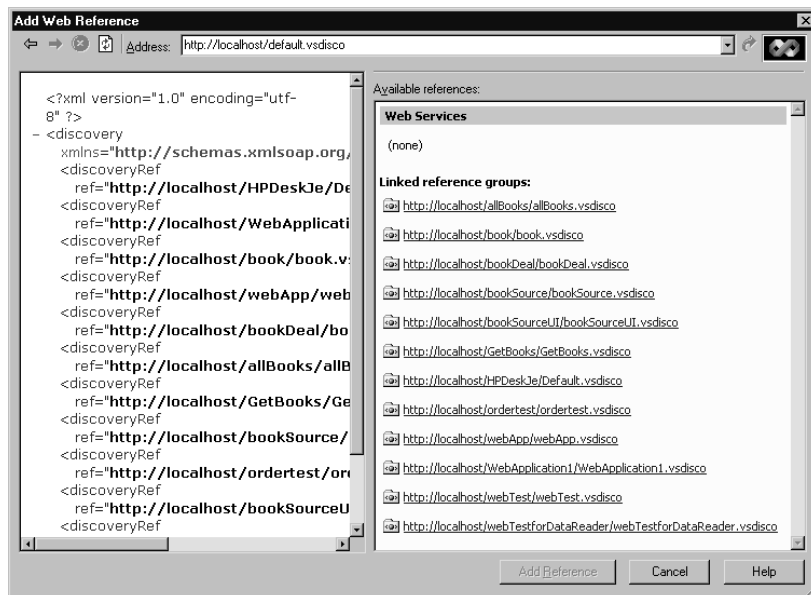
We will use WSDL and DISCO in our Web application project to connect to and add a reference to our Web Services Application (*bookSource*) and its individual Web Services and their Web methods.

Let's create a new C# Web application, named *bookSourceUI*.

The first thing you want to do is create a reference to your Web Services so that you can easily access the methods in our code.

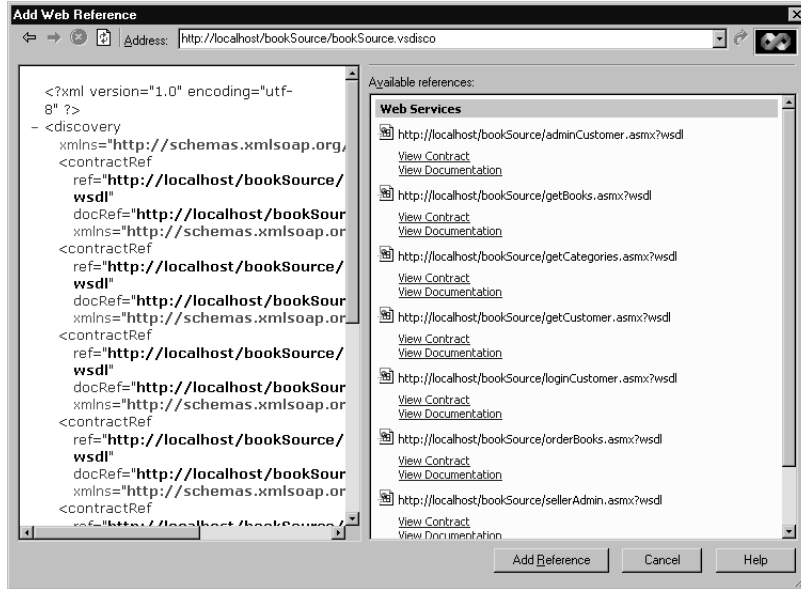
1. In the **Solution Explorer** pane, right-click **Web References**.
2. Select **Add Web Reference**. A new dialog will appear.
3. Select the last UDDI option, which is your local machine. VS.NET will check your server for all Web Services. It will then present you with a list of services you can view or select to add a reference to. See Figure 6.18.

**Figure 6.18** UDDI Server Discovery Dialog



4. Select the service group you would like to add a reference to. Look for your Web Service project name (`http://localhost/bookSource.vsdisco`).
5. The Services available will be displayed. See Figure 6.19.

**Figure 6.19** Services Available



6. You can view the Simple Object Access Protocol (SOAP) contracts and documentation for each Service Method by clicking on the link. Be sure to add the reference from this level in the menu. To add this Web Service and all its methods, click **Add Reference**. VS.NET will create proxy classes for each Service Method so that the method can be accessed just like a local class method. See Figure 6.20.

## Building the Site

Now that the back-end database interfaces and Web Services have been completed, you should turn your focus to the middle tier data classes and controls that act as a bridge between the backend and the Web UI. The site structure will look something like that depicted in Figure 6.21.

Figure 6.20 Proxy Classes Added to Solution Explorer in VS.NET UI

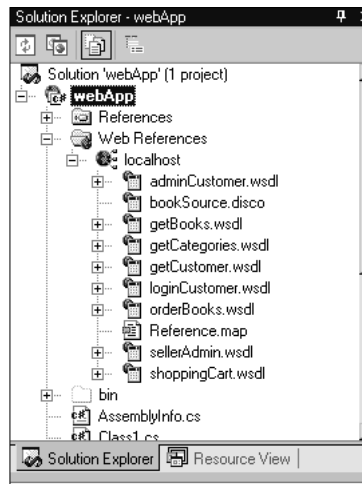
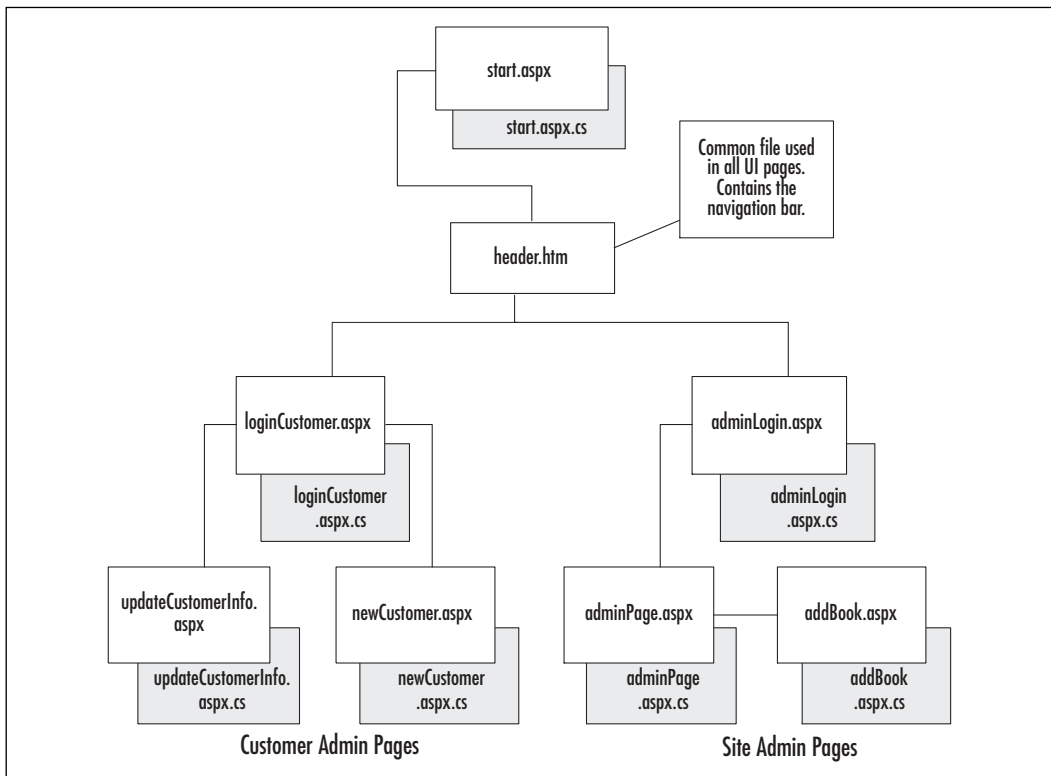


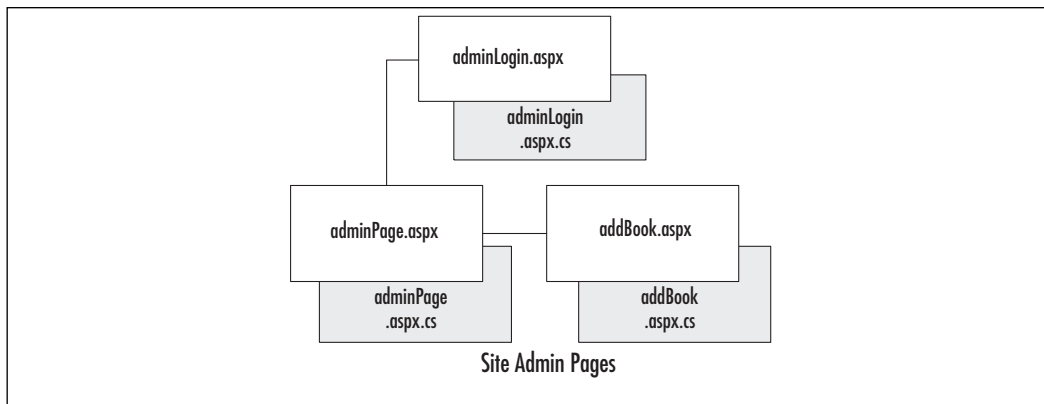
Figure 6.21 BookShop Site Overview



## Site Administration

In this section, we will develop the code that allows us to tie our site administration interface to our Web Services (see Figure 6.22). While creating the pages needed, we will cover creating the Administration login, creating the Administration page, and an *addBook* page for the administrator.

**Figure 6.22** Site Administration Page Group Overview



### Creating the Administration Login (adminLogin.aspx)

This is a fairly simple page that uses the *RequiredFieldValidator* server control.

There are several server controls that enable HTML form validation:

- *RequiredFieldValidator*
- *CompareValidator*
- *RangeValidator*
- *RegularExpressionValidator*
- *CustomValidator*
- *ValidationSummary*

All of these controls work in a similar fashion. In this example page, we use *RequiredFieldValidator* in a code behind page to show how to use a server control to validate user data in HTML forms.

1. In the Web application **bookSourceUI**, create a new aspx page, and name it **adminLogin.aspx**.

2. In **Design view**, drag and drop a **RequiredFieldValidator**.
3. Be sure not to position this element in Design view; in the **aspx page**, remove the style attribute from the element and use HTML layout techniques to position it. (See the sidebar in this section on ASP.NET and Netscape.)

Let's look at the code from the aspx page:

```
<tr>
  <td><FONT face="Verdana" size="2">User:</FONT>&nbsp;</td>
  <td style="WIDTH: 127px">
    <asp:textbox id="txtUser" runat="server"
      Width="106px" Height="24px">
    </asp:textbox>
  </td><td>
    <asp:requiredfieldvalidator id="passUser" runat="server"
      ErrorMessage="You must supply a user name"
      ControlToValidate="txtUser" Width="121px" Height="57px">
    </asp:requiredfieldvalidator>
  </td>
</tr>
```

Lets look at a code snippet from the code-behind file (the aspx.cs page). When we drag the RequiredFieldValidator onto the page, VS.NET will add the following:

```
protected System.Web.UI.WebControls.RequiredFieldValidator passUValid;
```

And that's all there is to it. When the page is run, a reference is made to a client-side JavaScript file that includes crossbrowser code to ensure that this field contains a value before allowing a *submit*. If the user tries to submit without filling in the text box, the error message "You must supply a user name" will appear in the table cell to the right of the text box (it actually appears wherever the **asp:requiredfieldvalidator** tag is placed in the HTML, in this case an adjacent table cell). Next, we will look at the admin page itself.



## Debugging...

### ASP.NET Server Controls Do Not Display Correctly in Netscape 4.x

A lot has happened over the last few years with Netscape and the open source Mozilla project. While the newer versions of Mozilla version .094 and later should handle this fine, there is still a significant Netscape 4.x user base. When we develop Web front-ends for our clients, we strive to ensure at least Netscape 4.72 will display and function correctly.

What's the issue? It seems that most of the examples showing you how to use server controls have you drag and drop the control to where you want it on the screen. In HTML, this creates span tags with inline style attributes containing "absolute positioning." Those of us that have dealt with cross-browser Dynamic HTML (DHTML) issues know that this can cause problems in Netscape. The solution: Use *FlowLayout* and good old-fashioned HTML elements and tricks for positioning. To do this, simply right-click a page in either **Design** or **HTML view** and switch the *pageLayout* property to *FlowLayout*.



## Creating the Administrator Page (adminPage.aspx)

The purpose of this page is to allow the site administrator the ability to remove and update book item information. In the following sections, we'll look specifically at retrieving the data, displaying the data, adding new books to the database, deleting books, and updating book details.

### Retrieving the Data: Creating the *getBooks.AllBooks* Web Method

To retrieve the list of books stored in the database, we will need to access the *GetAllBooks* stored procedure (MSSQL) or parameterized query (MS Access). We will do this by creating the *allBooks* method of the *getBooks* Web Service. This method will take no parameters, and will return a *DataSet* containing all Book data as well as the table structure of the database table that the data originated from.

The Web method *getBooks.AllBooks* can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the file *getBooks.asmx.cs*.

1. To create this method, we must first create a new Web Service named *getBooks*. (See the section on Web Services earlier in this chapter.)
2. Inside the code-behind page of *getBooks* (*getbooks.aspx.cs*), we need to create the method *allBooks*. *AllBooks* should return a *DataSet*:

```
public DataSet AllBooks()
```

3. Set the connection string:

```
string source = "Provider=SQLOLEDB.1;Persist Security Info=False ...
```

4. Create the *Connection* object:

```
OleDbConnection conn = new OleDbConnection ( source ) ;
```

5. Create the *Command* object accessing the *GetAllBooks* stored procedure:

```
OleDbCommand cmd = new OleDbCommand ( "GetAllBooks" , conn ) ;  
cmd.CommandType = CommandType.StoredProcedure;
```

6. Create a *DataAdapter* object for the *Command* object:

```
OleDbDataAdapter da = new OleDbDataAdapter (cmd) ;
```

7. Create a new *DataSet* and use the *DataAdapter* to fill it from the results of executing the stored procedure:

```
DataSet ds = new DataSet ( ) ;  
da.Fill ( ds , "Books" ) ;
```

8. Close the connection and return the *DataSet*:

```
conn.Close();  
return ds;
```

Here is the method in its entirety:

```
[WebMethod(Description="This will return all books in an XML String",  
    EnableSession=false)]  
public DataSet AllBooks()  
{  
    string source = "Provider=SQLOLEDB.1;Persist Security  
        Info=False;User ID=[userID];password = [password];  
        Initial Catalog=[database name];  
        Data Source=[server name];Use Procedure for Prepare=1;
```

```

        Auto Translate=True;Packet Size=4096;
OleDbConnection conn = new OleDbConnection( source );
conn.Open ( ) ;
OleDbCommand cmd = new OleDbCommand ( "GetAllBooks" , conn);
cmd.CommandType = CommandType.StoredProcedure;
OleDbDataAdapter da = new OleDbDataAdapter (cmd) ;
DataSet ds = new DataSet ( ) ;
da.Fill ( ds , "Books" ) ;
conn.Close();
return ds;
}

```

The data returned contains an embedded XSD schema describing the Database table *Books*.

```

<?xml version="1.0" encoding="utf-8"?>
<DataSet xmlns="http://tempuri.org/">
<xsd:schema id="NewDataSet" targetNamespace=""
xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xsd:element name="NewDataSet" msdata:IsDataSet="true">
    <xsd:complexType>
      <xsd:choice maxOccurs="unbounded">
        <xsd:element name="Books">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="isbn" type="xsd:string" minOccurs="0" />
              <xsd:element name="name" type="xsd:string" minOccurs="0" />
              <xsd:element name="id" type="xsd:int" minOccurs="0" />
              <xsd:element name="imgSrc" type="xsd:string" minOccurs="0" />
              <xsd:element name="author" type="xsd:string" minOccurs="0" />
              <xsd:element name="price" type="xsd:decimal" minOccurs="0" />
              <xsd:element name="title" type="xsd:string" minOccurs="0" />
              <xsd:element name="description" type="xsd:string"
                minOccurs="0" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</DataSet>

```

```

    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

The next section is the diffgram node, which contains all the table records:

```

<diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <NewDataSet xmlns="">
    <Books diffgr:id="Books1" msdata:rowOrder="0">
      <isbn>0072121599</isbn>
      <name>cisco</name>
      <id>2</id>
      <imgSrc>ccda.gif</imgSrc>
      <author>Syngress Media Inc</author>
      <price>49.99</price>
      <title>Ccda Cisco Certified Design Associate Study Guide</title>
      <description>Written for professionals intending on taking the CCDA
        test, this special guide covers all the basics of the test and
        includes hundreds of test questions on the enclosed CD.
      </description>
    </Books>
    <Books diffgr:id="Books2" msdata:rowOrder="1">
      <isbn>0072126671</isbn>
      <name>cisco</name>
      <id>2</id>
      <imgSrc>ccna.gif</imgSrc>
      <author>Cisco Certified Internetwork Expert Prog</author>
      <price>49.99</price>
      <title>CCNA Cisco Certified Network Associate Study Guide</title>
      <description>Cisco certification courses are among the fastest-
        growing courses in the training industry, and our guides are
        designed to help readers thoroughly prepare for the exams.
      </description>
    </Books>.
  
```

This XML file is interpreted by ASP.NET as a *DataSet* object and can be easily loaded into any variable of type *DataSet*.

The *DataGrid* control is designed to be *DataBound* to a *DataSet* object. This makes it easy to “data bind” to a Web Service Method that returns a *DataSet*. Data Binding a *DataSet* to the *DataGrid* is almost the same as loading the *DataSet* into the *DataGrid*. The *DataGrid* is then able to iterate through and perform operations on the *DataSet* as if it were an Access Form connected to an Access database. The *DataSet* in actuality is an in-memory XML representation of the database including the *Books* table.

## Displaying the Data: Binding a *DataGrid* to the *DataSet*

The *DataGrid* is actually bound to the *DataTable Books* which is a table within the *DataSet* returned by *getBooks.AllBooks*. We create a *DataView* of the *Books* table so that we can sort the data. This *DataView* is then bound to the *DataGrid*.

In the following code, *changeBooks* is the name of our *DataGrid* object:

```
Dt = Books.AllBooks().Tables["Books"];
myView = new DataView(Dt);
myView.Sort = "isbn";
    changeBooks.DataSource = myView;
    changeBooks.DataBind();
```

## Adding New Books to the Database: Creating the *allBooks.addItem* Web Method

The creation of this method was shown as an example earlier in the chapter, under the section “Web Services.”

## Deleting Books: Deleting from the *DataGrid* and the Database

Using the *DataGrid* event *changeBooks\_DeleteCommand*, fired when a user clicks the **Delete** button in the *DataGrid* UI, we will select the row in the *DataGrid* to remove by using the *RowFilter* property.

The following code selects the individual book by performing a filter on ISBN. It is analogous to the SQL statement:

```
Select * from Books where isbn = "@isbn"
```

The equivalent code for the *DataView* is:

```
myView.RowFilter = "isbn='"+upISBN+"'";
```

This will return an array or collection of items. Since ISBN is our primary key in the Books table, we know that this filter will return only one item. We delete this row from the *DataView* by simply calling the *Delete* method:

```
myView.Delete(0);
```

Next, we reset the filter so we can re-access the entire Books table:

```
myView.RowFilter = "";
```

Now we need to resync the *DataGrid* with the in-memory Books Table View so that the *DataGrid UI* reflects the change:

```
changeBooks.DataSource = myView;  
changeBooks.DataBind();
```

Next, we need to update the database to sync it with the *DataGrid*. This is accomplished by calling the Web method and passing it the ISBN of the book to delete:

```
removeBook.removeItem(upISBN);
```

## Updating Book Details: Updating the *DataGrid* and the Database

Using the *DataGrid* event *changeBooks\_UpdateCommand*, fired when a user clicks the **Update** button in the *DataGrid UI*, we will select the row in the *DataGrid* to update by using the *RowFilter* property.

1. Select the row to update by using the *RowFilter* property of the *DataView* (see the example in the preceding section).
2. Create a new *DataRow* Item and populate it with the changes (new Data). Store updated column values in local variables:

```
string upISBN = e.Item.Cells[2].Text;  
string upAuthor = ((TextBox)e.Item.Cells[3].Controls[0]).Text;  
double upPrice =  
double.Parse(((TextBox)e.Item.Cells[4].Controls[0]).Text);  
string upTitle = ((TextBox)e.Item.Cells[5].Controls[0]).Text;  
string upDescription = ((TextBox)e.Item.Cells[6].Controls[0]).Text;
```

```
int upCatId = int.Parse(e.Item.Cells[7].Text);
string upImage = ((TextBox)e.Item.Cells[8].Controls[0]).Text;
```

3. Delete the row that is being updated (see the example in the preceding section).
4. Create a *new DataRow* and populate it with the new data.

```
DataRow dr = Dt.NewRow();
dr["isbn"] = upISBN;
dr["author"] = upAuthor;
dr["price"] = upPrice;
dr["title"] = upTitle;
dr["description"] = upDescription;
dr["id"] = upCatId;
dr["imgSrc"] = upImage;
```

Insert the new *DataRow*:

```
Dt.Rows.Add(dr);
```

5. Resync the *DataGrid* with the *DataView* (see the example in the preceding section)

To update the database, simply call the Web method *sellerAdmin.updateItem*, passing it the new data.

```
localhost.sellerAdmin newData = new localhost.sellerAdmin();
newData.updateItem(upISBN, upAuthor, upPrice, upTitle, upDescription,
                  upImage, upCatId);
```

One limitation of the *DataGrid* is that it doesn't provide a UI for adding new records. We will handle this case by creating another page: *addBook.aspx*.

## Creating the addBook Page (addBook.aspx)

*AddBook* is another fairly straightforward page. It provides a UI where the site administrator can fill out a simple HTML form and submits. This data is handled by the code-behind page *addBook.aspx.cs*. This page simply passes the data to the database via the Web method *sellerAdmin.addBook*:

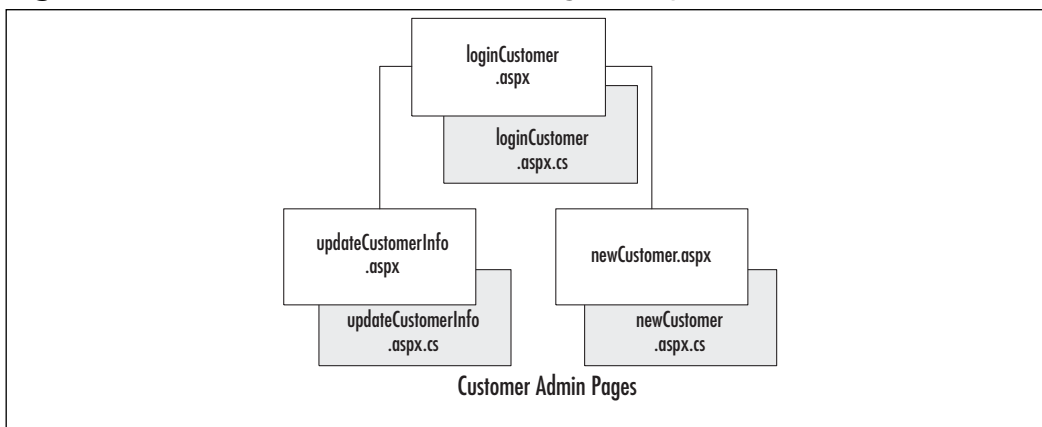
```
addNewBook = new localhost.sellerAdmin();
```

```
resultAdd = addNewBook.addItem(addISBN,addAuthor,addPrice,addTitle,  
    addDescription,addPath,addCatId);
```

## Customer Administration

In this section, we will develop the code that allows us to tie our customer administration interface to our Web Services (see Figure 6.23).

**Figure 6.23** Customer Administration Page Group Overview



## Creating the Customer Admin Section

This section of the site deals with user authentication, including creating a customer account and login. We use this to simulate order processing.

### Creating the *loginCustomer* Page

We will use the same form layout as we did for the admin login described in the preceding section. One change we'll implement is that we'll call a Web Service to verify the login of the customer.

1. Make a call to the Web Service *loginCustomer*. This should be routine by now, but let's look at the code to call the Web Service:

```
loggedCust = new WebReference1.loginCustomer();
```

2. Access the Web method *validCustomer*. Now you have access to all the methods contained in the class.

```
string resultId =  
loggedCust.validCustomer(validEmail,validPassword);
```



3. Return a value. You can now check the value of the variable *resultId* and either grant the customer access or return an error message.

```

if(resultId == "-1")
{
    loginLabel.Text = "Invalid Login please re-enter your password and
        email!";

}
else
{
    loginLabel.Text = "Welcome";
    Session["userId"] = int.Parse(resultId);
    Server.Transfer((string)Session["return2Page"]);
}

```

Now you have the customer logged in to the site and they can go to any page without having to sign in again.

## NOTE

We are using a session variable to track where the user is coming from when they are prompted to login. This will enable us to redirect them back to the page where they came from rather than sending them to some non-specific page and having them navigate through the site from scratch.

## Creating the updateCustomerInfo Page

We can now add a page that will let the customer update his or her information. This will be done identically to the example from site admin where we brought in all books and then enabled the site administrator to go through the books listed and delete, update, or add books at will. In this case, we will enable the customer to update only.

1. Select the row to update by using the *RowFilter* property of the *DataView*.
2. Create a new *DataRow* Item and populate it with the changes (new Data).

3. Delete the row that is being updated.
4. Insert the new *DataRow*.
5. Resync the *DataGrid* with the *DataView*.

All five steps are the same as covered in earlier examples. Let's look at the code one more time:

```
Dt = Customers.AllCustById((int)Session["userId"]).Tables["Customers"];
    myView = new DataView(Dt);
    myView.Sort = "CT_ID";
```

Set the *DataTable* value into the *DataView*:

```
custGrid.DataSource = myView;
custGrid.DataBind();
```

Set the data source of *DataGrid*:

```
myView.RowFilter = "CT_ID='"+upId+"'";
    if (myView.Count > 0)
    {
        myView.Delete(0);
    }
    myView.RowFilter = "";

    DataRow dr = Dt.NewRow();
    dr[0] = upId;
    dr[1] = upFName;
    dr[2] = upLName;
    dr[3] = upEmail;
    dr[4] = upPassword;
    Dt.Rows.Add(dr);
```

Delete the bad data row and the new one:

```
WebReference1.adminCustomer newData = new
WebReference1.adminCustomer();

    newData.updateCust(upId, upFName, upLName, upEmail, upPassword);
```

Lastly, update the database by calling the Web service.

In the previous examples we have made extensive use of the *DataGrid* control for *DataBinding DataSet* information to the UI. We must admit we were a bit reluctant to use the *DataGrid* since it seemed reminiscent to the *DataGrid* Design Time Controls (DTCs). DTCs were included with many versions of FrontPage, Visual InterDev, and Office. They made it easy for novice developers to quickly create data driven Web sites. Lets just say DTCs had some drawbacks, to put it politely! In the next two sections, ADOCatalog and XMLCart, we will use XSL/Transforms against XML data to produce our UI. This is accomplished by using the asp:xml server control as well as client side script and hidden asp:text controls. The ADOCatalog's primary interfaces will return *DataSet* objects so it could be easily tied to a *DataGrid* control. We will leave that as an exercise for you. The XMLCart is primarily a wrapper class around the *XmlDocument* object. Its primary interfaces will return *XmlDocument* objects.

## Creating an ADOCatalog

In this section, we will develop the code that allows us to tie our catalog interface to our Web Services. We will store a *DataSet* in an *Application* variable to reduce the load on the database, perform copy, clone, import, create, and filter operations on ADO.NET *DataSet* objects, and use XML and Extensible Stylesheet Language Transformations (XSLT) to render data stored in a *DataSet* as HTML via the asp:xml server control.

In our ADOCart application, all database interaction is handled via Web Services. Since our *Books* data is fairly static, we can retrieve the data in a *DataSet* once and store that *DataSet* in an application-level variable. This reduces the database traffic, while still providing quick access to the data.

Here is an overview of the process we will be following:

- Load all Books data to an *application* variable:
 

```
Application["AllBooks"];
```
- Create an instance of ADOCatalog (a.k.a., *BookCatalog*).
 

```
In Page_onload
```
- Initialize the instance by passing it.
 

```
(DataSet)Application["AllBooks"];
```
- Call *BookCatalog.CatalogRange(0,5)* to return the first five books.

- Convert return data to XML.
- Load XSLT.
- Set *Document* and *Transform* properties of the asp:xml control.

Now, let's create the code:

To store our data in an application object, open the `global.asax` file. Add this to the `Application_onstart` method:

```
localhost.getBooks DataSource = new localhost.getBooks();  
Application["AllBooks"] = DataSource.AllBooks();//DataSet
```

This will create an instance of the `getBooks` object called `DataSource`. Using this instance, we call the `AllBooks` method, which returns a `DataSet`. We then save the `DataSet` in an application-level variable, `allbooks`.

## NOTE

---

`localhost` is a reference to the name of the Web Reference containing the `getBooks` Web Service proxy (`getBooks.wsdl`).

---

Now add a new page to the Web Application project (`bookSourceUI`). Name it **start.aspx**.

Below the `#endregion` section in the `WebForm1` class, we will create a new class called `bookCatalog`.

## Creating the *BookCatalog* Class

The `BookCatalog` class will contain the following public methods: `InitCatalog`, `Catalog`, `CatalogItemDetails`, `CatalogRange`, `CatalogByCategory`, and the private methods `CatalogRangeByCategory`, and `CreateSummaryTable`. The following is a rough prototype of the `ADOCatalog` class that we'll be building in this section:

```
public class bookCatalog  
{  
    protected WebReference1.getBooks DataSource;  
    protected DataSet dsAllBooks;  
    protected DataTable dtSummary;  
  
    protected DataTable createSummaryTable(  

```

```

        int startPos, int range, int RecordCount)
public DataSet catalog()
public void initCatalog(DataSet ds )
public DataSet catalogItemDetails( string book_isbn )
public DataSet catalogRange(int startPos, int range)
public DataSet catalogByCategory( int catId)
protected DataSet catalogRangeByCategory(
    int startPos, int range, int catId, string book_isbn)
}

```

## Creating the *CreateSummaryTable* Method

The *CreateSummaryTable* method creates a *DataTable* that contains summary information about the *DataSet* being returned. This data is used by the XSLT to display Metadata (i.e., viewing records 6 thru 12 of 25). It is also useful when making a fetch next range of records call.

Based on the prototype, this method will take the parameters *int startPos*, *int range*, and *int RecordCount* and will return a *DataTable*. Let's get started.

1. Create a new empty *DataTable* named "Summary".

```
DataTable dtSummary = new DataTable("Summary");
```

In the XSD schema this makes the *DataTables* parent element a *summary* tag (i.e. <summary> )

2. Now add the *Columns RecordCount*, *FirstItemIndex*, and *LastItemIndex* to the *Summary DataTable*.

```
dtSummary.Columns.Add(
    new DataColumn("RecordCount", typeof(int)));
dtSummary.Columns.Add(
    new DataColumn("FirstItemIndex", typeof(int)));
dtSummary.Columns.Add(
    new DataColumn("LastItemIndex", typeof(int)));
```

3. Create a new *DataRow* object and assign it to a new *DataTable Row*.

```
DataRow drSummary;
drSummary = dtSummary.NewRow();
```

4. Populate the *DataRow* object and add it to the *DataTable*.

```
drSummary["RecordCount"]    = RecordCount;
drSummary["FirstItemIndex"] = startPos;
drSummary["LastItemIndex"]  = startPos + range;
dtSummary.Rows.Add( drSummary );
```

5. Return the new *DataTable*.

```
return dtSummary;
```

## Creating the *InitCatalog* Method

The *InitCatalog* method loads a *DataSet* into the *BookCatalog* object, then adds a default summary table to the private *DataSet* *dsAllBooks*.

Based on the prototype, this method will take the only parameter, a *DataSet*, and will return nothing.

```
public void initCatalog(DataSet ds )
{
    dsAllBooks = ds;
    int recordCount = dsAllBooks.Tables[0].Rows.Count;
    dsAllBooks.Tables.Add(
        createSummaryTable(0, recordCount-1, recordCount) );
}
```

## Creating the *Catalog* Method

The *Catalog* method returns the entire *DataSet* stored in the private variable *dsAllBooks*:

```
public DataSet catalog()
{
    return dsAllBooks;
}
```

## Creating the *catalogItemDetails*, *catalogRange*, and *catalogByCategory* Methods

The three methods, *catalogItemDetails*, *catalogRange*, and *catalogByCategory*, are specialized cases of *catalogRangeByCategory* and are really only logical interfaces to obtain desired result sets.

The method *catalogItemDetails* will return all data corresponding with the given ID (*Book\_isbn*):

```
public DataSet catalogItemDetails( string book_isbn )
{
    // returns a DataSet containing a single book
    return catalogRangeByCategory( -1, -1, -1, book_isbn);
}
```

The method *catalogRange* will return all data for items in a given range:

```
public DataSet catalogRange(int startPos, int range)
{
    //returns a given range of books
    return catalogRangeByCategory( startPos, range, -1, null);
}
```

The method *catalogByCategory* will return all data for items in a given category:

```
public DataSet catalogByCategory( int catId)
{
    //returns all books with the given categoryId
    return catalogRangeByCategory( -1, -1, catId, null);
}
```

## Creating the *catalogRangeByCategory* Method

The *catalogRangeByCategory* method creates a new *DataSet* containing a new *Books* Table, appends the appropriate *Summary* Table, and returns this new *DataSet*. It is used by the preceding methods to return a single item's node (to add to the shopping cart), to return a range of books (to handle browsing the catalog), and to return all books in a given category (to handle viewing by category). A method could easily be added that enables browsing by category.

In order to return a subset of the *DataSet AllBooks*, we need to create a new *DataTable* object that has the same table structure as *Books*. We can then import rows that meet our criteria into this new table. When the table is filled, we create a new *DataSet* object and add the new *DataTable* as well as a *Summary* Table. The resulting *DataSet* will contain the request subset of data and some meta-information (supplied by the *Summary* table).

Now, let's examine the code. Create a temporary *DataTable* that holds *allBooks* data:

```
DataTable dtTemp = dsAllBooks.Tables["Books"];
```

Clone the structure of this table in a new *DataTable*:

```
DataTable dtBooks = dtTemp.Clone();//create Empty Books Table
```

Set the filter expression property based on input parameters:

```
if( catId == -1)
{ //no filter is applied strExpr = "";
}
else
{ //select only one category
  strExpr = "id='" + catId + "'";
}
if( book_isbn != null)
{ //return a single item
  strExpr = "isbn='" + book_isbn + "'";
}
```

Set the Data filter to affect all current rows, sort by title, and apply the filter expression:

```
strSort ="title";
recState = DataViewRowState.CurrentRows;
foundRows = dtTemp.Select(strExpr, strSort, recState);
RecordCount = foundRows.Length;
```

Add *foundRows* to the *DataTable dtBooks*:

```
for(int i = startPos; i < endPos; i ++)
{
  dtBooks.ImportRow( (DataRow)foundRows[i] );
}
```

Add the *DataTable dtBooks* to the new *DataSet* along with *DataTable Summary*, then return this new *DataSet*:

```
dsBookRange = new DataSet();
dsBookRange.Tables.Add( dtBooks );
dsBookRange.Tables.Add(
  createSummaryTable( startPos, range, RecordCount ) );
return dsBookRange;
```



On page load, we will instantiate the object, retrieve *Application*["AllBooks"], return the requested subset *DataSet* object, convert it to XML using the *GetXml()* method of the *DataSet* object, and apply an XSL/Transform to render the Catalog in the UI.

In order to enable browsing, we will store the *FirstRecord*, *LastRecord*, *recordCount*, and user action (previous | next | by *CategoryID*) into hidden Text fields on the client, so this data can be read to determine which *bookCatalog* method to call and with which parameters to return the desired subset of *AllBooks*.

You can see the code on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) for a closer look at how to implement this class (see *start.aspx* and *start.aspx.cs*). The Solutions Web site also contains the XSLT used to render the UI (*Catalog.xslt*).

## Building an XMLCart

In this section, we will develop the code that allows us to tie our catalog to the shopping cart. We will use XML node operations to update our cart's contents, XSLT/XPath operations to calculate cart totals and taxes, XML and XSLT to render cart data as HTML, and the *asp:XML* server control to process transforms. The code for this class can be found on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the files *start.aspx* and *start.aspx.cs*.

The XMLCart is really a wrapper class around common XML functions. It performs the following basic operations: load data, add new item, remove item, and empty cart.

Looking at the class, you'll see there really isn't much to it.

```
public class xmlShoppingCart
{
    protected XmlDocument myCart;
    public void initCart( string dataSource )
    {
        myCart = new XmlDocument();
        if( dataSource != null )
        {
            myCart.LoadXml( dataSource );
        }
        else
        {
```

```

        myCart.LoadXml( "<shopcart-items></shopcart-items>" );
    }
}

public string addItem2Cart( XmlDocument book )
{
    try
    {
        //Import the last book node from doc2 into the original document.
        XmlNode newBook =
            myCart.ImportNode(book.DocumentElement.FirstChild, true);
        myCart.DocumentElement.AppendChild(newBook);
        return "Success";
    }
    catch(Exception e) {
        return e.ToString();
    }
}

public string removeItemFromCart( string isbn )
{
    XmlNode curnode =
        myCart.SelectSingleNode("//Books[isbn='" + isbn + "']");
    try
    {
        myCart.DocumentElement.RemoveChild( curnode );
        return "Success";
    }
    catch(Exception e)
    {
        return e.ToString();
    }
}

public void clearCart()
{
    XmlElement root = myCart.DocumentElement;
    root.RemoveAll();
}
}

```

```

public XmlDocument getCartDescription()
{
    return myCart;
}
public string getCartDescriptionString()
{
    return myCart.OuterXml;
}
}

```

When the page loads, the cart must be initialized. This is handled with the *init* method. If there is no data to load into the cart, the root node (*<shopcart-items>*) is added so that child nodes can be imported from the catalog.

```

public void initCart( string dataSource )
{
    myCart = new XmlDocument();
    if( dataSource != null )
    {
        myCart.LoadXml(dataSource);
    }
    else
    {
        myCart.LoadXml( "<shopcart-items></shopcart-items>" );
    }
}

```

When a user chooses to add an item to the shopping cart, the *onclick* event will call *bookCatalog.catalogItemDetails* and supply an ISBN. The resulting data will be an XML node for that item. The node will then be imported to the XMLCart document via the method *addItem2Cart*. The string representation will then be stored in *Session["myShoppingCart"]*.

```

public string addItem2Cart( XmlDocument book )
{
    //Import the last book node from doc2 into the
    //original document.
    XmlNode newBook =
    myCart.ImportNode(book.DocumentElement.FirstChild, true);
}

```

```
myCart.DocumentElement.AppendChild(newBook);
return "Success";
}
```

When a user selects to remove an item from the shopping cart, the *onclick* event will remove the node specified by the supplied ISBN via the *removeItemFromCart* method, and update *Session["myShoppingCart"]*.

```
public string removeItemFromCart( string isbn )
{
    XmlNode curnode = myCart.SelectSingleNode(
        "//Books[isbn='" + isbn + "']");
    myCart.DocumentElement.RemoveChild( curnode );
}
```

When a user selects **Checkout** from the shopping cart, the *onclick* event will call the Web Service *orderBooks.OrderItem* to update the orders table in the Database, clear the cart via the *clearCart* method, and display confirmation information to the UI.

```
public void clearCart()
{
    XmlElement root = myCart.DocumentElement;
    root.RemoveAll();
}
```

When the page is reloaded and the UI needs the latest version of cart, the XML representation is passed via the *getCartDescription* method:

```
public string getCartDescriptionString()
{
    return myCart.OuterXml;
}
```



## Creating the User Interface

ADOCatalog and XMLCart alone do not provide that much functionality; the real functionality is handled by the *showCatalog* and the *showCart* page methods. Before we take a closer look at that, let's see how the *start.aspx* page is laid out.

## Creating the start.aspx Page

start.aspx is the Web Form that hosts the controls to generate the UI for our catalog and cart. Here's the HTML:

```
<body onload="initializePagevariables()">
```

The preceding code makes a call to a JavaScript function that initializes the values of our hidden field variables.

This next line adds the HTML necessary to draw the navbar. You can also find this file on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the file header.htm.

```
<!-- #Include file="header.htm" -->
<form id="formstart" method="post" runat="server">
<div style="PADDING-RIGHT: 3px; PADDING-LEFT: 3px; PADDING-BOTTOM: 3px;
WIDTH: 800px; COLOR: white; PADDING-TOP: 3px; BACKGROUND-COLOR: dimgray"
align="left">
View Books by Category
```

The following asp:dropdown control reads the list of categories from the database and generates a drop-down select box:

```
<asp:dropdownlist id="CategoryList" runat="server"
    DataValueField="CAT_ID" DataTextField="CAT_Name"></asp:dropdownlist>
<input type="button" id="btnGo" value="Go !" onclick=
    "formstart.categoryState.value='Go';formstart.submit();">
</div>
<table width="800">
<tr>
<td>
```

The following asp:xml server control transforms the supplied XML data with catalog.xslt (see catalog.xslt on the Solutions Web site for the book):

```
<asp:xml id="catalog" runat="server"></asp:xml>
</td>
<td valign="top" align="middle" bgcolor="cornsilk">
```

The following asp:xml server control transforms the supplied XML data with cart.xslt (see cart.xslt on the Solutions Web site for the book):

```
<asp:xml id="cart" runat="server"></asp:xml>
<br>
```

The following asp:Label server control is used to insert HTML that is dynamically generated when the user clicks checkout:

```
<asp:Label id="lblFeedBack" runat="server"></asp:Label>
</td>
</tr>
</table>
```

The following *div* is used to hide a group of text box server controls—so why use a *div* to hide asp:textbox controls? First, while the asp:textbox control does have a visibility attribute, setting this attribute to hidden prevents the HTML from being written to the client, so when we view page source, the HTML for the text box isn't even there. Second, while using the HTML control `<input type="hidden" runat="server">` is also an option, this control lacks postback ability.

Each time a user clicks **Add, Remove, Checkout, Previous, Next**, or makes a change to the drop-down menu for category, we set these hidden variables accordingly and submit the page. Program control is then passed to our code-behind page “start.aspx.cs” (this file can also be found on the Solutions Web site for the book).

```
<div style="VISIBILITY: hidden">
  <asp:textbox id="addItem" runat="server" AutoPostBack="True" />
  <asp:TextBox id="removeItem" runat="server" AutoPostBack="True" />
  <asp:textbox id="firstRecord" runat="server" AutoPostBack="True" />
  <asp:textbox id="lastRecord" runat="server" AutoPostBack="True" />
  <asp:textbox id="direction" runat="server" AutoPostBack="True" />
  <asp:textbox id="recordCount" runat="server" AutoPostBack="True" />
  <asp:TextBox id="categoryState" runat="server" AutoPostBack="True" />
  <asp:TextBox id="Ready4Checkout" runat="server" AutoPostBack="True" />
</div>
</form>
</body>
```

In the following sections, we will see how the user-generated events are handled in our code-behind page: start.aspx.cs.

## Rendering the Catalog

On *page\_load*, we retrieve `Application["AllBooks"]` and apply an XSL/Transform to render the Catalog in the UI. In order to enable browsing, we store the

*FirstRecord*, *LastRecord*, *recordCount*, and user action (previous | next | by CategoryID) into hidden Text fields on the client, so this data can be read to determine which *bookCatalog* method to call and with which parameters to return the desired subset of *AllBooks*.

## Rendering the Cart

When a user makes a selection from the catalog (“Add item to cart,” Previous, Next, or selects a category) or the cart (Remove item, or Checkout), the user’s action is stored in hidden text boxes that are passed to the code-behind *onsubmit()*. In the *Page\_load* method, we will test for *addItem*, *removeItem*, or *Checkout* and handle each accordingly.

## Creating the Code

The code for the *start.aspx.cs* file can be found on the on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). Here is an overview of the page process flow:

- ***In Page\_Load()***
  1. Get list of categories and bind to asp:dropdownlist control *categories*.
  2. Display the default catalog UI by calling *showCatalog()*.
  3. Display the default cart UI by calling *showCart()*.
  4. Test for Add, Remove, and Checkout. Handle each appropriately.
- ***In showCatalog()***
  1. Create an instance of *ADOCatalog* (a.k.a., *bookCatalog*).
  2. Initialize the instance by loading all book data from *Application[“AllBooks”]*.
  3. Test for data filters.
    - Did user make a change to the category drop-down? Filter *AllBooks* for only the selected category.
    - Did user click Previous or Next? Filter *AllBooks* based on the contents in our hidden textboxes: direction, *recordCount*, *firstRecord*, and *lastRecord*.
    - If no filters, use default.



- Set the Document property of the asp:xml control, *catalog* to the filter results.
4. Load XSLT (see *catalog.xslt* on the Solutions Web site for the book).
  5. Set *Transform* properties of the asp:xml control *cart* to *catalog.xslt*.
- ***In showCart()***
    1. Create an instance of XMLCart (a.k.a., *xmlShoppingCart*).
    2. Initialize the instance by loading any previous cart information from *Session["myShoppingCart"]*.
    3. Load XSLT (see: *cart.xslt* on the Solutions Web site for the book).
    4. Set *Document* and *Transform* properties of the asp:xml control, "cart" to *cart.xslt*.

Note that *cart* and *catalog* will have already been initialized and rendered before the next three cases can occur.

- ***In AddItem***
  1. Retrieve from *AllBooks* the node corresponding to the ISBN value stored in the hidden text box *addItem*.
  2. Add this node to our shopping cart.
  3. Store updated cart information in *Session["myshoppingCart"]*.
  4. Rewrite the cart to update the UI.
- ***In RemoveItem***
  1. Using the ISBN stored in the hidden text box *removeItem*, remove the corresponding XML node from *cart*.
  2. Store updated cart information in *Session["myshoppingCart"]*.
  3. Rewrite the cart to update the UI.
- ***In Checkout***
  1. Login user to simulate order processing.
  2. Loop through the *Nodes* in *cart* and update the orders table, then remove ordered item from *cart*, while generating the HTML necessary to display the items ordered in an HTML table.



3. Store updated cart information in *Session*["*myshoppingCart*"]; the cart is empty at this point.
4. Rewrite the cart to update the UI.

There are many ways to display data held in a *DataSet* in XML, or for that matter in ASP.NET. In fact, there are a multitude of controls, including the popular *DataGrid* control that make this relatively simple. We have opted to use XML and XSLT to show other approaches to the same problem. Also, if your current ASP application uses XML and XSLT, the migration to ASP.NET is fairly easy. In fact, your existing XSLT stylesheets and XML content can still be used. For more information on XSLT, visit [www.w3c.org/TR/xslt](http://www.w3c.org/TR/xslt), [www.w3c.org/Style/XSL/#Learn](http://www.w3c.org/Style/XSL/#Learn), and [www.w3schools.com/XSL](http://www.w3schools.com/XSL).

It is important to note that the *Application* and *Session* objects still have issues with regards to server farms and scalability. We used *Session* in this example for simplicity and to show that it can still be useful. Relatively simple changes can be made to the Start page to convert *Session* variables into hidden fields stored on the page, or state can be stored in a database.

## Summary

We have developed an application that enables customers to browse a catalog of books by category or range, add selections to a virtual shopping cart, remove items from the cart and simulate processing an order by logging in and submitting updates to the order table in the database. We have leveraged the power of XML and its ability to represent data and structure, explored Web Services and their methods, designed databases and stored procedures, developed custom code-behind classes in C# and covered a multitude of uses for ADO.NET.

We also explored database design and implementation, creating two databases for the application, one for Access and one for SQL. We then covered entities and their attributes and how both work with each other to create a normalized database. Lastly, we developed a set of stored procedures that will handle all data interaction with the database, preventing the use of “ad hoc” queries against the database.

To see the ADOCart application on the Web go to the Solutions Web site for this books ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



## Solutions Fast Track

### Setting Up the Database

- ☑ A relationship between the two tables is created by the use of *primary* and *foreign keys*.
- ☑ The different types of relationships between tables are one-to-one, one-to-many, and many-to-many. In a one-to-one relationship, exactly one row corresponds with a matching row of the related table. In a one-to-many relationship, one row corresponds to many rows of the related table. In a many-to-many relationship, many rows correspond to many rows of the related table.
- ☑ Using parameterized queries in MS Access and stored providers in MSSQL results in performance gain. In addition, you no longer have to run ad hoc queries against the database. Pre-compiled queries perform better.

## Creating the Web Services

- ☑ Web Services provide separation of the data tier from the user interface (UI). This also makes it possible to access our data from any platform.
- ☑ Web Services help separate our data tier from our application logic. This creates a more robust and portable application.
- ☑ Web Services leverage the power of XML and its interoperability. All pages can communicate with the common language and exist in the same context.

## Using WSDL Web References

- ☑ DISCO, or vsdisco, written in WSDL, enables access to all Web Services and methods for that site. This provides a one-stop shop, if you will, into the server's cupboards.
- ☑ Proxy classes can easily be generated using WSDL, which enables code to access remote services as if they were local classes.

## Building the Site

- ☑ Create an overview of the site structure: What pieces need to be built and how the pages relate to one another. In our example, we focus on the middle tier data classes and controls that act as a bridge between the backend and the Web UI.

## Site Administration

- ☑ Tie the site administration to the Web Services, enabling the administration functions for the site to be done without accessing the code or database. The `adminPage.aspx` page in our example allows the site administrator to retrieve and display data, and to add, delete, and update product.
- ☑ To retrieve the list of books stored in the database, we need to access the `GetAllBooks` stored procedure (MSSQL) or parameterized query (MS Access) by creating the `allBooks` method of the `getBooks` Web Service. This method will take no parameters, and will return a `DataSet`

containing all Book data as well as the table structure of the Database table that the data originated from.

- ☑ The *DataSet* is an in-memory XML representation of the database, including the Books table.
- ☑ Display the data by binding a *DataGrid* to the *DataSet*. The *DataGrid* is actually bound to the *DataTable* Books which is a table within the *DataSet* returned by *getBooks.AllBooks*. We create a *DataView* of the Books table so we can sort the data. This *DataView* is then bound to the *DataGrid*.
- ☑ Using the *DataGrid* event *changeBooks\_DeleteCommand*, fired when a user clicks the **Delete** button in the *DataGrid* UI, we can select a row in the *DataGrid* to delete by using the *RowFilter* property.
- ☑ Using the *DataGrid* event *changeBooks\_UpdateCommand*, fired when a user clicks the **Update** button in the *DataGrid* UI, we can select the row in the *DataGrid* to update by using the *RowFilter* property.

## Customer Administration

- ☑ The Customer Administration pages tie our customer administration interface to our Web Services, enabling the customer to update their personal information. This is an added benefit to the user of the site.
- ☑ Customer administration will be identical to the example of the site administrator, except we will enable the customer to update only.

## Creating an ADOCatalog

- ☑ Creating an ADOCart application allows us to tie our catalog interface to our Web Services. In our ADOCart application, all database interaction is handled via Web Services.
- ☑ Create a new class to explore ADO.NET *DataSet* operations in order to: copy, clone, import, create, and filter.
- ☑ Since our *Books* data is fairly static, we can retrieve the data in a *DataSet* once and store that *DataSet* in an application-level variable. This reduces the database traffic, while still providing quick access to the data.

- ☑ Use XML and XSLT to render data stored in a *DataSet* as HTML via the `asp:Xml` server control.

## Building an XMLCart

- ☑ Building an XMLCart allows us to tie our catalog to the shopping cart.
- ☑ We will use XML node operations to update our cart's contents, XSLT/XPath operations to calculate cart totals and taxes, XML and XSLT to render cart data as HTML, and the `asp:XML` server control to process transforms.
- ☑ An *XmlDocument* wrapper class provides add, remove, clear, and checkout operations.

## Creating the User Interface

- ☑ *ADOCatalog* and XMLCart alone do not provide that much functionality; the real functionality is handled by the *showCatalog* and the *showCart* page methods.
- ☑ `start.aspx` is the Web Form that hosts the controls to generate the UI for our catalog and cart.
- ☑ Use of XML and XSLT generates portions of the UI via the `asp:xml` server controls.

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** My project has a few different pages in it. Unfortunately, the last page I created is the one that is loaded when I run the project. How do I set the first page to open when I run the project?

**A:** In your **Project Explorer**, right-click the file you want and set it to **Start Page**.

**Q:** I am working with the *XmlDocument* object in my code-behind page, and I am not getting any IntelliSense. What am I doing wrong?

**A:** Make sure you have included “Using System.Xml” in the top section of the page.

**Q:** I just started using VS.NET Beta 2 and I am trying to create a WSDL proxy to my Web Service. Is there an easy way to do this in VS.NET?

**A:** Right-click your Project Explorer and select **Add Web reference**.

**Q:** I renamed a file in my Solutions Explorer, but the corresponding “.aspx.cs” and “.aspx.resx” names did not change. Because of this, the project will not compile correctly. How can I fix this?

**A:** In your Solutions Explorer, make sure all child files are collapsed in the parent when renaming and this will change all the associated files. If you have already changed one file, change it back to the name prefix of the other files, then collapse the children and rename it to the new name. Also, check the first line in the “.aspx” page and ensure that the Inherits attribute lists the correct filename.

**Q:** How can I get the specific event functions written to the code-behind page?

**A:** In the **Properties** window, select the icon **Events**. From there, you can select whatever event can be fired from a given object.



## Building a SQLXML Web Service Application

### Solutions in this chapter:

- SQLXML Web Services
- Developing the TimeTrack Application
- Creating a SQL Server Virtual Directory
- Creating a Client Application in ASP.NET
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions



# Introduction

The main benefit of using SQLXML Web Services is that these services simplify the interaction between SQL, multiple platforms, and various languages including versions of Oledb (ADO), ODBC, JDBC, and so on. SQLXML Web Services help to better connect databases over an extranet, an intranet, or the Internet through the use of Active Server technologies. This enables multiple platforms to access the same services and attain the same results. In this chapter we will look at all the functionality that SQL Web Services can bring to our applications. One of the benefits of SQLXML Web Services is that they essentially cut out all use of Active Data Objects (ADO). By using a Web Service to complete our data interaction we can remove a level of complexity and thus remove a layer of potential errors, for instance, ADO recordset object errors. Like all database driven applications, you must always provide the back-end before designing and building the front-end application. By using SQLXML Web Services, deployment time is reduced considerably and resources are increased by virtually “killing two birds with one stone.” SQLXML Web Services also opens the back-end to multiple front-ends that may be built by third parties, with no additional development work.

There are some added features that need to be installed before we can get started. The main thing is to have all the current service packs for SQL 2000 and Windows 2000 Server installed before going forward. In the overview we will cover how to install and configure the SQLXML Web Services Toolkit. To run the application we are going to build you must have Windows 2000 Server and SQL Server 2000 with the SQL Web Services Toolkit installed.

Specifically, system requirements are:

- Windows 2000 Professional
- Windows 2000 Professional SP2
- SQL Server 2000 Developer’s Edition
- SQLXML 3.0
- SQLXML Web Services Toolkit

Using SQLXML Web Services is also a low-cost way to upgrade your current application (if it is not .NET) to a .NET platform. Instead of completely redoing the entire code base to upgrade the software to .NET there has been some investigation into using SQL Web Services as a bridge to achieving .NET functionality without dropping the whole project into the .NET sphere. This incremental migration strategy provides a low-cost and effective way to get an organization

into .NET. The hope is that this type of migration will be easier for clients to adopt a .NET philosophy more easily than having organizations completely relearn how to work a system.



In this chapter we will be building an application that will track projects for an organization—the project is aptly named TimeTrack. All of the files necessary for creating this application are available on the Solutions Web site for this book, at [www.syngress.com/solutions](http://www.syngress.com/solutions). First, we will be building all the database elements and stored procedures, which will then be turned into Web Services. These Web Services will then be accessed by our ASP.NET application.

## SQLXML Web Services

Once you install the SQL 2000 Web Services Toolkit (most importantly for this discussion, SQLXML 3.0) you will be able to communicate to SQL Server via HTTP by creating a SQL Virtual Root Directory from one of your selected databases. SQLXML 3.0 is configured to work with .NET and will do the work of generating your Web Service Description Language (WSDL) file in order to process the stored procedures or user defined functions as Web Services. A host of managed classes are available via .NET, but because they are beyond the scope of this topic, we will not discuss them here. However, the SQLXML Web Services Toolkit includes a white paper.

One of the key benefits of using SQL Web Services is the cost savings on time for development. If you have developed any type of case management software you know that the work you do in the back-end pays huge dividends on the front-end application. Every database requires some set of stored procedures to interact with it, for example adding new records to an employee's table or editing those same records. After creating your set of stored procedures you then have to access them via ADO.NET in the application logic in order to pass the data. In this sample application we will get around this last bit of coding by enabling our set of stored procedures to be Web Services and to handle all of our database interaction.

## Developing the TimeTrack Application

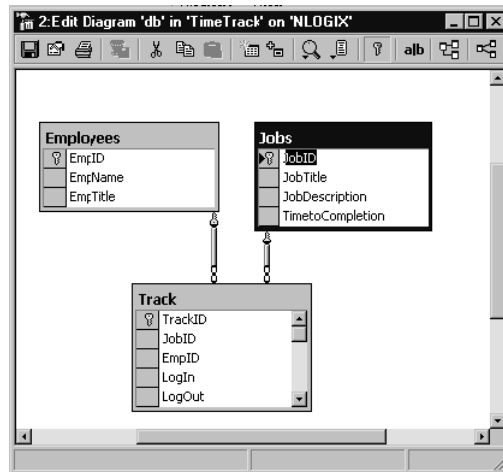
We will now develop a time tracking application that will use an employee sign in to track time spent on a given project or task. This is always a useful tool to have developed for your organization for reporting and using the data for cost analysis. Everyone in IT has had the problem of over- or under-projected timelines for a given job or task. By tracking current project times an organization can better formulate future job timeline projections.

This is a simple example of how to cut out ADO and enable stored procedures to do the work instead. Our application has a simple database with a set of stored procedures to handle the data interactions. This application will track employees and the projects they are working on. We have laid out the shell of the application and will let you fill in or expand upon the application framework to suit your own needs.

## Creating the Database

The first thing we need to design is the database. Let's take a look at the schema layout, which can be seen in Figure 7.1.

**Figure 7.1** Database Schema Layout



As you can see by the schema, we will have a simple three-table database. To reiterate, this is just a shell—if you want to add tables to this sample feel free. This will track time spent on a project and by what employee. First we need to create the database in SQL Server. Right-click on **Databases** in EM and select **New Database** (see Figure 7.2). You can call it what you want; we used the name TimeTracker. Leave all of the create database defaults set and select **OK**. You may attach our database from the code section on the Solutions Web site ([www.syngress.com/solutions](http://www.syngress.com/solutions)), filename TimeTrack\_data.mdf. To attach a database to your sever, right-click on **Databases** in EM and select **Attach Database** as seen in Figure 7.3. To create the tables simply navigate in Enterprise Manager (EM) and right-click on **Tables** and select **New**.

If you are using the database that has been provided for you on the Solutions Web site for this book ([www.syngress.com/solutions](http://www.syngress.com/solutions)), please be sure that you have

selected the TimeTrack\_data.mdf file. Make sure that you set the correct Database Owner (DBO) for the database; most likely it will be SA. Now that you have the database created you can add the tables. If you have created a new database instead of attaching the one provided on the Solutions Web site for this book, you must also add each of the tables. Right-click on the **Table menu** under database and select **New Table**. Add additional tables to the schema if needed.

Figure 7.2 New Database Menu

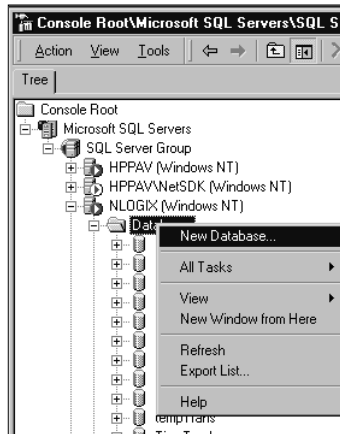
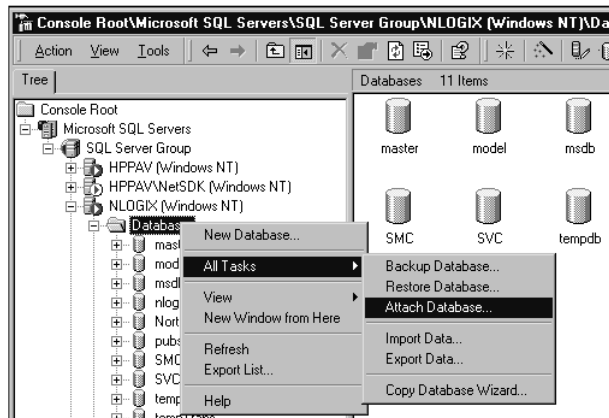


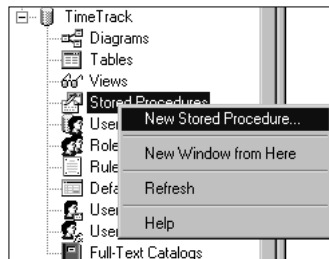
Figure 7.3 Attach Database Menu



## Creating the Stored Procedures

Now that we have the framework set up, let's add the stored procedures to this database. To create a new stored procedure right-click on **Stored Procedures** and select **New Stored Procedure** (see Figure 7.4).

Figure 7.4 Stored Procedure Menu



Looking at the code for one stored procedure, you can add as many as you like based on what functionality you want to have. Here you can look at the code for the *addEmp* stored procedure, which is used for adding employees.

```
create proc addEmp
    @Name nvarchar(50),
    @Title nvarchar(50)
as
insert into Employees (EmpName,EmpTitle) Values (@Name,@Title)
GO
```

The syntax is fairly simple—the main thing to remember is that if you want to pass parameters you must declare them before the keyword *AS*. Here is a stored procedure that we have named *showPastDueJobs* that will be used to generate our report on overdue projects:

```
create proc showPastDueJobs
as
select j.JobTitle, SUM(t.total) as TOTAL
from Track t
join Jobs j
    on t.JobID = j.JobID
where TOTAL >= TimetoCompletion
group by JobTitle
GO
```

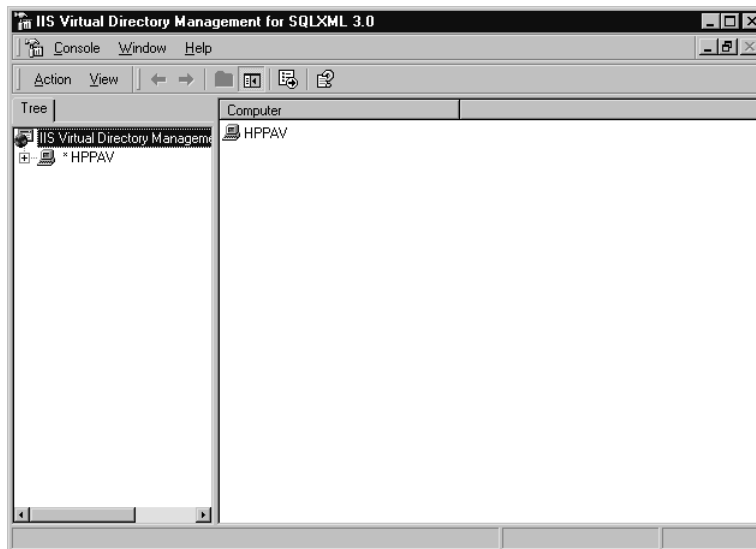
As you can see there are no parameters declared, so you can write a *select* statement to return your results. You want to use stored procedures because they offer a huge gain in performance over using ad hoc queries against the database. Using stored procedures will give you precompiled and optimized results, which will be waiting for delivery, as opposed to having a SQL Server parse the query,

optimize it, and finally return the result set. Ad hoc queries are statements sent to SQL Server that must be parsed and optimized before being delivered. By using stored procedures, you are subverting those steps, thus saving time and increasing performance and scalability.

## Creating a SQL Server Virtual Directory

Somewhere you need to create the root folder for the virtual directory and the set of subfolders we will be using. You will need a folder named `Track`; inside of `Track` put the subfolder named `SOAP`. You can either establish the directory from `C:\`, or put the folder within the `C:\inetpub\wwwroot` folder next to your client application. Now that you have the database complete, you can create the vital piece to making this whole process work, the SQL Server Virtual root. Without this you will be unable to communicate to your SQL Server through HTTP. However, it is a simple process. First, either install or verify that the full install of SQLXML 3.0 from the Web Services Toolkit is ready to go. The Web Services Toolkit comes with the SQL 2000 Server. Once you have the Toolkit installed select the **Configure IIS Support** option from the submenu. You should be looking at the window shown in Figure 7.5.

**Figure 7.5** IIS Virtual Directory Management for SQLXML 3.0 Window



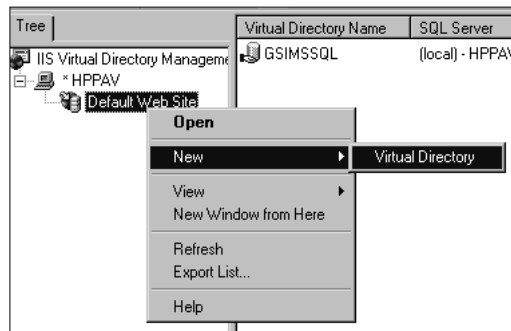
**NOTE**

Another way to create a SQLXML virtual directory is to use the IIS Virtual Directory Management for SQL Server object model. You can create a virtual directory in VBScript by using this object model. An advantage of using this method is that you can then create an installation script to set up the environment. For more details on how to use this method and create an installation script we urge you to refer to the SQLXML documentation, *Creating the nwind Virtual Directory by Using the Object Model*, located in the SQLXML help file that comes with the SQLXML Toolkit.

Your server name will be at the highest level of the tree.

After you have done this, go back to the SQL IIS configuration window and expand your **Server name**. Right-click on **Default Web Site**, and then select the option **New Virtual Directory**. This selection path is illustrated in Figure 7.6.

**Figure 7.6** Virtual Directory Menu



We will now go through a series of options and set the values for each. The first step is to give your virtual root a name and location. The window for selecting the name and location of the Virtual Root Directory is shown in Figure 7.7.

Select the **Security** tab. Here you will set the type of security you will use to access SQL Server; for the purpose of this example, use the SQL Authentication security. To do this, make sure that the **Always log on as** option is checked, and select the **SQL Server** option as your account type. Provide a username and password (see Figure 7.8). You could use Windows Integrated Authentication, which means that you would need all associated rights within the domain for both the IIS server and SQL Server; however, for development purposes it is less convoluted to use SQL Authentication and separate the security element from the project.

Figure 7.7 Virtual Directory General Tab

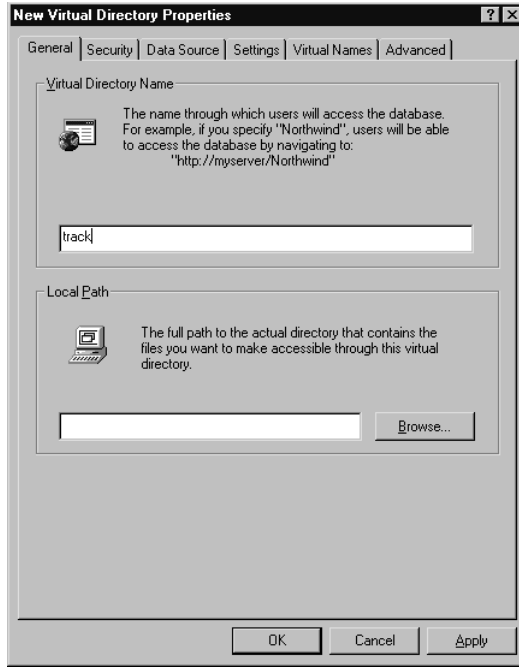
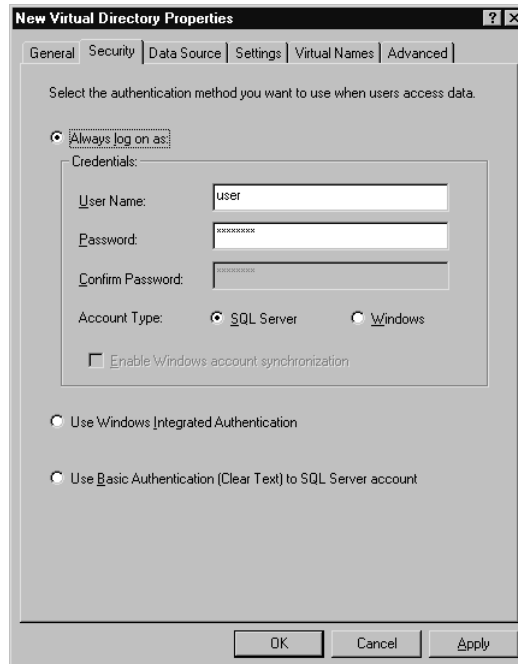


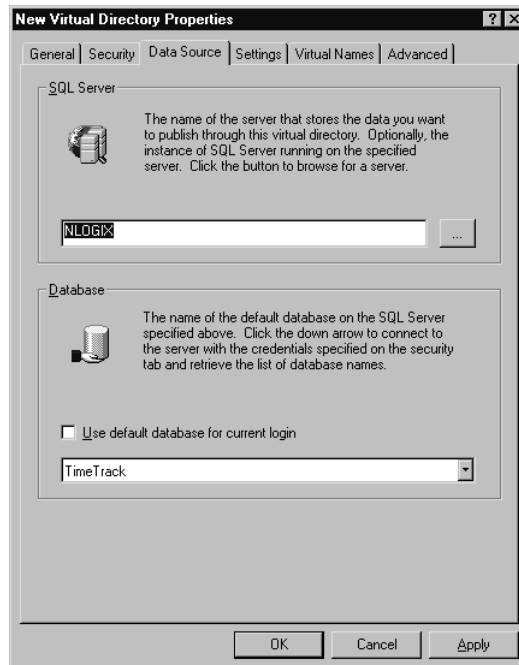
Figure 7.8 Virtual Directory Security Tab





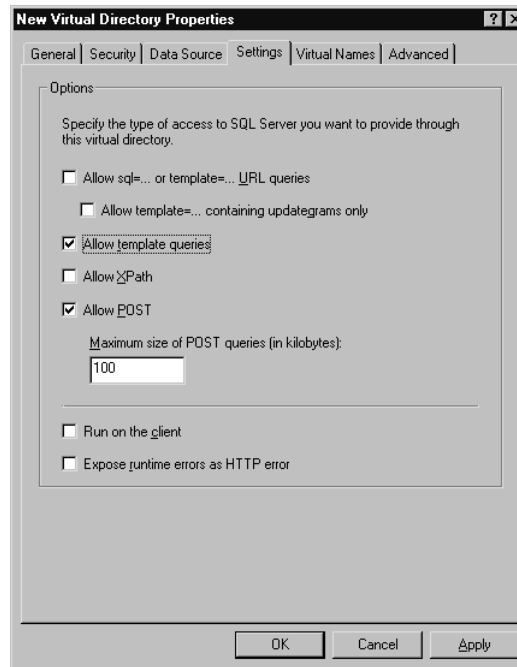
Moving on to the Data Source Tab, here it is possible to search for all of your available servers and select the one you want. Note, you should uncheck the **Use default database for current login** option and then browse to either the TimeTrack database that has been downloaded from the Solutions Web site for the book, or the database that you created for this project. Click on the database that you will be using, and be sure that it appears in the dialog box. You should see an image similar to Figure 7.9.

**Figure 7.9** Virtual Directory Data Source Tab



Now we can set the settings for this Virtual Root. For our sample application we will just be using the Allow POST option. This will allow us to post a query to the database through HTTP. You can also set up XML updateagrams and use that option or enable sql= queries and return result sets via the URL. Depending upon the security needed for your application you may or may not want to do this, because you will be exposing your database schema and objects in the URL. For now select **Allow POST** (see Figure 7.10) and leave the default size of **100 KB**; we will not be sending anything larger than that.

Figure 7.10 Virtual Directory Settings Tab



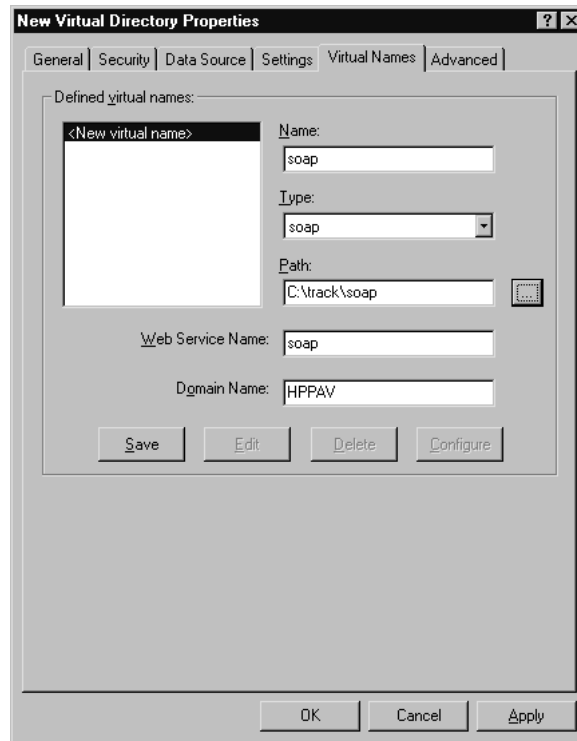
## Configuring & Implementing...

### SQL Template Queries

Previously accessing SQL Templates server-side from within an ASP.NET application would fail to load the XML because the security context of the user would be lost when hopping from IIS to SQL. SQLXML 3.0 solves this problem by allowing server-side access to Template queries by setting `SqlXmlCommand.CommandType = SqlXmlCommandType.TemplateFile`.

Now you can map your subfolder soap to our virtual name, called “soap.” Select the **Virtual Names** tab. First, select **New Virtual Name** and give it the name **soap**; its type will be soap. These steps should resemble the image in Figure 7.11. This will create the WSDL file for your Web Service enabled stored procedures. You will have to navigate to the path of the subfolder named soap within the root folder track that you created earlier in the chapter.

Figure 7.11 Virtual Directory Virtual Names Tab



The Domain name will be your server name and you can leave the Web Services name as soap. Now select **Save**. Select **Configure**; this is where we will load all of our SQL Objects that we want to become Web Services.

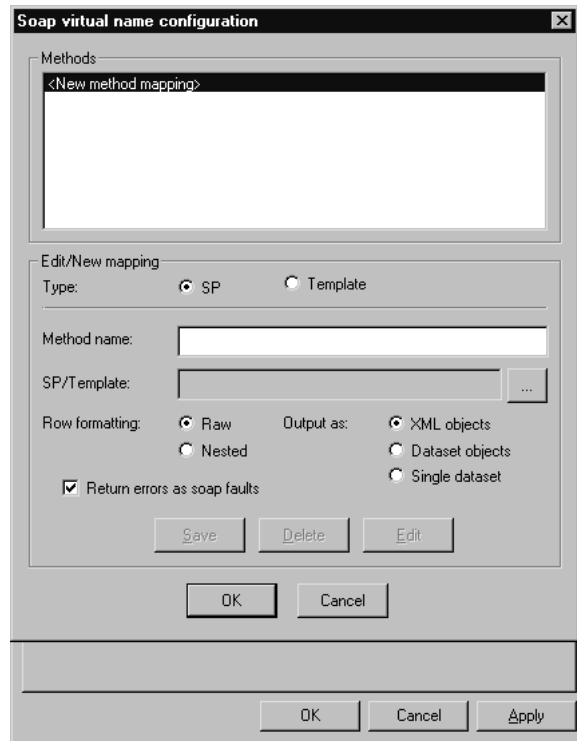
## Enabling Stored Procedures for Soap

In this section we will get down to the heart of the application, which is the enabling of the stored procedures to be Web Services. This is the whole purpose of doing this exercise and is a vital step not to be missed.

Now we can select the specific procs that we wish to use in our client application as Web Services.

1. Select **New Method Mapping** (see Figure 7.12).
2. Check to see if the Edit/New Mapping type option SP is selected. Then select the **push button browse** option and view all of the stored procedures available. Your list should resemble the list shown in Figure 7.13.

Figure 7.12 Method Mapping



3. Select **showEmployees**. The method name will remain the same as the stored procedure. We suggest that you leave this alone; it will be less confusing when you access this method in the client application. You will also want to be sure that your Output option has the Single Dataset radio button selected and ensure that what is returned is a single Dataset, as shown in Figure 7.14. If you need an array of Datasets or XML objects you could choose the other options (XML Objects or Dataset Objects), but for this sample application we will use the Single Datasets Output exclusively.
4. Continue to do this for the rest of the stored procedures in the database.

Figure 7.13 Stored Procedures List

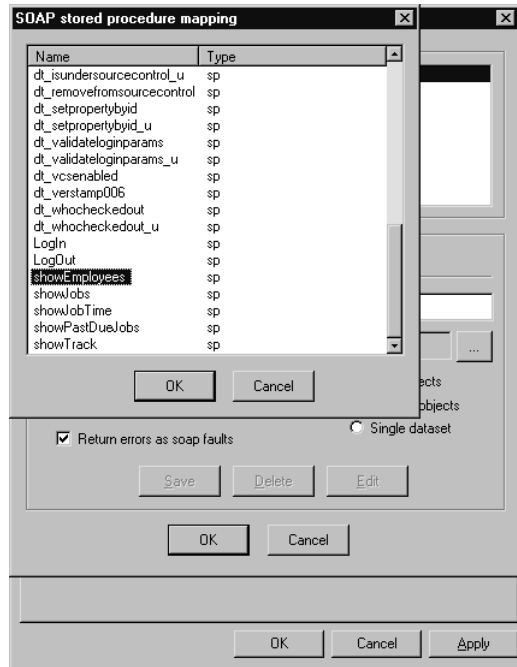
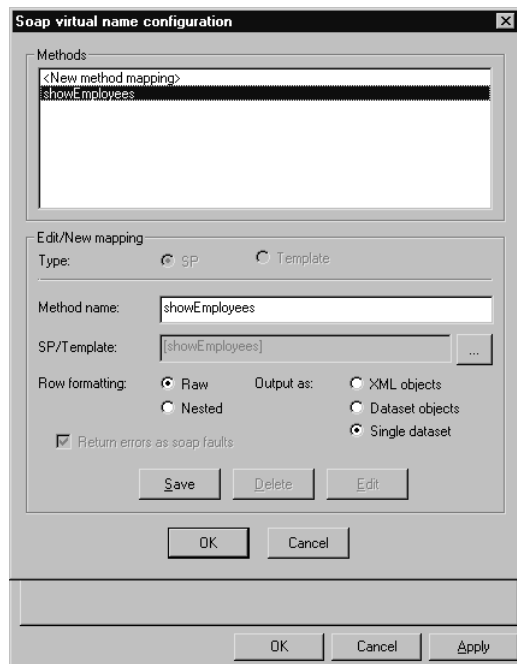


Figure 7.14 Method showEmployees



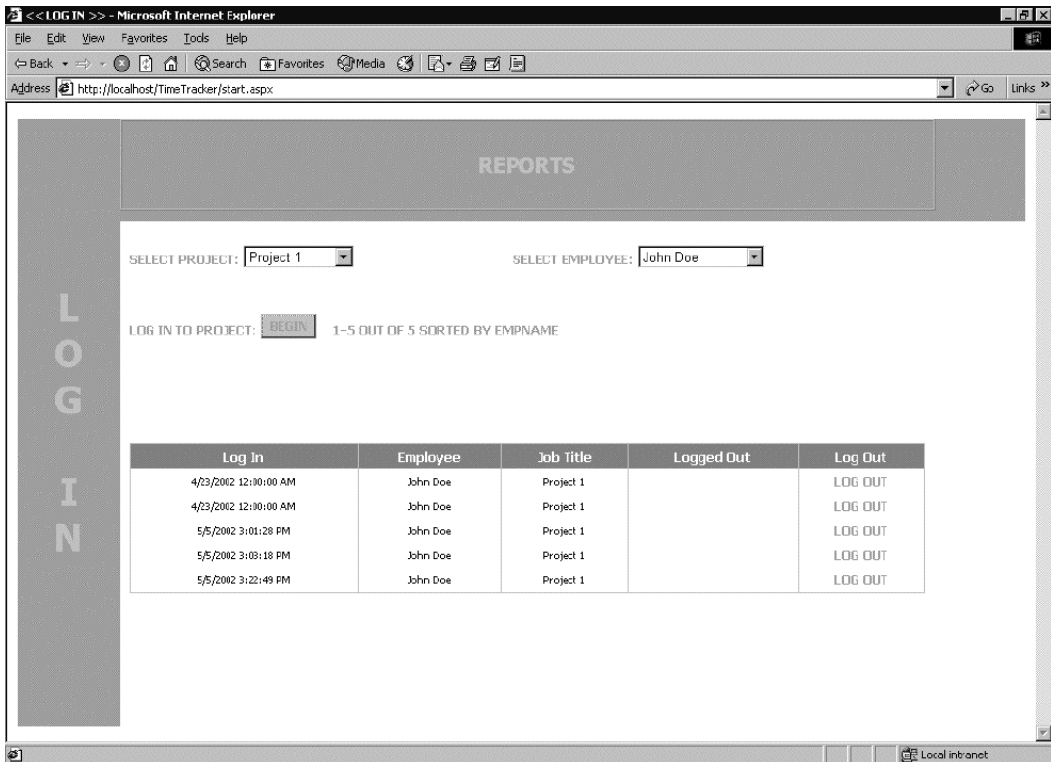
# Creating a Client Application in ASP.NET

Open VS.NET and select a new C# Web application. Name it TimeTracker and select **Create**. Our sample application we be very simple, just three pages:

- A Log In page
- A Log Out page
- A Reports page

We will be covering the Log In page, which shows how easy it is to use the SQL Web Services we created earlier in our sample application. The layout or front-end design of our ASP.NET application can be seen in Figure 7.15.

**Figure 7.15** ASP.NET Front-end Design



Let's see the code for this page. First we will go over the HTML portion of the Web form. The code for the start.aspx page is listed in Figure 7.16. The full source code for this application is also available on the Solutions Web site for this book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 7.16** start.aspx

```

<%@ Page language="c#" Codebehind="default.aspx.cs" AutoEventWireup=
    "false" Inherits="TimeTracker.WebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN" >
<HTML>
    <HEAD>
        <title><< LOG IN >></title>
        <meta content="Microsoft Visual Studio 7.0" name="GENERATOR">
        <meta content="C#" name="CODE_LANGUAGE">
        <meta content="JavaScript" name="vs_defaultClientScript">
        <meta content="http://schemas.microsoft.com/intellisense/ie5"
            name="vs_targetSchema">
        <style>A.Up { FONT-WEIGHT: bold; FONT-SIZE: 15pt; TEXT-TRANSFORM:
            uppercase; COLOR: wheat; FONT-FAMILY: Tahoma; TEXT-DECORATION: none }
        A.Up:hover { FONT-WEIGHT: bold; FONT-SIZE: 15pt; TEXT-TRANSFORM:
            uppercase; COLOR: white; FONT-FAMILY: Tahoma; TEXT-DECORATION: none }
        A { FONT-WEIGHT: bold; FONT-SIZE: 10pt; TEXT-TRANSFORM: uppercase; COLOR:
            tomato; FONT-FAMILY: Tahoma; TEXT-DECORATION: none }
        A:hover { FONT-WEIGHT: bold; FONT-SIZE: 10pt; TEXT-TRANSFORM:
            uppercase; COLOR: wheat; FONT-FAMILY: Tahoma; TEXT-DECORATION: none }
        .text { FONT-WEIGHT: bold; FONT-SIZE: 15pt; TEXT-TRANSFORM: uppercase;
            COLOR: wheat; FONT-FAMILY: Tahoma; TEXT-DECORATION: none }
        .title { FONT-WEIGHT: bold; FONT-SIZE: 28pt; TEXT-TRANSFORM:
            uppercase; COLOR: wheat; FONT-FAMILY: Tahoma }
        .stext { FONT-WEIGHT: bold; FONT-SIZE: 10pt; TEXT-TRANSFORM:
            uppercase; COLOR: tomato; FONT-FAMILY: Tahoma; TEXT-DECORATION: none }
        </style>
    </HEAD>
    <body MS_POSITIONING="Flowlayout">
        <form id="Form1" runat="server" method="post" action="default.aspx">
<table height="100%" cellSpacing="0" cellPadding="0" width="100%" border="0">
<tr>
    <td class="title" align="middle" width="100" bgColor=
        "tomato" rowSpan="2">L<br>
    O<br>
    G<br>
    <br>

```

Continued





**Figure 7.16 Continued**


---

```

"#DEDFDE" AutoGenerateColumns="False" BorderStyle="None" BorderWidth=
"1px" BackColor="White" CellPadding="4" GridLines=
"Vertical" ForeColor="Black">
<SelectedItemStyle Font-Size="XX-Small" Font-Names="Tahoma" Font-Bold=
"True" HorizontalAlign="Center" ForeColor=
"White" BackColor="#CE5D5A"></SelectedItemStyle>
<EditItemStyle Font-Size="XX-Small" Font-Names=
"Tahoma" HorizontalAlign="Center"></EditItemStyle>
<AlternatingItemStyle Font-Size="XX-Small" Font-Names=
"Tahoma" HorizontalAlign="Center"
BackColor="White"></AlternatingItemStyle>
<ItemStyle Font-Size="XX-Small" Font-Names=
"Tahoma" HorizontalAlign="Center" BackColor="#F7F7DE"></ItemStyle>
<HeaderStyle Font-Size="Smaller" Font-Names=
"Tahoma" Font-Bold="True" HorizontalAlign="Center" ForeColor=
"White" VerticalAlign="Middle" BackColor="#6B696B"></HeaderStyle>
<FooterStyle BackColor="#CCCC99"></FooterStyle>
<Columns>
<asp:BoundColumn DataField="LogIn" HeaderText="Log In"></asp:BoundColumn>
<asp:BoundColumn DataField=
"EmpName" HeaderText="Employee"></asp:BoundColumn>
<asp:BoundColumn DataField="JobTitle" HeaderText=
"Job Title"></asp:BoundColumn>
<asp:BoundColumn DataField="LogOut" HeaderText=
"Logged Out"></asp:BoundColumn>
<asp:HyperLinkColumn Text="Log Out" Target="_top" HeaderText=
"Log Out" NavigateUrl="logout.aspx"></asp:HyperLinkColumn>
</Columns>
</asp:datagrid></td>
</tr>
</table>
</td>
</tr>
</table>
</form>

```

---

Continued

## Figure 7.16 Continued

```
</body>  
</HTML>
```

This code contains the three basic pieces you will need:

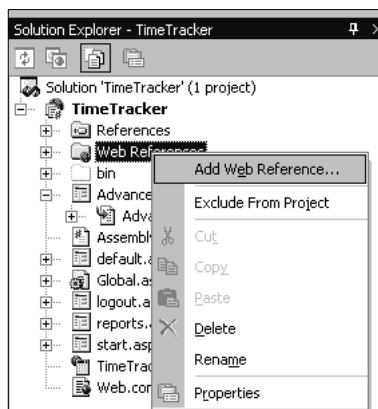
1. A *DataGrid* added to the page to show the results of when a user logged in and what project they are logged into.
2. Two drop-downs that will be populated dynamically from the database to provide the projects and employees available to log in with.
3. Everything else on this page is subject to change if you want; it is all styles and layout properties that can be changed with a few clicks.

Previous chapters extensively covered the topic of *DataGrids*, therefore we will not repeat ourselves here. Your best option is to use the property builder and add the bound columns needed from the table *track* and add one more for a hyperlink column to our log out page.

## Consuming the Web Services

Now that you have built the front-end of the Web form we can go to the code behind the cs page and add our program logic. This is where we will use our SQL Web Services to bind our controls with data. We first need to add a Web reference to our SQL Web Service. Right-click on **Web References** and select **Add Reference**. This path is shown in Figure 7.17.

### Figure 7.17 Add Web Reference



This will bring up the UDDI discovery tool built into VS.NET. Type your server name (most likely it will be (localhost)); specify the virtual root folder name, track, followed by the subfolder virtual name, soap; and finally add ?wsdl. Your URL should look like this: `http://localhost/track/soap?wsdl`.

Your WSDL should populate in the left-hand pane; if so, select **Add Reference**. Let's look at the resulting WSDL output. The output of `soap.wsdl` is listed in Figure 7.18; this output is also available on the Solutions Web site for this book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) for additional review.

**Figure 7.18** `soap.wsdl`

```
<?xml version="1.0" ?>
- <wsdl:definitions name="soapSQL"
targetNamespace="http://NLOGIX/track/soapSQL"
xmlns:tns="http://NLOGIX/track/soapSQL"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:sql="http://schemas.microsoft.com/SQLServer/2001/12/SOAP"
xmlns:sqltypes="http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types"
xmlns:sqlmessage="http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types/
  SqlMessage"
xmlns:sqlresultstream="http://schemas.microsoft.com/SQLServer/2001/12/SOAP/
  types/SqlResultStream">
- <wsdl:types>
- <xsd:schema targetNamespace=
  "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types"
  elementFormDefault="qualified" attributeFormDefault="qualified">
  <xsd:import namespace="http://www.w3.org/2001/XMLSchema" />
- <xsd:simpleType name="nonNegativeInteger">
- <xsd:restriction base="xsd:int">
  <xsd:minInclusive value="0" />
  </xsd:restriction>
  </xsd:simpleType>
  <xsd:attribute name="IsNested" type="xsd:boolean" />
- <xsd:complexType name="SqlRowSet">
- <xsd:sequence>
  <xsd:element ref="xsd:schema" />
```

Continued

**Figure 7.18 Continued**


---

```

<xsd:any />
</xsd:sequence>
<xsd:attribute ref="sqltypes:IsNested" />
</xsd:complexType>
- <xsd:complexType name="SqlXml" mixed="true">
- <xsd:sequence>
  <xsd:any />
</xsd:sequence>
</xsd:complexType>
- <xsd:simpleType name="SqlResultCode">
- <xsd:restriction base="xsd:int">
  <xsd:minInclusive value="0" />
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>
- <xsd:schema targetNamespace=
  "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types/SqlMessage"
  elementFormDefault="qualified" attributeFormDefault="qualified">
  <xsd:import namespace="http://www.w3.org/2001/XMLSchema" />
  <xsd:import namespace=
    "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types" />
- <xsd:complexType name="SqlMessage">
- <xsd:sequence minOccurs="1" maxOccurs="1">
  <xsd:element name="Class" type="sqltypes:nonNegativeInteger" />
  <xsd:element name="LineNumber" type="sqltypes:nonNegativeInteger" />
  <xsd:element name="Message" type="xsd:string" />
  <xsd:element name="Number" type="sqltypes:nonNegativeInteger" />
  <xsd:element name="Procedure" type="xsd:string" />
  <xsd:element name="Server" type="xsd:string" />
  <xsd:element name="Source" type="xsd:string" />
  <xsd:element name="State" type="sqltypes:nonNegativeInteger" />
</xsd:sequence>
<xsd:attribute ref="sqltypes:IsNested" />
</xsd:complexType>
</xsd:schema>

```

---

Continued

## Figure 7.18 Continued

---

```

- <xsd:schema targetNamespace=

"http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types/SqlResultStream"
  elementFormDefault="qualified" attributeFormDefault="qualified">
  <xsd:import namespace="http://www.w3.org/2001/XMLSchema" />
  <xsd:import namespace=
    "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types" />
  <xsd:import namespace=
    "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types/SqlMessage" />
- <xsd:complexType name="SqlResultStream">
- <xsd:choice minOccurs="1" maxOccurs="unbounded">
  <xsd:element name="SqlRowSet" type="sqltypes:SqlRowSet" />
  <xsd:element name="SqlXml" type="sqltypes:SqlXml" />
  <xsd:element name="SqlMessage" type="sqlmessage:SqlMessage" />
  <xsd:element name="SqlResultCode" type="sqltypes:SqlResultCode" />
</xsd:choice>
</xsd:complexType>
</xsd:schema>
- <xsd:schema targetNamespace="http://NLOGIX/track/soapSQL"
  elementFormDefault="qualified" attributeFormDefault="qualified">
  <xsd:import namespace="http://www.w3.org/2001/XMLSchema" />
  <xsd:import namespace=
    "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types" />
  <xsd:import namespace=
    "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types/SqlMessage" />
  <xsd:import namespace=
    "http://schemas.microsoft.com/SQLServer/2001/12/SOAP/types/
    SqlResultStream" />
- <xsd:element name="LogIn">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="0" maxOccurs="1" name="EmpID" type=
    "xsd:int" nillable="true" />
  <xsd:element minOccurs="0" maxOccurs="1" name="JobID" type=

```

---

Continued

**Figure 7.18 Continued**

---

```
    "xsd:int" nillable="true" />
  <xsd:element minOccurs="0" maxOccurs="1" name="LogIn" type=
    "xsd:dateTime" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="LogInResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "LogInResult" type="sqltypes:SqlRowSet" />
  <xsd:element name="returnValue" type="xsd:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="LogOut">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="0" maxOccurs="1" name="TrackID" type=
    "xsd:int" nillable="true" />
  <xsd:element minOccurs="0" maxOccurs="1" name="LogOut" type=
    "xsd:dateTime" nillable="true" />
  <xsd:element minOccurs="0" maxOccurs="1" name="Notes" type=
    "xsd:string" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="LogOutResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "LogOutResult" type="sqltypes:SqlRowSet" />
  <xsd:element name="returnValue" type="xsd:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
```

---

Continued

**Figure 7.18 Continued**


---

```

    </xsd:element>
- <xsd:element name="showEmployees">
- <xsd:complexType>
  <xsd:sequence />
</xsd:complexType>
</xsd:element>
- <xsd:element name="showEmployeesResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "showEmployeesResult" type="sqltypes:SqlRowSet" />
  <xsd:element name="returnValue" type="xsd:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="showJobs">
- <xsd:complexType>
  <xsd:sequence />
</xsd:complexType>
</xsd:element>
- <xsd:element name="showJobsResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "showJobsResult" type="sqltypes:SqlRowSet" />
  <xsd:element name="returnValue" type="xsd:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="showJobTime">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="0" maxOccurs="1" name="JobID" type=
    "xsd:int" nillable="true" />
</xsd:sequence>

```

---

Continued

**Figure 7.18 Continued**

---

```
</xsd:complexType>
</xsd:element>
- <xsd:element name="showJobTimeResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "showJobTimeResult" type="sqltypes:SqlRowSet" />
  <xsd:element name="returnValue" type="xsd:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="showPastDueJobs">
- <xsd:complexType>
  <xsd:sequence />
</xsd:complexType>
</xsd:element>
- <xsd:element name="showPastDueJobsResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "showPastDueJobsResult" type="sqltypes:SqlRowSet" />
  <xsd:element name="returnValue" type="xsd:int" nillable="true" />
</xsd:sequence>
</xsd:complexType>
</xsd:element>
- <xsd:element name="showTrack">
- <xsd:complexType>
  <xsd:sequence />
</xsd:complexType>
</xsd:element>
- <xsd:element name="showTrackResponse">
- <xsd:complexType>
- <xsd:sequence>
  <xsd:element minOccurs="1" maxOccurs="1" name=
    "showTrackResult" type="sqltypes:SqlRowSet" />
```

---

Continued



**Figure 7.18 Continued**


---

```

    <xsd:element name="returnValue" type="xsd:int" nillable="true" />
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>
</wsdl:types>
- <wsdl:message name="LogInIn">
  <wsdl:part name="parameters" element="tns:LogIn" />
</wsdl:message>
- <wsdl:message name="LogInOut">
  <wsdl:part name="parameters" element="tns:LogInResponse" />
</wsdl:message>
- <wsdl:message name="LogOutIn">
  <wsdl:part name="parameters" element="tns:LogOut" />
</wsdl:message>
- <wsdl:message name="LogOutOut">
  <wsdl:part name="parameters" element="tns:LogOutResponse" />
</wsdl:message>
- <wsdl:message name="showEmployeesIn">
  <wsdl:part name="parameters" element="tns:showEmployees" />
</wsdl:message>
- <wsdl:message name="showEmployeesOut">
  <wsdl:part name="parameters" element="tns:showEmployeesResponse" />
</wsdl:message>
- <wsdl:message name="showJobsIn">
  <wsdl:part name="parameters" element="tns:showJobs" />
</wsdl:message>
- <wsdl:message name="showJobsOut">
  <wsdl:part name="parameters" element="tns:showJobsResponse" />
</wsdl:message>
- <wsdl:message name="showJobTimeIn">
  <wsdl:part name="parameters" element="tns:showJobTime" />
</wsdl:message>
- <wsdl:message name="showJobTimeOut">
  <wsdl:part name="parameters" element="tns:showJobTimeResponse" />

```

---

**Continued**

## Figure 7.18 Continued

---

```
</wsdl:message>
- <wsdl:message name="showPastDueJobsIn">
  <wsdl:part name="parameters" element="tns:showPastDueJobs" />
</wsdl:message>
- <wsdl:message name="showPastDueJobsOut">
  <wsdl:part name="parameters" element="tns:showPastDueJobsResponse" />
</wsdl:message>
- <wsdl:message name="showTrackIn">
  <wsdl:part name="parameters" element="tns:showTrack" />
</wsdl:message>
- <wsdl:message name="showTrackOut">
  <wsdl:part name="parameters" element="tns:showTrackResponse" />
</wsdl:message>
- <wsdl:portType name="SXSPort">
- <wsdl:operation name="LogIn">
  <wsdl:input message="tns:LogInIn" />
  <wsdl:output message="tns:LogInOut" />
</wsdl:operation>
- <wsdl:operation name="LogOut">
  <wsdl:input message="tns:LogOutIn" />
  <wsdl:output message="tns:LogOutOut" />
</wsdl:operation>
- <wsdl:operation name="showEmployees">
  <wsdl:input message="tns:showEmployeesIn" />
  <wsdl:output message="tns:showEmployeesOut" />
</wsdl:operation>
- <wsdl:operation name="showJobs">
  <wsdl:input message="tns:showJobsIn" />
  <wsdl:output message="tns:showJobsOut" />
</wsdl:operation>
- <wsdl:operation name="showJobTime">
  <wsdl:input message="tns:showJobTimeIn" />
  <wsdl:output message="tns:showJobTimeOut" />
</wsdl:operation>
- <wsdl:operation name="showPastDueJobs">
```

---

Continued

**Figure 7.18 Continued**


---

```

    <wsdl:input message="tns:showPastDueJobsIn" />
    <wsdl:output message="tns:showPastDueJobsOut" />
  </wsdl:operation>
- <wsdl:operation name="showTrack">
  <wsdl:input message="tns:showTrackIn" />
  <wsdl:output message="tns:showTrackOut" />
</wsdl:operation>
</wsdl:portType>
- <wsdl:binding name="SXSBinding" type="tns:XSXPath">
  <soap:binding style="document" transport=
    "http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="LogIn">
  <soap:operation soapAction=
    "http://NLOGIX/track/soapSQL/LogIn" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="LogOut">
  <soap:operation soapAction=
    "http://NLOGIX/track/soapSQL/LogOut" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
- <wsdl:operation name="showEmployees">
  <soap:operation soapAction=
    "http://NLOGIX/track/soapSQL/showEmployees" style="document" />
- <wsdl:input>

```

---

**Continued**

## Figure 7.18 Continued

---

```
<soap:body use="literal" />
</wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="showJobs">
  <soap:operation soapAction=
    "http://NLOGIX/track/soapSQL/showJobs" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="showJobTime">
  <soap:operation soapAction=
    "http://NLOGIX/track/soapSQL/showJobTime" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="showPastDueJobs">
  <soap:operation soapAction=
    "http://NLOGIX/track/soapSQL/showPastDueJobs" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
  </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
  </wsdl:output>
```

---

Continued

**Figure 7.18 Continued**


---

```

    </wsdl:operation>
- <wsdl:operation name="showTrack">
    <soap:operation soapAction=
        "http://NLOGIX/track/soapSQL/showTrack" style="document" />
- <wsdl:input>
    <soap:body use="literal" />
    </wsdl:input>
- <wsdl:output>
    <soap:body use="literal" />
    </wsdl:output>
    </wsdl:operation>
    </wsdl:binding>
- <wsdl:service name="soapSQL">
- <wsdl:port name="SXSPort" binding="tns:SXSBinding">
    <soap:address location="http://NLOGIX/track/soapSQL" />
    </wsdl:port>
    </wsdl:service>
</wsdl:definitions>

```

---

Granted this file is generated by VS.NET, but you can also edit this file for a method name change or whatever you may need to do. This file consists of valid XML and has all the proper soap headers and protocols to communicate with our client application. This file is stored in the soap folder.

Let's look at the code behind the page and then explain each piece. All of these code snippets are available on the Solutions Web site for this book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

```

using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Web;
using System.Web.SessionState;
using System.Web.UI;
using System.Web.UI.WebControls;

```

```

using System.Web.UI.HtmlControls;

namespace TimeTracker
{
    /// <summary>
    /// Summary description for WebForm1.
    /// </summary>
    public class WebForm1 : System.Web.UI.Page
    {
        protected System.Web.UI.WebControls.DropDownList selPro;
        protected System.Web.UI.WebControls.DropDownList selEmp;
        protected System.Web.UI.WebControls.Button submit;
        protected System.Web.UI.WebControls.DataGrid dgTrack;
        protected DataSet dsE;
        protected DataSet dsJ;
        protected DataSet dsT;

        protected localhost.soapSQL proxyShow;
        protected int returnValue = 0;
        protected int returnValueT = 0;
        protected int returnValueJ = 0;
        protected int returnValueE = 0;

        protected DateTime now;

        private void Page_Load(object sender, System.EventArgs e)
        {
            if(!IsPostBack)
            {

                proxyShow = new localhost.soapSQL();

                dsJ = proxyShow.showJobs(out returnValueJ);
                selPro.DataSource = dsJ;
                selPro.DataBind();
            }
        }
    }
}

```

```

        dsE = proxyShow.showEmployees(out returnValueE);
        selEmp.DataSource = dsE;
        selEmp.DataBind();

        dsT = proxyShow.showTrack(out returnValueT);
        dgTrack.DataSource = dsT;
        dgTrack.DataBind();
    }
}

#region Web Form Designer generated code
override protected void OnInit(EventArgs e)
{
    //
    // CODEGEN: This call is required by the ASP.NET Web Form
Designer.
    //
    InitializeComponent();
    base.OnInit(e);
}

/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.submit.Click += new System.EventHandler(this.submit_Click);
    this.Load += new System.EventHandler(this.Page_Load);
}
#endregion

private void submit_Click(object sender, EventArgs e)
{
    now = DateTime.Now;

```

```

proxyShow = new localhost.soapSQL();

proxyShow.Login(int.Parse(selEmp.SelectedItem.Value),int.Parse
    (selPro.SelectedItem.Value),now,out returnValue);

dsT = proxyShow.showTrack(out returnValueT);
dgTrack.DataSource = dsT;
dgTrack.DataBind();

    }

}
}

```

The first key piece you need to have in place in order for this page to execute properly is to declare a variable of type *localhost.soapSQL*, which is just the name of the Web reference. This is done at the top of the page; by doing this you can use this variable through your class to instantiate all of the Web Service methods made available to you via SQL. If we look again at the *PageLoad* class in Figure 7.19 as part of the *start.aspx* code behind you can see this in action.



**Figure 7.19** start.aspx Code Behind

```

private void Page_Load(object sender, System.EventArgs e)
{
    if(!IsPostBack)
    {
        proxyShow = new localhost.soapSQL();
        dsJ = proxyShow.showJobs(out returnValueJ);
        selPro.DataSource = dsJ;
        selPro.DataBind();
        dsE = proxyShow.showEmployees(out returnValueE);
        selEmp.DataSource = dsE;
        selEmp.DataBind();
        dsT = proxyShow.showTrack(out returnValueT);
        dgTrack.DataSource = dsT;
        dgTrack.DataBind();
    }
}

```



Set the variable *proxyShow* equal to a new instance of the object. Now, set the value of the *DataSet dsJ* equal to the result set of your method *showJobs*. The method *showJobs* has a default output value for error handling with a try catch block checking to see if the returned value is anything other than a 1. Continue to do this for the rest of the data bound controls on the page.

This completes the default start page. The other pages are exact mirrors to this one, which are included in the Solutions Web site for this book at [www.syngress.com/solutions](http://www.syngress.com/solutions). We have left the application wide open to let you expand upon it to suit your own needs.



## Summary

We have gone through how to set up our SQL Virtual Root and target a specific database. We first set up our file system to mirror the virtual root, specifying the correct folders and names for each. From this we have been able to load and configure multiple stored procedures that we exposed as Web Services. We created methods within a class specific to each stored procedure within the database. This has, in a sense, tackled two problems at once relieving us from having to use ADO.NET to access the procs in our database in order to pass data

We viewed the WSDL file generated by SQL, also noting that it is an XML file that can be edited and revised without going through the whole method setup. This file contains all the SOAP (Simple Object Access Protocol) protocols necessary to communicate to the client application. These include the SOAP header and body and are contained within the SOAP envelope. We set each method to the same name as the stored procedure in order to keep it simple. Each method was also set to return a single dataset, which we then accessed via our client application that required a dataset for values.

In our client application we created three simple pages. The main page is the Log In page. We added our Web reference, which pointed to the virtual SQL root that we created. This exposed all of the methods contained within the WSDL file via SQL. We then bound the resulting datasets to the Web controls, for instance the data grid control that shows the listing of projects and who is logged in. We also dynamically joined drop-down boxes to display other data like our listing of projects and employees.

In conclusion we can see the many benefits of using SQLXML Web Services. We have looked at the possibility of upgrading current applications without redoing entire projects. This chapter also showed how you could use the versatility of SQLXML Web Services as opposed to using ADO (Active X Data Objects). This method of data interaction is cost effective in the way that a programmer already has to create and use stored procedures when dealing with a database, so to leverage that same code for use in the client application is a huge benefit. This technology is in the very early stages and will undoubtedly change in the future, but the overall fundamentals will be the same. This is truly the beginning of a new frontier for developers. Good luck and code well!

# Solutions Fast Track

## SQLXML Web Services

- ☑ SQLXML 3.0 is configured to work with .NET and will do the work of generating our WSDL file in order to process our stored procedures or User Defined Functions as Web Services.
- ☑ The main benefit is that it simplifies the interaction between SQL and multiple platforms and languages including various versions of Oledb (ADO), ODBC, JDBC, and so on; to better connect the Enterprise over an extranet, an intranet, or the Internet through the use of Active Server technologies (i.e., ASP, ASP.NET, JSP, PHP, etc.) and desktop or network applications built using various technologies (i.e., .NET, VB, C/C++, Java, etc.) on multiple platforms (Windows, UNIX, Linux, etc.).
- ☑ Using SQL Web Services is also a low-cost way to upgrade your current application to a .NET platform.

## Developing the TimeTrack Application

- ☑ If you choose, you may attach my database from the code section for Chapter 7 from the Solutions Web site ([www.syngress.com/solutions](http://www.syngress.com/solutions)). To attach a database to your sever, right-click on **Databases** in EM and select **Attach Database**.
- ☑ You want to use stored procedures because it is a huge performance gain over using ad hoc queries against the database. If you use stored procedures you will have precompiled and optimized results waiting for delivery as opposed to having SQL Server run through and parse the query and optimize it and then cache it and return the result set.

## Creating a SQL Server Virtual Directory

- ☑ SQLXML 3.0 enables you to be able to communicate to SQL Server via HTTP by creating a SQL Virtual Root Directory from one of your selected databases.
- ☑ With the virtual directory we can select the specific procs that we wish to use in our client application as Web Services.

- ☑ Creating a SQLXML virtual directory can also be accomplished through the IIS Virtual Directory Management for SQL Server object model.

## Creating a Client Application in ASP.NET

- ☑ Type your server name (most likely it will be (localhost)); specify the virtual root folder name, track, followed by the subfolder virtual name, soap; and add ?wsdl.
- ☑ WSDL is valid XML and has all the proper soap headers and protocols to communicate with our client application. This file is stored in the soap folder.

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** Is the security risk greater with SQLXML 3.0 versus ADO?

**A:** No, as long as you have followed the correct security guidelines designated by Microsoft for SQL Server.

**Q:** What is one of the key drawbacks of using Web Services over ADO?

**A:** Currently the use of Web Services opens the server, and in this case the database server, to denial of service attacks. This is currently an inherent drawback to the use of any URL access to a data source.

**Q:** Can I use my existing template query with SQLXML3?

**A:** Yes, you can upgrade these older template queries to run in the new environment or you can use them in their current form and have them work in the same instance side by side.

**Q:** Can I use SQLXML Web Services with my Legacy ASP 3.0 application?

**A:** Yes; in fact, it's a migration track that is used to upgrade a system without changing the front-end application.

**Q:** If I make a change to the SQL WSDL file do I have to reload the Web reference?

**A:** Yes; refresh the Web references within the project window and get the most recent version.

**Q:** Can I use SQLXML Web Services with a stand-alone application written in VB?

**A:** Yes; SQLXML Web Services can be accessed by any application that can interpret soap. Most importantly these Web Services can be accessed by any platform and are platform independent.

**Q:** Can I use my SQL Server 7 database stored procedures?

**A:** No, you must use SQL 2000 with SQLXML.

## Building a Jokes Web Service

### Solutions in this chapter:

- Motivation and Requirements for the Jokes Web Service
- Functional Application Design
- Implementing the Jokes Data Repository
- Implementing the Jokes Middle Tier
- Creating a Client Application
  
- ☑ Summary
- ☑ Solutions Fast Track
- ☑ Frequently Asked Questions

# Introduction

In this chapter, we show you—step-by-step—how to build a real-world Web Service, using all the tools and techniques covered in the previous chapters. This Web Service case study will take you through all the important factors to be considered when creating a Web Service application.

Together, we create a two-tier Web Service consisting of the following:

- A business logic layer (middle tier) written in C#
- A database backend using SQL Server 2000

We also show you how to access this service through a Windows Forms-based front-end portal application. While developing this application, we cover a range of subjects relevant to real-world Web Service projects. We start off by offering some suggestions for proper Web Service application architecture. We then discuss how to pass structured data from your Web Service to your client application, including basic marshalling and complex object serialization. We talk about namespaces and extended Web Service attributes, and how to properly use them. Further topics include how to secure Web Services, how to generate client proxies, error handling both on the server and on the client, working with Event Logs, and the automatic generation of documentation.

## Motivation and Requirements for the Jokes Web Service

In the case study presented by this chapter, we won't be showing you an ordering application for buying or selling anything, instead we're giving away free content in the form of jokes. Think of our application of the future as a modern version of the venerable Quote Of The Day (*quotd*) Internet service. *quotd* has been around for almost two decades, used mostly as a TCP/IP test service (see [www.faqs.org/rfcs/rfc865.html](http://www.faqs.org/rfcs/rfc865.html)). It runs on port 17, and all it does is send you an uplifting quote of some wise dead person, before closing the connection again. You can install it as part of the so-called “Simple TCP/IP Services” through **Control Panel | Add/Remove Programs | Add/Remove Windows Components | Networking Services**. Many servers on the Internet still have this service installed, even though it has maybe fallen out of favor in recent years; for an example, simply use Telnet to establish a TCP connection to port 17 of server 209.21.91.3.

Let's try to formulate some design goals for the Jokes Web Service, and see how they compare with what was possible twenty years ago when *quotd* was designed:

- Although we still give away free content, we would like to know who our users are! Hence, there should be some sort of registration process.
- We want to be highly interactive. We are interested in user feedback for particular jokes, and our users should also be able to add content—that is, jokes—of their own. However, too much interactivity can be a dangerous thing, so there should be moderators standing by to remove objectionable content.
- *quotd* is essentially a 7-bit ASCII service (in fact it's limited to the 94 printable ASCII characters). That's great if you live in the U.S., but even Europeans will already be a little bit annoyed at you, because their accented characters will get lost, and users in Asia won't be able to use the service in their native language. Clearly, we want *our* updated service to fully support Unicode.
- Our service should be universally accessible. *quotd* is usually blocked by firewalls because it uses a nonstandard port.

To summarize, we would like to develop a Web Service that delivers jokes to registered users, has portal functionality to let users register, and allows them to submit their own jokes. Moreover, we want a mechanism for registered users to rate jokes, say on a scale from 1 to 5. Finally, there should be a class of super users, called moderators, who should be able to administer both users and jokes.

Note that we get support for international users and universal accessibility for free by using Web Services technology:

- Because Web Services are based on XML, we can ensure Unicode support by specifying, say, UTF-8 as our underlying character set (which is the default, anyway). Also, we need to ensure, of course, that our data repository can hold Unicode information.
- Because Web Services usually run on either port 80 (HTTP) or port 443 (HTTPS), firewalls should not be a problem, and clients should be able to establish a connection to our server. However, when designing the service, we also need to ensure that the data we transport through SOAP can easily be read by potential clients, particularly if they run on non-Microsoft systems. We talk about this issue more when we go about sending SQL record data through SOAP.



# Functional Application Design

Coming up with a good application design is critically important for the success of any software application. The first step is to move top-down from goals to design by starting to define (in still very general terms) the functionality exposed by the Jokes service, and then developing a back-end schema that supports that functionality from a data perspective. In a second step, we then create in more detail an object model suitable to implement the services the Jokes application is supposed to provide. At this juncture, it is also appropriate to make decisions about security, state management, and error handling.

## Defining Public Methods

Let's start the application design process by writing down the specific methods we think our Jokes Web Service should offer, and the categories that reflect their function:

The application needs methods dealing with user administration:

- **addUser** Lets a user register to our service.
- **addModerator** Lets a moderator add an existing user to become a moderator.
- **checkUser** Verifies that a user has previously registered with the service. Refer to the “State Management” section to see why this is a useful method for the service to expose.

Then, the application needs methods dealing with delivering and managing jokes:

- **addJoke** Lets a registered user add a joke.
- **getJokes**: Delivers some randomly selected jokes, say up to 10 per request, to our registered users.
- **addRating** Lets our users add a rating to a joke, say on a scale of 1 to 5.
- **getUnmoderated** Registered moderators can call this method to get the jokes added by the users for moderation.
- **addModerated** If moderators agree to add a joke to the database, they can use this method.
- **deletedUnmoderated** If a submitted joke is considered offensive, a moderator should be able to delete it.

## Defining the Database Schema

Let's define the database schema for the Jokes Web Service: The Jokes database supports three basic data entities: Users, jokes, and joke ratings. We therefore define the corresponding tables as follows:

- **users** A table containing user information.
- **jokes** A table containing joke information.
- **ratings** A table containing joke rating information.

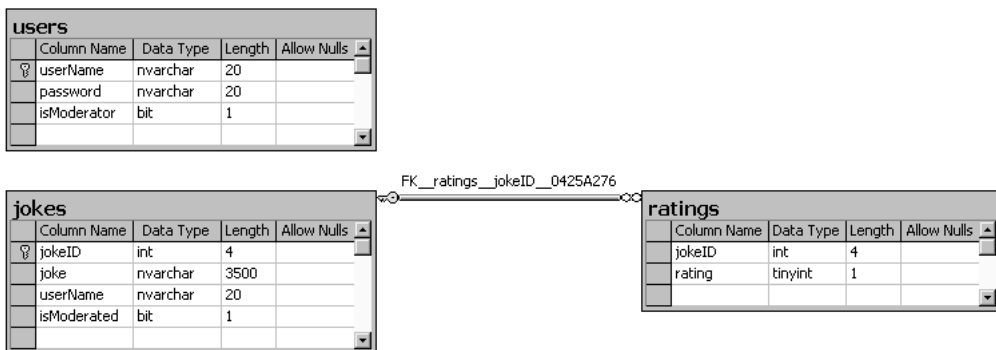
To keep things simple, all we want to know about our users are their usernames, their passwords, and whether they are moderators or not. We limit both usernames and passwords to 20 Unicode characters. We add a primary key constraint to usernames to speed lookup access and to ensure that usernames are unique.

For the jokes table, we record the username and the actual joke, which we want to be 3,500 Unicode characters or less, keeping SQL Server 2000 limitations on row size in mind. We give each joke a unique identifier through an identity column. Note that we don't relate the users and the jokes table with each other, because users may choose to unsubscribe from the service (but we sure want to keep their jokes!).

Finally, we add a rating column to the ratings table and relate the jokes to the ratings table through a one-to-many relationship on the unique joke identifier.

Let's look at a visual representation of our jokes database (see Figure 8.1).

**Figure 8.1** The Jokes Database Tables



## Defining the Web Service Architecture

Typically, the actual Web Service layer will be a very small layer of your application. You expose the Web Service methods to your clients, but leave the implementation of those methods to internal implementation classes. The advantage of this architecture is that you can then always change the implementation of your Web Services in the future, while keeping the Web Service interface stable.

Nothing is more annoying to consumers of your service (your business clients, that is) than if a change in your server-side code requires them to rewrite their applications. Also, typically, you will already have code on your servers that handles most or all of the business logic required to process client requests; this could be code to access legacy systems or enterprise data. You then simply wrap this already existing code in a lean layer of Web Service access code.

In our example of the Jokes Web Service, we are going to define two Web Services, one to handle the portal aspects of our application, that is managing users and moderators, and a second one dealing with managing and retrieving the actual jokes. We could, of course, collapse these two Web Services easily into one larger service, and there are certainly good arguments for doing so, but keeping the two services apart allows us to architect our application in a nice, symmetric way.

We then define the two corresponding implementation classes, one for user administration, and the other one for handling the jokes. Additionally, we need classes for error management and database access, and a class that allows us to return structured data containing our jokes to clients of our service.

To visualize the architecture, you can use a tool such as Microsoft Visual Modeler. The UML diagram of the class structure looks as follows, ignoring method signature and a few other details, such as destruction methods you don't care about too much at this point (see Figure 8.2).

Let us first look at the details of the *userAdmin* Web Service (see Figure 8.3).

As you can see in the Figure, the *userAdmin* class, which exposes the Web Service of the same name, has methods to add a new user, make an existing user become a moderator, and verify that a given user does in fact exist in the system. The class *userAdminImplement* contains implementations of the corresponding methods, and also contains methods that wrap the SQL stored procedures we defined in the previous section.

Now take a look at the details of the Jokes Web Service in Figure 8.4.

Figure 8.2 UML Diagram of *jokesService* Middle Tier Architecture

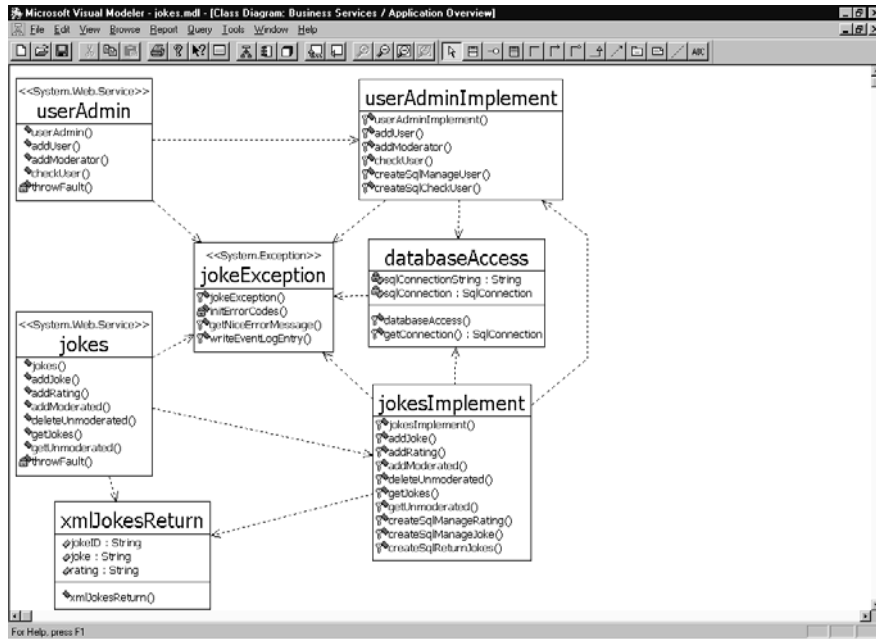


Figure 8.3 Detailed UML Diagram of *userAdmin* Web Service

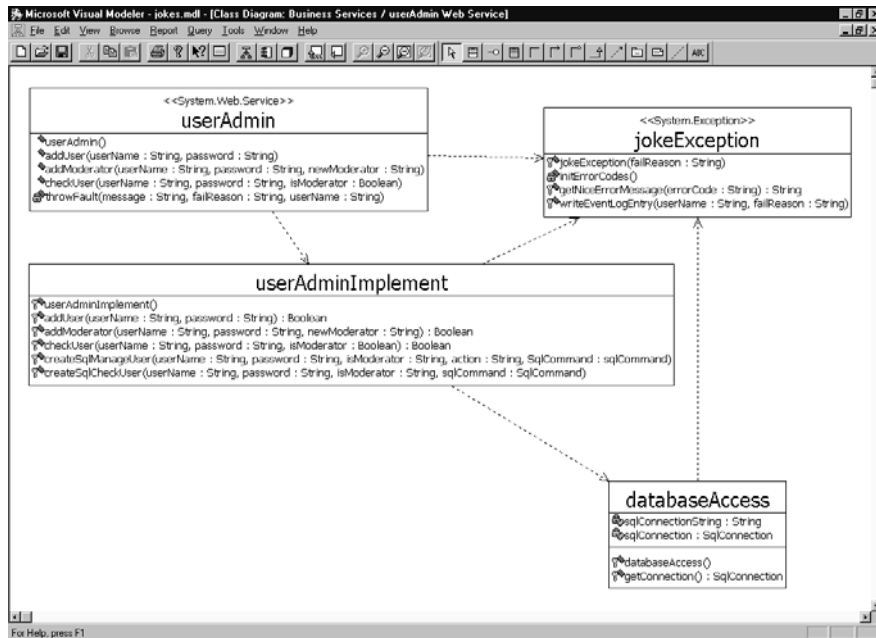
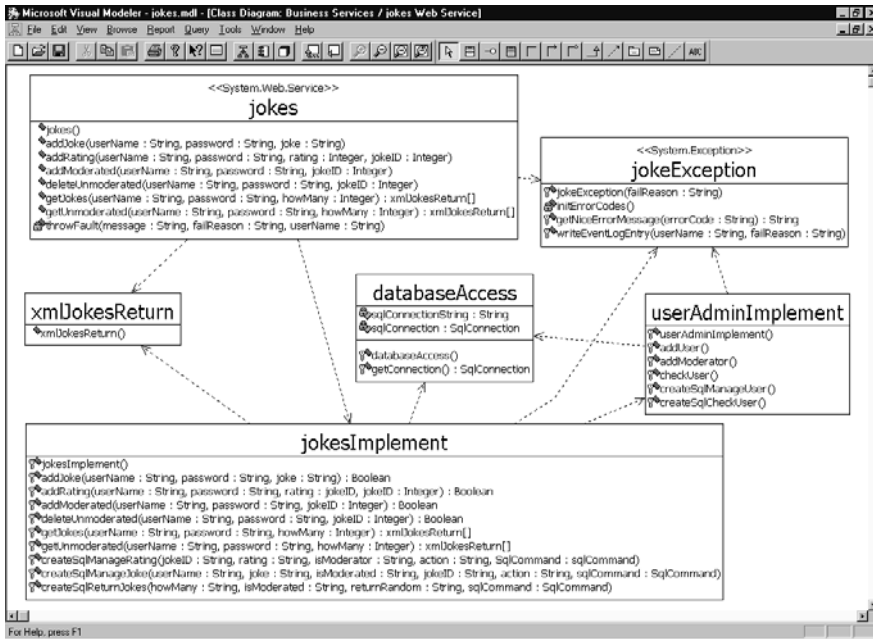


Figure 8.4 Detailed UML Diagram of the Jokes Web Service



The *jokes* class, which exposes the Web Service of the same name, has methods to add, manage, and retrieve jokes. The class *jokesImplement* contains implementations of the corresponding methods, and also contains methods that wrap the SQL stored procedures we defined in the previous section. Before we continue, let us briefly talk about security, state management, and error handling.

## Security Considerations

Our Web Service will be wide open to the world, which of course we like to think as being a good thing. As we would like to have control over who is accessing our application, the first thing we have to do for each request is to check if the requesting client is a registered user. That's why all of our public methods have *userName* and *password* as arguments. User look-ups are done in the *userAdminImplement* class, and therefore the very first thing the *jokesImplement* class does is to call the *userAdminImplement* class to check if the credentials passed match a credential in the database.

Now, we can cheat a little bit and pretend the Web has state. For instance, we can and will create a client application for our Jokes Web Service that will remember the user's credentials. Using the *checkUser* method in the *userAdmin* class, we can let our users log on, and then simply cache the user name and

password on the client. While that information still needs to be sent to the server with every single request, at least our clients don't need to input it again during the duration of a "session" with the Jokes application.

Obviously, this means that user names and passwords are sent in clear text over the wire. If this is of concern (it probably should be!), then we need to encrypt either the whole data transfer (by using, for example, a secure channel over HTTPS), or at least the confidential parts of the message like the password.

## State Management

Stateful Web Service applications should almost always be avoided. The only reason for the Jokes Web Service to be stateful would be to support client sessions in order to simplify authentication and authorization to the service. However, a better way to deal with security for this particular application seems to us to store user credentials in the Web Service client, as described in the preceding paragraph.

## Error Handling

For error handling, we would like to have more control over what happens during program execution than the standard *System.Exception* class gives us; in particular, we would like to gather enough information so that we can give a meaningful, user-friendly error message to our clients. The *jokeException* class, which extends *System.Exception*, is designed to do exactly that. We will encounter more details on proper error handling as we go about implementing this class.

## Implementing the Jokes Data Repository

Now that the structure of the Jokes Web Service is firmly in place we can start the work of actual implementation. It is usually a good idea to start with the back end and spend a fair amount of time fleshing out the exact interface to store and retrieve data.

We will start off by installing the actual database system. We will then set up our data tables using a SQL installation script, before writing all the stored procedures needed to manage our jokes in the database.

### WARNING

Later changes in methods exposed by the back end almost always require major rewrites of the whole application, so it really pays to be very careful when writing your back end methods.

## Installing the Database

The first step in working with a back end is of course to actually have a back end to work with... Because we offer dynamic content, a simple flat file approach probably won't scale very well. Instead, let us use a relational database, such as SQL Server 2000. If you don't have a copy of this server, you're in luck, because the Microsoft .NET SDK actually comes with its own copy of Microsoft SQL Server Desktop Engine, a slightly scaled down version of the full server product, which is more than sufficient for our purposes. To install it, proceed as follows:

- Open up %ProgramFiles%\Microsoft.NET\Microsoft.NET\FrameworkSDK\Samples\setup\html\Start.htm.
- Click on **Step 1: Install the .NET Framework Samples Database** and follow the instructions.
- Verify in the list of services on your computer that the services MSSQL\$NetSDK and SQLAgent\$NetSDK are up and running.

This will install the SQL Server Desktop Engine, and configure the .NET SDK database instance.

Note that SQL Server Desktop Engine does not come with any of the standard GUI client tools. But it does ship with *osql*, a command line utility, which is certainly sufficient for what we try to do. *osql* is described in detail in the Visual Studio.NET Combined Help Collection, but all we really need to know is how to execute a SQL command script, which is done as follows:

```
osql -S (local)\NetSDK -U sa -P -i myScript.sql
```

However, we can compensate for this lack of user friendliness by using the Server Explorer tool in Visual Studio.NET, to which we will get at soon.

First, we give ourselves a database to work with, which we fittingly call “jokes.” Run the following SQL script:

```
create database jokes
go
```

Now we can go about setting up the data tables as defined in Figure 8.5. Also, to bootstrap our system, we prepopulate our *users* database with a default moderator, which we call “admin,” with password “secret”. We also include a first joke, so that we can show our first user something. See Figure 8.5 for the complete listing of the database installation script. The complete database installation

script is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 8.5** The Database Installation Script (installJokes.sql)

---

```
use jokes
go

/* object: table [dbo].[users] */
create table [dbo].[users] (
    [userName] [nvarchar] (20) not null primary key,
    [password] [nvarchar] (20) not null ,
    [isModerator] [bit] not null
) on [primary]
go

/* object: table [dbo].[jokes] */
create table [dbo].[jokes] (
    [jokeID] [int] identity(1,1) primary key ,
    [joke] [nvarchar] (3500) not null ,
    [userName] [nvarchar] (20) not null ,
    [isModerated] [bit] not null ,
) on [primary]
go

/* object: table [dbo].[ratings] */
create table [dbo].[ratings] (
    [jokeID] [int] not null references jokes(jokeID),
    [rating] [tinyint] not null,
) on [primary]
go

create index "jokeID" on [dbo].[ratings](jokeID)
go

/* insert data into users table */
insert into users (userName,password, isModerator) values
```

---

Continued



**Figure 8.5 Continued**


---

```

("admin","secret", 1)
go

/* insert data into jokes table */
insert into jokes (joke,userName, isModerator) values
    ("Have you heard about the new sushi bar that caters exclusively
to lawyers? --It's called, Sosumi.,"admin", 1)
go

```

---

Once we've created our Web Service project, we'll be able to look at our database right through the Visual Studio.NET IDE (from which the database diagram in Figure 8.1 is taken). Also, if you don't like working with SQL command line scripts, you can create this database through the Visual Studio.NET Server Explorer, but by doing so, you probably open yourself up to errors when setting up your back end manually. Also, you can only write out SQL Create Scripts from Visual Studio.NET if you have the SQL Server client tools installed, which don't come with the SQL Server Desktop Engine—you have to purchase these separately.

## Creating the Stored Procedures

Now that you have defined and implemented the database schema, you need to develop the stored procedures to manage your data, which will be used by the Web Service business components. You need to be able to add, modify, and possibly delete users, jokes, and joke ratings. The Jokes service is so simple that you may be tempted to just hard code the corresponding SQL statements directly in your business components, but of course, you know that is a beginner's mistake, and that you will never get away with doing that in a real-world application. Because this example should show how to write a real application, you should do things the right way and create the corresponding stored procedures.

Right from the start, you want to have a comprehensive error-handling mechanism in place. Therefore all our stored procedures have a return argument that carries a string-valued return code determined by what's happening during execution of the stored procedures back to the calling function in the middle tier. This return parameter is called, simply enough, *return*. In considering what can possibly go wrong during a stored procedure call, you may come up with the following values shown in Table 8.1:

**Table 8.1** Uniform Stored Procedure Return Codes

Status/error code	User-friendly message
<i>S_OK</i>	Operation completed successfully.
<i>F_ratingInvalid</i>	Joke rating must be between 1 and 5.
<i>F_jokeDoesNotExist</i>	Joke selected does not exist in the system.
<i>F_unknownAction</i>	Internal error when accessing the database.
<i>F_userDoesNotExist</i>	This is not a registered user.
<i>F_userExists</i>	Somebody has already registered under this name.
<i>F_userInfoWrong</i>	You are not authorized to do this action. Change user name or password.
<i>F_noJokes</i>	No matching jokes in the system at this moment in time.

Make a note, then, that we will need a method that's part of the common error-handling procedure used by the middle tier that will translate error codes coming from the database (and elsewhere) into user-friendly messages sent back to the clients of the Web Service.

The errors defined in Table 8.1 are exceptions caught by our code—that's why you are able to return an error code in the first place. Errors may occur over which you have little control, and which cause the stored procedure to abort. In that case, all you can do is catch the exception in the middle tier and return an “unknown system error” back to the clients of the Web Service.

Secondly, in order to minimize the amount of code, you can employ a mechanism by which you tell the stored procedure what action you want to have done on a table, such as *add*, *modify*, or *delete*. That's why three of the stored procedures have an *action* input parameter indicating the action to perform.

In the upcoming section “Implementing the Jokes Middle Tier,” we will talk more about security. For now, let's simply assume that all access checks happen *before* program execution reaches a stored procedure, so that at this point you don't need to check on permissions any more.

To make the Jokes Web Service possible, we define the following five stored procedures, which are detailed in Tables 8.2, 8.3, 8.4, 8.5, and 8.6.

**Table 8.2** Stored Procedure *sp\_manageUser*

Name	<b>sp_manageUser</b>
Purpose	Allows you to add, modify, or delete a user.
Input parameters	<i>username</i> The user name to <i>add, modify, or delete</i> . <i>password</i> The corresponding password. <i>isModerator</i> A Boolean value that tells you if this is a moderator or not. <i>action</i> What to do: <i>add or modify or delete</i> .
Output parameters	<i>return</i> Status/error code.
Returns	Standard SQL numerical return code.

**Table 8.3** Stored Procedure *sp\_checkUser*

Name	<b>sp_checkUser</b>
Purpose	Allows you to check the user information provided in the arguments against information stored in the database.
Input parameters	<i>username</i> The user name to verify <i>password</i> The corresponding password <i>isModerator</i> A Boolean value that tells you if this is supposedly a moderator or not.
Output parameters	<i>return</i> Status/error code.
Returns	Standard SQL numerical return code.

**Table 8.4** Stored Procedure *sp\_manageJoke*

Name	<b>sp_manageJoke</b>
Purpose	Allows you to add, modify, or delete a joke.
Input parameters	<i>username</i> The user name of the registered user (used when adding a joke). <i>joke</i> The actual joke (used when adding a joke). <i>isModerated</i> A Boolean value that tells you if this joke is moderated or not. <i>jokeID</i> The unique identifier of the joke (used when modifying or deleting a joke). <i>action</i> What to do: <i>add or modify or delete</i> .
Output parameters	<i>return</i> Status/error code.
Returns	Standard SQL numerical return code.

**Table 8.5** Stored Procedure *sp\_manageRating*

Name	<b>sp_manageRating</b>
Purpose	Allows you to add a rating for a joke.
Input parameters	<i>jokeID</i> The unique identifier of the joke. <i>rating</i> The rating, from 1 to 5, the joke gets. <i>action</i> What to do: <i>add</i> or <i>delete</i> .
Output parameters	<i>return</i> Status/error code.
Returns	Standard SQL numerical return code.

**Table 8.6** Stored Procedure *sp\_returnJokes*

Name	<b>sp_returnJokes</b>
Purpose	Allows you to return jokes.
Input parameters	<i>howMany</i> How many jokes you want to return. <i>isModerated</i> : A Boolean value that allows you to specify whether you want moderated or unmoderated jokes (or both, if null). <i>returnRandom</i> A Boolean value that allows you to specify whether you want to get randomly selected jokes (for users) or not (for moderators when reviewing unmoderated jokes).
Output parameters	<i>return</i> Status/error code.
Returns	A record set

Some of the stored procedures have what amounts to optional parameters; for example, in order to delete a joke, you only need to pass the corresponding unique identifier of the joke to delete, along with the *action* parameter set to *delete* to *sp\_manageJoke*. Because T-SQL does not allow you to overload stored procedure calls, you can simply pass null references to the remaining input parameters, and remember to set up your middle tier code accordingly. Figure 8.6 shows the part of the SQL installation script that sets up the stored procedure needed by the Jokes Web Service. The complete script is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.6** Setting up the Stored Procedures (installJokes.sql)

```

use jokes
go

/* Create stored procedures */
create procedure sp_manageUser (
  -- add, modify, or delete a user
  @@userName nvarchar(20),
  @@password nvarchar(20),
  @@isModerator bit,
  @@action nvarchar(20),      -- one of 'add' or 'modify' or 'delete'
  -- returns:
  -- 'S_OK'                : success
  -- 'F_userExists'        : failed: user already exists
  -- 'F_userDoesNotExist': failed: user does not exist
  -- 'F_unknownAction'    : action command unrecognized
  @@return nvarchar(20) output
) as

declare @@userCount int
select @@userCount = count(*) from users where userName = @@userName

-- sanity checks
if (@@userCount = 0 and ((@@action = 'modify') or
  (@@action = 'delete'))))
begin
  select @@return = 'F_userDoesNotExist'
  return
end

if @@userCount = 1 and @@action = 'add'
begin
  select @@return = 'F_userExists'
  return
end

```

---

**Continued**

**Figure 8.6 Continued**

---

```
-- start
if @@action = 'add'
    begin
        insert into users (userName,password,isModerator)
            values (@@userName,@@password,@@isModerator)
        select @@return = 'S_OK'
        return
    end

if @@action = 'delete'
    begin
        delete from users where userName = @@userName
        select @@return = 'S_OK'
        return
    end

if @@action = 'modify'
    begin
        update users
            set userName = @@userName,
            isModerator = @@isModerator
        where userName = @@userName
        if @@password is not null
            update users
                set password = @@password
            where userName = @@userName
        select @@return = 'S_OK'
        return
    end

-- otherwise
select @@return = 'F_unknownAction'
return
go
```

---

Continued

**Figure 8.6 Continued**

---

```

create procedure sp_checkUser (
    -- checks user information provided against information in
    -- the database
    @@userName nvarchar(20),
    @@password nvarchar(20),
    @@isModerator bit,
    -- returns:
    -- 'S_OK'           : information matches
    -- 'F_userInfoWrong' : information does not match
    @@return nvarchar(20) output
) as

declare @@userCount int

-- sanity checks
if @@userName is null
begin
    select @@return = 'F_userInfoWrong'
    return
end

-- start
if @@password is null and @@isModerator is null
begin
    select @@userCount = count(*) from users where
        userName = @@userName
    goto checkCount
end

if @@isModerator is null
begin
    select @@userCount = count(*) from users where
        userName = @@userName and password = @@password
    goto checkCount
end

```

---

**Continued**

**Figure 8.6 Continued**

---

```
if @@password is null
begin
    select @@userCount = count(*) from users where
        userName = @@userName and isModerator = @@isModerator
    goto checkCount
end

select @@userCount = count(*) from users where userName = @@userName
    and password = @@password and isModerator = @@isModerator

checkCount:
if @@userCount = 0
begin
    select @@return = 'F_userInfoWrong'
    return
end

select @@return = 'S_OK'
return

go

create procedure sp_manageRating (
    -- add a joke rating
    @@jokeID int,
    @@rating tinyint,
    @@action nvarchar(20),      -- one of 'add' or 'delete'
    -- returns:
    -- 'S_OK' : success
    -- 'F_jokeDoesNotExist': failed: joke does not exist
    -- 'F_unknownAction' : action command unrecognized
    @@return nvarchar(20) output
) as

-- sanity checks on arguments done in middle tier
```

---

**Continued**



**Figure 8.6 Continued**

---

```
declare @@jokeCount int

-- does the joke even exist?
select @@jokeCount = count(*) from jokes where jokeID = @@jokeID
if @@jokeCount = 0
begin
    select @@return = 'F_jokeDoesNotExist'
    return
end

if @@action = 'add'
begin
    insert into ratings (jokeID,rating) values (@@jokeID,@@rating)
    select @@return = 'S_OK'
    return
end

if @@action = 'delete'
begin
    delete from ratings where jokeID = @@jokeID
    select @@return = 'S_OK'
    return
end

-- otherwise
select @@return = 'F_unknownAction'
return
go

create procedure sp_manageJoke (
    -- add, modify, or delete a joke
    @@userName nvarchar(20),
    @@joke nvarchar(3500),
    @@isModerated bit,
    @@jokeID int,
```

---

**Continued**

**Figure 8.6 Continued**

---

```
@@action nvarchar(20),      -- one of 'add' or 'modify' or 'delete'
-- returns:
-- 'S_OK'                   : success
-- 'F_jokeDoesNotExist': failed: joke does not exist
-- 'F_unknownAction'      : action command unrecognized
@@return nvarchar(20) output
    ) as

-- sanity checks on arguments done in middle tier

declare @@jokeCount int

if @@action = 'add'
begin
    insert into jokes (userName,joke,isModerated)
        values (@@userName,@@joke,@@isModerated)
    select @@return = 'S_OK'
    return
end

if @@action = 'modify'
begin
    select @@jokeCount = count(*) from jokes where jokeID = @@jokeID
    if @@jokeCount = 0
        begin
            select @@return = 'F_jokeDoesNotExist'
            return
        end
    if @@isModerated is not null
        update jokes
            set isModerated = @@isModerated
            where jokeID = @@jokeID
    if @@userName is not null
        update jokes
            set userName = @@userName
```

---

Continued

**Figure 8.6 Continued**


---

```

        where jokeID = @@jokeID
    if @@joke is not null
        update jokes
            set joke = @@joke
            where jokeID = @@jokeID
        select @@return = 'S_OK'
    return
end

if @@action = 'delete'
    begin
        select @@jokeCount = count(*) from jokes where jokeID = @@jokeID
        if @@jokeCount = 0
            begin
                select @@return = 'F_jokeDoesNotExist'
                return
            end
        declare @@dummy nvarchar(40)
        execute sp_manageRating @@jokeID, null, 'delete', @@dummy output
        delete from jokes where jokeID = @@jokeID
        select @@return = 'S_OK'
        return
    end

-- otherwise
select @@return = 'F_unknownAction'
return
go

create procedure sp_returnJokes (
    -- returns jokes
    @@howMany int,
    @@isModerated bit,
    @@returnRandom bit
    -- returns a recordset containing jokeID, joke, and average rating
) as

```

---

**Continued**

**Figure 8.6 Continued**

---

```
-- sanity checks on arguments done in middle tier

declare @@jokeCount int
declare @baseJokeID int
declare @baseJokeRelPos int
declare @cmd varchar(1000)

-- random start position?
-- note that in this case, we implicitly assume that
-- * isModerated = 1
-- * howMany <> null
if @@returnRandom = 1
begin
    select @@jokeCount = count(*) from jokes where isModerated = 1
    if @@jokeCount = 0
        return

    if @@jokeCount < @@howMany
        set @@howMany = @@jokeCount

    -- get a random number between 0 and 1
    declare @random decimal(6,3)
    set @random = cast(datepart(ms, getdate()) as decimal(6,3))/1000

    -- set a random start position
    set @baseJokeRelPos =
        ((@jokeCount - @howMany + 1) * @random) + 1

    -- get the corresponding jokeID
    declare jokeTempCursor cursor scroll for select jokeID from
        jokes where isModerated = 1 order by jokeID
    open jokeTempCursor
    fetch absolute @baseJokeRelPos from jokeTempCursor
    into @baseJokeID
```

---

Continued

**Figure 8.6 Continued**

---

```

        close jokeTempCursor
        deallocate jokeTempCursor
    end

-- start building our command
set @cmd = 'select '

if @@howMany is not null
    set @cmd = @cmd + 'top ' + cast(@@howMany as varchar(10)) + ' '

set @cmd = @cmd + 'jokes.jokeID, left(ltrim(joke),3500) '
set @cmd = @cmd + ', cast(avg(cast(rating as decimal(5,4)))
    as decimal(2,1)) '
set @cmd = @cmd + 'from jokes left outer join ratings on
    jokes.jokeID = ratings.jokeID '

if @@isModerated is not null
    begin
        if @@isModerated = 1
            begin
                set @cmd = @cmd + 'where isModerated = 1 '
                if @@returnRandom = 1
                    set @cmd = @cmd + 'and jokes.jokeID >= ' +
                        cast(@baseJokeID as varchar(10)) + ' '
            end
        if @@isModerated = 0
            set @cmd = @cmd + 'where isModerated = 0 '
    end

set @cmd = @cmd + 'group by jokes.jokeID, joke order by jokes.jokeID'

exec (@cmd)
go

```

---

That completes setting up the back-end infrastructure. You can find the complete installation script in directory `SQLSetup/` on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

We are now ready to start up Visual Studio.NET to begin working on the meat of the Web Service, namely the Web Service itself.



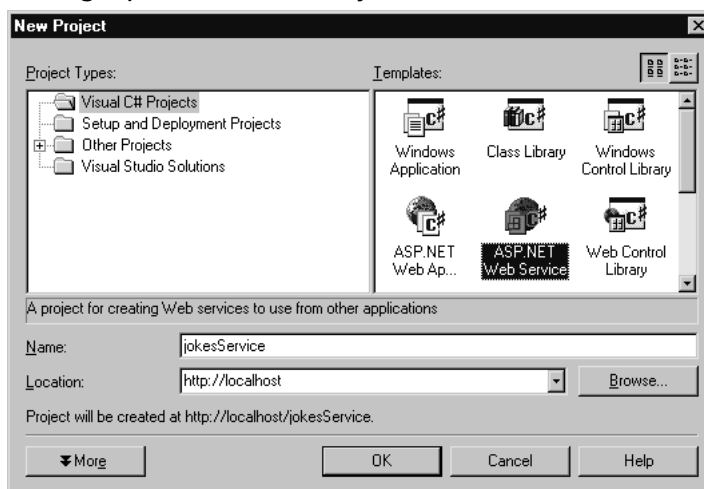
## Implementing the Jokes Middle Tier

Now that you have the back-end database system in place, you can go about implementing the actual Web Service that clients will be calling. Of course, you will want to do this work in Visual Studio.NET. Note that you can find the complete code for this project on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

### Setting Up the Visual Studio Project

Start the setup of the Visual Studio project by creating a new ASP.NET Web Service project, called *jokesService*. Go to **File | New | Project**, choose the entry **ASP.NET Web Service** under the Visual C# Projects folder, keep the default Location, and enter **jokesService** as the Name of the project (Figure 8.7).

**Figure 8.7** Setting Up a New Web Project

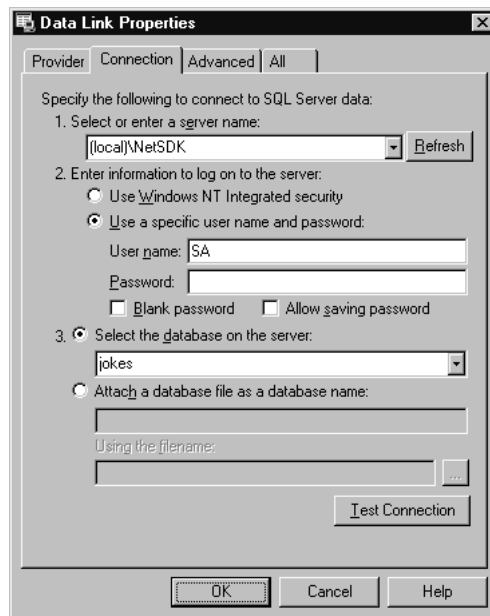


This will set up a new virtual directory of the same name, configure the necessary FrontPage server extensions, define an assembly, and create supporting project files.

Rather annoyingly, the ASP.NET Web Service wizard creates a default Web Service called Service1, which you may want to remove from the project right away (or rename it later when we go about adding Web Services to our project).

Next, check on the database we created earlier: Click on **Server Explorer**, which by default is on the upper left hand corner of the window. Right click under **Data Connections**, and enter the connection information for the .NET SDK database as follows: Under Server enter **(local)\NetSDK**, the user name is **SA**, no Password, and the Database we are interested in is **jokes** (Figure 8.8).

**Figure 8.8** Opening Up A Connection to the Jokes Database



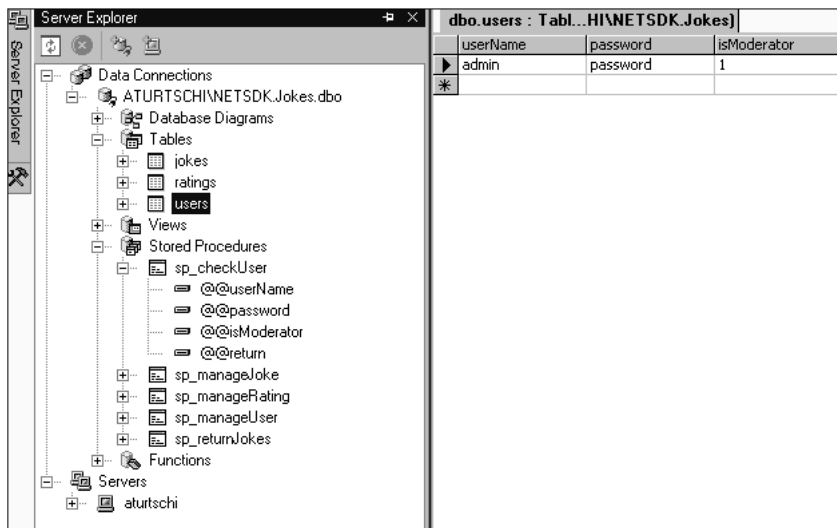
The connection is then added to Server Explorer, and you can go about exploring your database, and, say, look at your *users* table (Figure 8.9).

Now you are in a position to create the two Web Services: Right click on the **jokesService** project in the Solution Explorer, and choose **Add | Add New Item**. Choose **Web Service** from the list of available templates, and call it **userAdmin.asmx** (Figure 8.10).

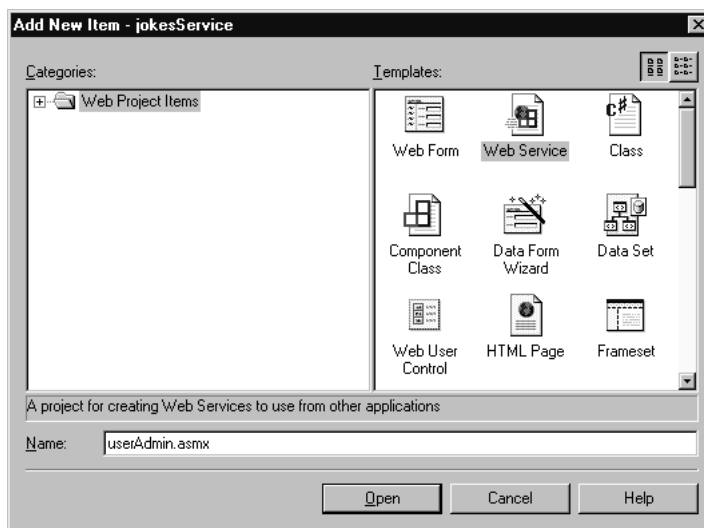
Note that apart from creating the ASMX file, this will also create the corresponding C# class file *userAdmin.asmx.cs*, and the resource file *userAdmin.asmx.resx*.

Perform the same step for the second service, called *jokes.asmx*.

**Figure 8.9** Exploring the Jokes Database through Visual Studio.NET Server Explorer



**Figure 8.10** Adding a New Web Service



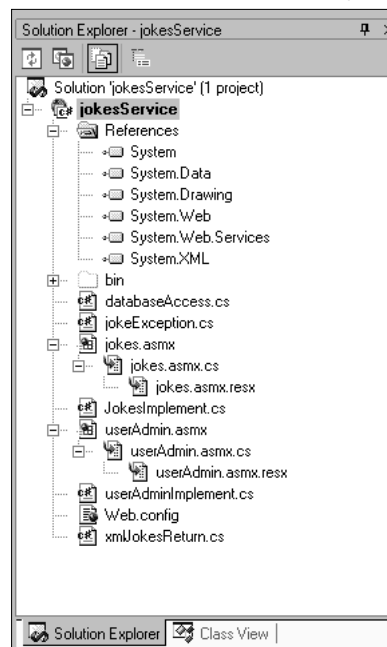
Next, you need to set up the supporting classes. Again, right click on the **jokesService** project in the Solution Explorer, and choose **Add | Add New Item**, but this time select **Class** instead. Repeat this procedure five times, for the five C# classes you need:



- *userAdminImplement.cs*
- *JokesImplement.cs*
- *databaseAccess.cs*
- *jokeException.cs*
- *xmlJokesReturn.cs*

When looking at the Solution Explorer, and clicking on the **Select All Files** icon, your project should now look like the one shown in Figure 8.11.

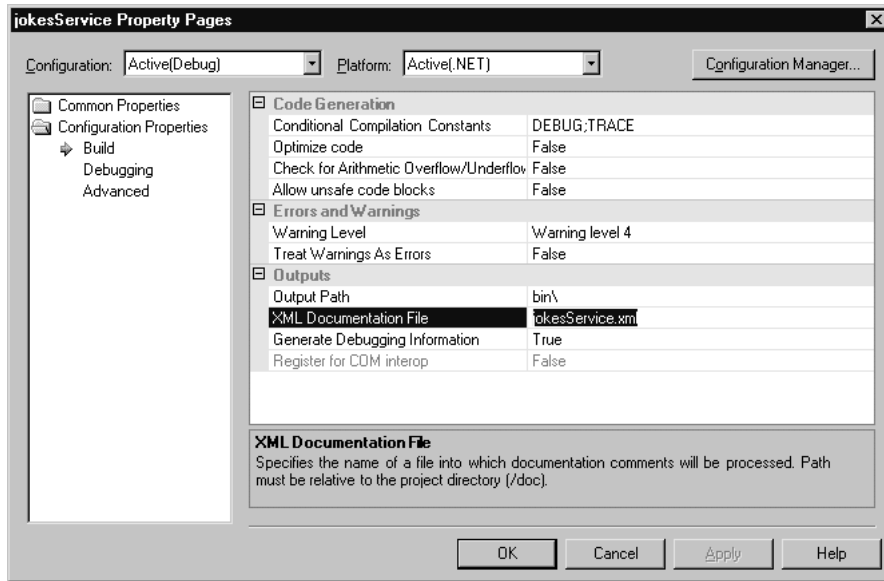
**Figure 8.11** Overview of All Files Needed for the *jokesService* Web Service



Lastly, you need to instruct the C# compiler to automatically generate an XML documentation file of your work for you (see the “Making Documentation Part of Your Daily Life” sidebar). Go to the Solution Explorer, right click on the **jokesService** project, and select **Properties**. A dialog will open, as shown in Figure 8.12. Select the **Build** option under the Configuration Properties folder, and enter **jokesService.xml** as the XML Documentation File name.

Now you can code away. Note that the complete code for the Jokes Web Service can be found in *jokesService* directory on the on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

Figure 8.12 Automatically Generating XML Documentation Output



## Developing & Deploying...

### Making Documentation a Part of Your Everyday Life

Documenting your work does not need to be an afterthought—in fact, it should occupy center-stage of your work from the very beginning of a project. The Visual Studio.NET environment supports this philosophy by offering you a set of predefined XML elements allowing you to document your code inside your source files as you are developing it.

This functionality is still rather limited, quite frankly, but it is a start. Among others, there are currently tags defined to describe the function of a class or method (`<summary>`), and what parameters (`<param>`) and return values (`<return>`) a method has. But you are certainly free to add your own set of tags, suitable for your needs. The C# compiler then allows you to extract your XML documentation into a separate XML output file, which you can then use for further processing, for instance to create documentation in HTML format by applying a suitable XSLT style sheet. The compiler validates some of the XML documentation tags for you, such as those describing the method input parameters. You can find more information in the XML Documentation Tutorial that's part of the Visual Studio.NET C# Programmer's Reference.

Continued

Because documentation is vitally important for the success of any software project, all of our code for the Jokes Web Service application uses the C# documentation tags liberally.

## Developing the Error Handler

Introducing error handling as you start to code is a good thing. However, you need to have a good idea first as to what could possibly go wrong. In the section on setting up the SQL stored procedures we have already identified a number of errors that can be caught at the database level. As user input data validation checking is done in our business components, you get two more possible exceptions, having to do with invalid ratings (should be between 1 and 5), and requests for “too many” jokes (should be between 1 and 10). Obviously, when you go about creating a client for your Web Service, you will not allow the client application to ask for, say, 10,000 jokes at once. But because your Web Service can certainly be used by “unauthorized” client applications—it is an Internet service, after all—you need to check for user data on the server, and you need to be able to return meaningful information to your clients.

You can then simply set up a hash table *errorCodes* with internal error codes and the corresponding nice messages for end users, and add the method, *getNiceErrorMessage()*, that translates one into the other. The instance variable *failReason* captures the error code and keeps it available as you travel back the call stack after an exception has occurred.

Creating an entry in the server application event log whenever an error does occur is probably a good idea, and that’s what the method *writeEventLogEntry()* does. Putting everything together, see Figure 8.13 for the complete code of the *jokeException* class which is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) as file *jokeException.cs*.

**Figure 8.13** Custom Error Handling Class *jokeException* (*jokeException.cs*)

```
using System;
using System.Collections;
using System.Diagnostics;

namespace jokesService
{
    /// <summary>
```

**Figure 8.13 Continued**

---

```
/// Custom error handling class
/// </summary>
/// <remarks>
/// Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001
/// </remarks>
public class jokeException : Exception {
    /// <value>
    /// fail reason error code
    /// </value>
    public string failReason;
    private static Hashtable errorCodes = new Hashtable();
    private static bool isInit = false;

    /// <summary>
    /// Public class constructor.
    /// </summary>
    /// <param name='failReason'
    /// type='string'
    /// desc='fail reason error code'>
    /// </param>
    protected internal jokeException(string failReason) {
        this.failReason = failReason;
    }

    private static void initErrorCodes() {
        errorCodes.Add("S_OK",
            "Operation completed successfully!");
        errorCodes.Add("F_System",
            "An unknown system error occurred!");
        errorCodes.Add("F_ratingInvalid",
            "Joke rating must be between 1 and 5!");
        errorCodes.Add("F_jokeDoesNotExist",
            "Joke selected does not exist in the system!");
        errorCodes.Add("F_unknownAction",
            "Internal error when accessing the database!");
    }
}
```

---

Continued

**Figure 8.13 Continued**


---

```

        errorCodes.Add("F_userDoesNotExist",
            "This is not a registered user!");
        errorCodes.Add("F_userExists",
            "Somebody has already registered under this name!");
        errorCodes.Add("F_userInfoWrong",
            "You are not authorized to do this action. Change " +
            "user name or password!");
        errorCodes.Add("F_noJokes",
            "No matching jokes in the system at this moment in time!");
        errorCodes.Add("F_10JokesMax",
            "You can only retrieve up to 10 jokes at one time!");
    }

    /// <summary>
    ///     The getNiceErrorMessage method converts an error code into
    ///     a user friendly error message, returned through a SOAP fault.
    /// </summary>
    /// <param name='errorCode'
    ///     type='string'
    ///     desc='error code'>
    /// </param>
    /// <returns>a friendly user error message</returns>
    protected internal static string getNiceErrorMessage(
        string errorCode) {
        if (!isInit) {
            // initialize error look up table once and for all
            initErrorCodes();
            isInit = true;
        }
        string temp = errorCodes[errorCode].ToString();
        if(temp.Length < 1) {
            // generic error, if error code unknown...
            return errorCodes["F_System"].ToString();
        } else {
            return temp;
        }
    }

```

---

**Continued**

**Figure 8.13 Continued**

---

```
    }
}

/// <summary>
///     The writeEventLogEntry method writes an error log entry
///     into the Application event log
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='failReason'
///     type='string'
///     desc='fail reason error code'>
/// </param>
/// <returns>nothing</returns>
protected internal static void writeEventLogEntry(
    string userName, string failReason) {
    //Create the source, if it does not already exist.
    if(!EventLog.SourceExists("jokeService")) {
        EventLog.CreateEventSource("jokeService", "Application");
    }
    //Create an EventLog instance and assign its source.
    EventLog eventLog = new EventLog();
    eventLog.Source = "jokeService";

    //Write an informational entry to the event log.
    eventLog.WriteEntry(userName + ": " + failReason);
}
}
}
```

---

## Developing the Database Access Component

The next task is to write a component that will take care of all back-end data access and offer a single gateway to the database. Externalizing the database

connection string is a good programming practice, and the .NET Framework offers a good place to put it: The `web.config` file. Just add the `appSettings` element into the `web.config` file as shown in Figure 8.14.

**Figure 8.14** Putting the Database DSN into `web.config`

---

```
<configuration>
  <appSettings>
    <add key="dsn" value="server=(local)\NetSDK;
      database=Jokes;User ID=SA;Password=" />
  </appSettings>
  <system.web>
    ...standard settings...
  </system.web>
</configuration>
```

---

The database access class (`databaseAccess.cs`) is a simple class that returns a (closed) SQL connection object to the database. Unfortunately, class constructors are not allowed to return objects, so you can add a single method to do just that, called `getConnection()`. See Figure 8.15 for the complete code for the `databaseAccess` class. The code for the `databaseAccess` class is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.15** Database Access Class `databaseAccess` (`databaseAccess.cs`)

---

```
using System;
using System.Data.SqlClient;

namespace jokesService
{
  /// <summary>
  ///   The databaseAccess sets up the connection to the
  ///   data repository.
  /// </summary>
  /// <remarks>
  ///   Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001
  /// </remarks>
  public class databaseAccess {
```

---

Continued

**Figure 8.15 Continued**

---

```
private SqlConnection sqlConnection;

/// <summary>
///   Public class constructor.
/// </summary>
protected internal databaseAccess() {
    sqlConnection = new SqlConnection(
        ConfigurationSettings.AppSettings["dsn"]);
}

/// <summary>
///   The getConnection method sets up the database connection
/// </summary>
/// <returns>the (closed) SQL connection object</returns>
protected internal SqlConnection getConnection() {
    return sqlConnection;
}
}
```

---

## Developing the User Administration Service

Now that you have taken care of error handling and database access, you will want to develop the core classes for managing users and jokes. Let's first look at how you will want to manage users: You need to be able to add new users, change existing user information, check if a user exists in the system, and be able to promote an existing user to become a moderator.

### Adding New Users

Going through the steps needed to add a new user to the system, you can start by writing the method *addUser()* in *userAdminImplement*, the class that implements user management functionality. The method takes a user name and a password as an argument, sets up the necessary infrastructure to call the SQL stored procedure *sp\_manageUser()*, gets a connection object from an instance of the class *databaseAccess*, opens the connection, and calls the stored procedure. If everything



goes well, the stored procedure will return a status code `S_OK`, and control will go back to the calling Web Service. If an exception occurred, you can create a new custom exception object of type `jokeException`, remember the error code, and throw the exception back to the caller.

The `createSqlManageUser()` method is the method that sets up our call to the stored procedure `sp_manageUser`. It takes a user name, a password, and a flag denoting if the user is a moderator as arguments. Note that all arguments are of type `string`, even the Boolean flag. The reason for this is that some arguments are, in fact, optional. For instance, when deleting a user, all you need to know is the user's username. You could certainly overload this method to do this, but in the end not a lot would change. Also, as this is an internal method of a class (and is therefore marked as protected internal) implementing functionality exposed by another public class, type consistency is not really an issue. So you can adopt the convention that all arguments to the methods that set up your SQL calls take string arguments, and that an empty string passed will mean that a SQL null value should be passed to the corresponding stored procedure. Note, though, that you can't just pass the keyword `null` to SQL; instead, we have to use `System.DBNull.value`.

You can use the MS SQL Managed Provider created specially for high performance access to MS SQL server database, which is found in the `System.Data.SqlClient` namespace (which you declare in the declaration section of your class).

Figure 8.16 shows the `createSqlManageUser()` method call that sets up the SQL command object for the stored procedure `sp_manageUser`, which deals with adding, updating, and deleting users and managers. The code in Figure 8.16 is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.16** `createSqlManageUser` Method (userAdminImplement.cs)

```

/// <summary>
///     The createSqlManageUser method sets up the SQL command object
///     for the stored procedure sp_manageUser, which deals with
///     adding, updating, and deleting users and managers
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='password'
///     type='string'

```

Continued

**Figure 8.16 Continued**

---

```
/// desc='password of registered user (zero length if N/A)'>
/// </param>
/// <param name='isModerator'
/// type='string'
/// desc='true/false if this user is a moderator'>
/// </param>
/// <param name='action'
/// type='string'
/// desc='the action the SQL stored procedure should take
/// (see the stored procedure definition for allowed action
/// keywords)'>
/// </param>
/// <param name='sqlCommand'
/// type='SqlCommand'
/// desc='a reference to a SQL command object'>
/// </param>
/// <returns>the prepared SQL command object</returns>
protected internal void createSqlManageUser(
    string userName, string password,
    string isModerator, string action, SqlCommand sqlCommand) {

    sqlCommand.CommandType = CommandType.StoredProcedure;
    sqlCommand.CommandText = "sp_manageUser" ;

    SqlParameter argUserName =
        new SqlParameter("@@userName", SqlDbType.NVarChar, 20);
    argUserName.Value = userName;
    sqlCommand.Parameters.Add(argUserName);

    SqlParameter argPassword =
        new SqlParameter("@@password", SqlDbType.NVarChar, 20);
    if(password.Length > 0) {
        argPassword.Value = password;
    } else {
        argPassword.Value = DBNull.Value;
    }
}
```

---

Continued

**Figure 8.16 Continued**


---

```

    }
    sqlCommand.Parameters.Add(argPassword);

    SqlParameter argIsModerator =
        new SqlParameter("@isModerator", SqlDbType.Bit);
    argIsModerator.Value = bool.Parse(isModerator);
    sqlCommand.Parameters.Add(argIsModerator);

    SqlParameter argAction =
        new SqlParameter("@action", SqlDbType.NVarChar, 20);
    argAction.Value = action;
    sqlCommand.Parameters.Add(argAction);

    SqlParameter argReturn =
        new SqlParameter("@return", SqlDbType.NVarChar, 20,
            ParameterDirection.Output, true, 0, 0, "",
            DataRowVersion.Current, "");
    sqlCommand.Parameters.Add(argReturn);
}

```

---

After the SQL side of adding a new user has been taken care of in method *createSqlManageUser()*, the implementation of the *addUser()* method is now straightforward, as shown in Figure 8.17. The code for *userAdminImplement.cs* is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.17 addUser Method (userAdminImplement.cs)**


---

```

/// <summary>
///     The addUser method adds a new user to the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of new user'>
/// </param>
/// <param name='password'
///     type='string'

```

---

**Continued**

## Figure 8.17 Continued

---

```
/// desc='password of new user'>
/// </param>
/// <returns>true</returns>
protected internal bool addUser(string userName, string password) {
    try {
        string retCode;
        SqlCommand sqlCommand = new SqlCommand();
        createSqlManageUser(
            userName, password, "false", "add", sqlCommand);

        databaseAccess myDatabase = new databaseAccess();
        sqlCommand.Connection = myDatabase.getConnection();
        sqlCommand.Connection.Open();

        sqlCommand.ExecuteNonQuery();
        sqlCommand.Connection.Close();
        retCode = sqlCommand.Parameters["@@return"].Value.ToString();

        // catch problems within the stored procedure
        if (retCode == "S_OK") {
            return true;
        } else {
            throw new jokeException(retCode);
        }
    }
    // catch problems with the database
} catch (Exception e) {
    throw e;
}
}
```

---

Note that the code first inspects the return code set during execution of the stored procedure. If things are not okay, say because the user has already registered previously, you can remember the error code, and throw a custom exception of type *jokeException*. If an exception occurred over which you have no control, say because the database is not accessible, you can't do much more than throw an ordinary exception of type *System.Exception*.

## Checking Existing User Information

The next method you will want to add is *checkUser()*, which matches a set of given credentials, consisting of a user name, a password, and a flag indicating whether this is a moderator, against the information in the database. You should first set up the *createSqlCheckUser* method, which wraps the call to the stored procedure *sp\_checkUser()*, shown in Figure 8.18 and also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). as *userAdminImplement.cs*.

**Figure 8.18** *createSqlCheckUser* Method (*userAdminImplement.cs*)

```

/// <summary>
///     The createSqlCheckUser method sets up the SQL command object
///     for the stored procedure sp_checkUser, which verifies passed
///     user information with user information in the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user (zero length if N/A)'\>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of registered user (zero length if N/A)'\>
/// </param>
/// <param name='isModerator'
///     type='string'
///     desc='true/false if this user is a moderator
///     (zero length if N/A)'\>
/// </param>
/// <param name='sqlCommand'
///     type='SqlCommand'
///     desc='a reference to a SQL command object'\>
/// </param>
/// <returns>the prepared SQL command object</returns>
protected internal void createSqlCheckUser(
    string userName, string password,
    string isModerator, SqlCommand sqlCommand) {

    sqlCommand.CommandType = CommandType.StoredProcedure;

```

## Figure 8.18 Continued

---

```
sqlCommand.CommandText = "sp_checkUser" ;

SqlParameter argUserName =
    new SqlParameter("@@userName", SqlDbType.NVarChar, 20);
if(userName.Length > 0) {
    argUserName.Value = userName;
} else {
    argUserName.Value = DBNull.Value;
}
sqlCommand.Parameters.Add(argUserName);

SqlParameter argPassword =
    new SqlParameter("@@password",SqlDbType.NVarChar, 20);
if(password.Length > 0) {
    argPassword.Value = password;
} else {
    argPassword.Value = DBNull.Value;
}
sqlCommand.Parameters.Add(argPassword);

SqlParameter argIsModerator =
    new SqlParameter("@@isModerator",SqlDbType.Bit);
if(isModerator.Length > 0) {
    argIsModerator.Value = bool.Parse(isModerator);
} else {
    argIsModerator.Value = DBNull.Value;
}
sqlCommand.Parameters.Add(argIsModerator);

SqlParameter argReturn =
    new SqlParameter("@@return",SqlDbType.NVarChar, 20,
        ParameterDirection.Output, true, 0, 0, "",
        DataRowVersion.Current, "");
sqlCommand.Parameters.Add(argReturn);
}
```

---

Next you need to implement the actual method, *checkUser()*, that verifies a user's credentials (Figure 8.19). This code is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.19** *createSqlCheckUser* Method (userAdminImplement.cs)

```

/// <summary>
///     The checkUser method checks if a user or moderator is
///     already defined in the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of user or moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of user or moderator'>
/// </param>
/// <param name='isModerator'
///     type='bool'
///     desc='check for moderator status (if false,
///     we do not check)'>
/// </param>
/// <returns>nothing</returns>
protected internal bool checkUser(
    string userName, string password, bool isModerator) {
    string retCode;

    try {
        SqlCommand sqlCommand = new SqlCommand();

        if(isModerator) {
            // check if user is a moderator...
            createSqlCheckUser(userName, password, "true", sqlCommand);
        } else {
            // ... or a registered user
            createSqlCheckUser(userName, password, "", sqlCommand);
        }
    }

```

Continued

**Figure 8.19** Continued

```
databaseAccess myDatabase = new databaseAccess();
sqlCommand.Connection = myDatabase.getConnection();
sqlCommand.Connection.Open();

sqlCommand.ExecuteNonQuery();
retCode = sqlCommand.Parameters["@return"].Value.ToString();

// catch problems within the stored procedure
if (retCode == "S_OK") {
    return true;
} else {
    throw new jokeException(retCode);
}
// catch problems with the database
} catch (Exception e) {
    throw e;
}
}
```

## Adding Moderators

Lastly, you need to think about adding moderators to the system. You want to let only moderators add moderators, and those new moderators already need to be registered with the system as regular users.

So the *addModerator* method has to have three arguments: The use name and password of the moderator adding a new moderator, and the user name of the user who should become moderator. You need to first check that the credentials given are indeed the ones of an existing moderator, for which we can use the *checkUser()* method, and then you need to modify the entry in the user table for the new moderator, which consists in simply changing the *isModerator* flag to True.

A lot of things can go wrong even with this simple call: The moderator requesting the change may not be a moderator, or the user slated to become a moderator may not exist in the database. Thankfully, you no longer need to worry about these eventualities, because your error handling system will handle



those exceptions automatically. Figure 8.20 shows the code for *addManager()*, which is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) as *userAdminImplement.cs*.

**Figure 8.20** *addModerator* Method (*userAdminImplement.cs*)

```

/// <summary>
///     The addModerator method sets a previously added user to become
///     a moderator
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator making the call'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator making the call'>
/// </param>
/// <param name='newModerator'
///     type='string'
///     desc='user name of registered user who will become
///     a moderator'>
/// </param>
/// <returns>true</returns>
protected internal bool addModerator(
    string userName, string password, string newModerator) {
    string retCode;

    try {
        // check if user is a moderator
        SqlCommand sqlCommand = new SqlCommand();
        createSqlCheckUser(userName, password, "true", sqlCommand);

        databaseAccess myDatabase = new databaseAccess();
        sqlCommand.Connection = myDatabase.getConnection();
        sqlCommand.Connection.Open();

```

Continued

**Figure 8.20 Continued**

---

```
sqlCommand.ExecuteNonQuery();
retCode = sqlCommand.Parameters["@return"].Value.ToString();

// catch problems within the stored procedure
if (retCode != "S_OK") {
    sqlCommand.Connection.Close();
    throw new jokeException(retCode);
}

// make newModerator a moderator
sqlCommand.Parameters.Clear();
createSqlManageUser(
    newModerator, "", "true", "modify", sqlCommand);

sqlCommand.ExecuteNonQuery();
sqlCommand.Connection.Close();

retCode = sqlCommand.Parameters["@return"].Value.ToString();

// catch problems within the stored procedure
if (retCode == "S_OK") {
    return true;
} else {
    throw new jokeException(retCode);
}
// catch problems with the database
} catch (Exception e) {
    throw e;
}
}
```

---

## Creating the Public Web Methods—Users

The implementation of the user administration service is now complete, and all that remains to do is to expose this service to the world. To do this, simply add new

(public!) Web methods to the *userAdmin* class, which is found in file *userAdmin.asmx.cs* on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). First, you need to add some custom initialization code to the *userAdmin* Web Service class, as show in Figure 8.21.

**Figure 8.21** Code to Set Up the *userAdmin* Web Service (*userAdmin.asmx.cs*)

```
using System;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Web.Services.Protocols;
using System.Xml;

namespace jokesService {
    /// <summary>
    ///     The userAdmin class provides methods to manage users and
    ///     moderators in the database.
    /// </summary>
    /// <remarks>
    ///     Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001
    /// </remarks>
    [WebServiceAttribute(Description="The userAdmin web service " +
        "provides methods to manage users and moderators in the database",
        Namespace="urn:schemas-syngress-com-soap")]
    public class userAdmin : System.Web.Services.WebService {
        // SOAP error handling return document structure
        /// <value>error document thrown by SOAP exception</value>
        public XmlDocument soapErrorDoc;
        /// <value>text node with user friendly error message</value>
        public XmlNode xmlFailReasonNode;

        /// <summary>
        ///     Public class constructor.
        /// </summary>

```

Continued

**Figure 8.21 Continued**


---

```

public userAdmin() {
    InitializeComponent();
    // initialize SOAP error handling return document
    soapErrorDoc = new System.Xml.XmlDocument();
    xmlFailReasonNode =
        soapErrorDoc.CreateNode(XmlNodeType.Element, "failReason", "");
    }
}
}

```

---

The code for the *addUser()* method which adds a new user to the database is shown in Figure 8.22 and is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.22 addUser Web Method (userAdmin.asmx.cs)**


---

```

01: /// <summary>
02: ///     The addUser method adds a new user to the database
03: /// </summary>
04: /// <param name='userName'
05: ///     type='string'
06: ///     desc='name of new user'>
07: /// </param>
08: /// <param name='password'
09: ///     type='string'
10: ///     desc='password of new user'>
11: /// </param>
12: /// <returns>nothing</returns>
13: [SoapDocumentMethodAttribute(Action="addUser",
14:     RequestNamespace="urn:schemas-syngress-com-soap:userAdmin",
15:     RequestElementName="addUser",
16:     ResponseNamespace="urn:schemas-syngress-com-soap:userAdmin",
17:     ResponseElementName="addUserResponse")]
18: [WebMethod(Description="The addUser method adds a new user to " +
19:     "the database")]
20: public void addUser(string userName, string password) {

```

---

Continued

**Figure 8.22 Continued**


---

```

21:  userAdminImplement userAdminObj = new userAdminImplement();
22:  try {
23:      userAdminObj.addUser(userName, password);
24:      // catch jokeExceptions
25:  } catch (jokeException e) {
26:      throwFault("Fault occurred", e.failReason, userName);
27:  }
28:  // then, catch general System Exceptions
29:  catch (Exception e) {
30      throwFault(e.Message, "F_System", userName);
31:  }
32: }

```

---

Note how simple things suddenly become once you have set the stage correctly: You need just two lines to add a new user to the system. There are two things to focus on in Figure 8.22:

- First, some decorations to the Web method (which Microsoft calls metadata). They specify the namespaces (lines 14 and 16) and element names (lines 15 and 17) used by the SOAP protocol, as described in previous chapters.
- Second, if an exception occurs, you call a custom error handler that returns extended error information as part of a SOAP fault (lines 25 and 26).

## Error Handling for the Public Web Methods

If you look at the code that adds users to the system, you will see that *throwFault* (Figure 8.22, lines 26 and 30) is the name of the method that actually throws a SOAP fault and ends execution of the Web Service method. But it does a whole lot more:

- The (internal) error code is replaced by a user friendly error message.
- A log entry is written to the *Application* event log.
- The standard SOAP fault XML document is appended with a custom element, called *failReason*, where client applications can find the error message to display to users.

The details of the *throwFault* method are shown in Figure 8.23 and is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).



**Figure 8.23** *throwFault* Method (userAdmin.asmx.cs)

```
/// <summary>
///     The throwFault method throws a SOAP fault end ends
///     execution of the Web Service method
/// </summary>
/// <param name='message'
///     type='string'
///     desc='start of text node of faultstring element in
///     SOAP fault message'>
/// </param>
/// <param name='failReason'
///     type='string'
///     desc='text node for custom failReason element in SOAP
///     fault message'>
/// </param>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <returns>nothing</returns>
private void throwFault(string message, string failReason, string
userName) {
    xmlFailReasonNode.AppendChild(soapErrorDoc.CreateTextNode(
        jokeException.getNiceErrorMessage(failReason)));
    jokeException.writeEventLogEntry(userName, failReason);
    throw new SoapException(message, SoapException.ServerFaultCode,
        Context.Request.Url.AbsoluteUri, null,
        new System.Xml.XmlNode[] {xmlFailReasonNode});
}
```

For instance, if we try to add a user who is already registered, a SOAP fault as pictured in Figure 8.24 will be returned.

**Figure 8.24** A SOAP Fault Extended by a Custom XML Element

```

VBScript
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <soap:Fault>
      <faultcode>soap:Server</faultcode>
      <faultstring>System.Web.Services.Protocols.SoapException: Fault occurred
at jokes.userAdmin.throwFault(String message, String failReason, String userName) in
c:\inetpub\wwwroot\jokes1\useradmin.asmx.cs:line 177
at jokes.userAdmin.addUser(String userName, String password) in c:\inetpub\wwwroot\jokes1\useradmin.asmx.cs:line
75</faultstring>
      <faultactor>http://localhost/Jokes1/userAdmin.asmx</faultactor>
      <detail/>
      <failReason>Somebody has already registered under this name</failReason>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
  
```

## Creating The Public Web Methods—Administrators

The two other public Web methods of the *userAdmin* Web Service are very similar in their structure to the *addUser* Web method; they are the Web method *addModerator()* which adds a new moderator to the database, and the Web method *checkUser()* which checks if a user or moderator is already defined in the database. Those two methods are presented in Figures 8.25 and 8.26, respectively, and the corresponding code is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.25** *addModerator* Web method (*userAdmin.asmx.cs*)

```

/// <summary>
///   The addModerator method adds a new moderator to the database
/// </summary>
/// <param name='userName'
///   type='string'
///   desc='name of moderator'>
/// </param>
/// <param name='password'
///   type='string'
///   desc='password of moderator'>
/// </param>
/// <param name='newModerator'
///   type='string'
///   desc='user name of user who will become a moderator'>
  
```

Continued

**Figure 8.25 Continued**


---

```

/// </param>
/// <returns>nothing</returns>
[SoapDocumentMethodAttribute(Action="addModerator",
    RequestNamespace="urn:schemas-syngress-com-soap:userAdmin",
    RequestElementName="addModerator",
    ResponseNamespace="urn:schemas-syngress-com-soap:userAdmin",
    ResponseElementName="addModeratorResponse")]
[WebMethod(Description="The addModerator method adds a new " +
    "moderator to the database")]
public void addModerator(
    string userName, string password, string newModerator) {
    userAdminImplement userAdminObj = new userAdminImplement();
    try {
        userAdminObj.addModerator(userName, password, newModerator);
        // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
    }
    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
    }
}

```

---

**Figure 8.26** *checkUser* Web Method (userAdmin.asmx.cs)

---

```

/// <summary>
///     The checkUser method checks if a user or moderator is
///     already defined in the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of user or moderator'>
/// </param>
/// <param name='password'

```

---

Continued



**Figure 8.26 Continued**


---

```

///  type='string'
///  desc='password of user or moderator'>
/// </param>
/// <param name='isModerator'
///  type='bool'
///  desc='check for moderator status (if false, we do
///  not check)'>
/// </param>
/// <returns>nothing</returns>
[SoapDocumentMethodAttribute(Action="checkUser",
    RequestNamespace="urn:schemas-syngress-com-soap:userAdmin",
    RequestElementName="checkUser",
    ResponseNamespace="urn:schemas-syngress-com-soap:userAdmin",
    ResponseElementName="checkUserResponse")]
[WebMethod(Description="The checkUser method checks if a user " +
    "or moderator is already defined in the database")]
public void checkUser(
    string userName, string password, bool isModerator) {
    userAdminImplement userAdminObj = new userAdminImplement();
    try {
        userAdminObj.checkUser(userName, password, isModerator);
        // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
    }
    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
    }
}
}

```

---

Et voilà! You're done with your first "real" Web Service: the *userAdmin* Web Service: The user administration module for the Jokes application.

## Testing the Public Web Methods

We can immediately check if things work properly by calling the Web Service from a Visual Basic script. The VBS script shown in Figure 8.27 will add a new user.

**Figure 8.27** A Simple Visual Basic Script to Test Adding a New User to the Database

---

```
myWebService = "http://localhost/Jokes1/userAdmin.asmx"
myMethod = "addUser"

' ** create the SOAP envelope with the request
myData = ""
myData = myData & "<?xml version=""1.0"" encoding=""utf-8""?>"
myData = myData & "<soap:Envelope xmlns:soap=""http://schemas."
myData = myData & "xmlsoap.org/soap/envelope/"">"
myData = myData & "  <soap:Body>"
myData = myData & "    <addUser xmlns=""urn:schemas-syngress-"
myData = myData & "com-soap:userAdmin"">"
myData = myData & "      <userName>newUser</userName>"
myData = myData & "      <password>newPassword</password>"
myData = myData & "    </addUser>"
myData = myData & "  </soap:Body>"
myData = myData & "</soap:Envelope>"
msgbox(myData)

set requestHTTP = CreateObject("Microsoft.XMLHTTP")

msgbox("xmlhttp object created")

requestHTTP.open "POST", myWebService, false
requestHTTP.setRequestHeader "Content-Type", "text/xml"
requestHTTP.setRequestHeader "SOAPAction", myMethod
requestHTTP.Send myData

msgbox("request sent")

set responseDocument = requestHTTP.responseXML
```

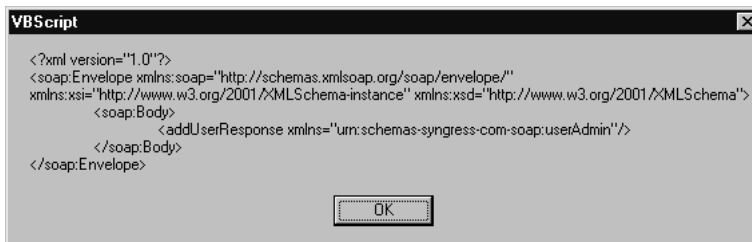
---

Continued

**Figure 8.27 Continued**

```
msgbox(requestHTTP.status)
msgbox(responseDocument.xml)
```

If things go right, a new user should be added to the database and a message box depicting a SOAP return envelope should appear as shown in Figure 8.28.

**Figure 8.28 A Successful Call to Add a New Registered User**

## Developing the Jokes Service

The second Web Service to develop is the jokes Web Service. The main feature of this Web Service is that it lets registered users retrieve jokes. Additionally, it contains methods to administer jokes, such as adding and removing jokes, approving jokes submitted by users to be visible to other users, and giving users a way to rate existing jokes. In many respects, things are set up in parallel from what we have already seen in the *userAdmin* Web Service, which is the Web Service to manage user information.

## Best Practices for Returning Highly Structured Data

Compared with the *userAdmin* Web Service you just developed the Jokes Web Service has one key additional difficulty: How to return joke data. Our requirements are as follows:

- Return anywhere from 1 to 10 jokes.
- Along with each joke, return its average user rating, and the joke identifier (if for example, a user wants to rate that joke).

From the stored procedure *sp\_getJokes* you can get a SQL record set. One possibility, then, is to simply return our jokes as “record sets” (the correct term here is objects of type *System.Data.DataSet*). This magic works because the .NET

SOAP serializer, which is the piece of code that converts the data into XML format to be sent back inside a SOAP return envelope, can indeed serialize that kind of data out of the box. However, returning serialized *DataSets* may often not be a good idea because in practice it pretty much forces your clients to run on a Microsoft .NET platform, counter to the idea of Web Services to be an open standard.

What alternatives do you have? Again, our advice is to use a simple structure adapted to the problem at hand. If you want your clients to validate the XML against a DTD or an XML Schema, you can always pass that information as a URL (maybe to another Web Service!), but don't pass that information by default with every call to the client. In this case, you can pass a structure that looks essentially like everything above starting from the *NewDataSet* element; that is, you want an XML element delineating rows of data, and additional XML elements delineating the fields of data within each row of data.

This is done very simply by creating a custom C# class, the *xmlJokesReturn* class, which is designed to hold a single row of data as shown in Figure 8.29, and is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)). Of course, if you prefer, the same could be achieved by using a structure.



**Figure 8.29** The *xmlJokesReturn* Class that Holds the Jokes (*xmlJokesReturn.cs*)

```
using System;

namespace jokesService
{
    /// <summary>
    ///     The xmlJokesReturn class is the return type of all public
    ///     methods returning joke data.
    /// </summary>
    /// <remarks>
    ///     Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001
    /// </remarks>
    public class xmlJokesReturn {
        /// <value>ID of joke returned</value>
        public string jokeID;
        /// <value>the actual joke</value>
        public string joke;
        /// <value>average rating of the joke (can be empty)</value>
```

Continued

**Figure 8.29 Continued**


---

```

    public string rating;

    /// <summary>
    ///     Public class constructor.
    /// </summary>
    public xmlJokesReturn() {
    }
}

```

---

Because you may return more than one row of data, of course, you can set up the *getJokes* Web method to return an array of objects of type *xmlJokesReturn*. The SOAP serializer does the rest automatically.

In Figure 8.30, also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)), you see the definition of the *getJokes* Web method (note that we haven't talked about the corresponding implementation method yet).

**Figure 8.30 *getJokes* Web Method (jokes.asmx.cs)**


---

```

[WebMethod]
public xmlJokesReturn[] getJokes(
    string userName, string password, int howMany) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        xmlJokesReturn[] myJokes =
            jokesObj.getJokes(userName, password, howMany);
        return myJokes;
    }
    // error handler omitted
}

```

---

The SOAP object serializer does what it is supposed to do, that is it returns a serialized array of *xmlJokesReturn* objects, and you retrieve a SOAP envelope on the client that may look as in Figure 8.31, containing two jokes.

**Figure 8.31** SOAP Response Envelope Containing Two Jokes as Serialized *xmlJokesReturn* Objects

---

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
    <getJokesResponse xmlns="urn:schemas-syngress-com-soap:jokes">
      <jokeData>
        <jokeID>1</jokeID>
        <joke>this is the first joke</joke>
        <rating>3.5</rating>
      </jokeData>
      <jokeData>
        <jokeID>2</jokeID>
        <joke>this is the second joke</joke>
        <rating />
      </jokeData>
    </getJokesResponse>
  </soap:Body>
</soap:Envelope>
```

---

## Setting Up Internal Methods to Wrap the Stored Procedure Calls

Similar to the way you proceeded when developing the *userAdmin* Web Service, you want to create internal methods to wrap calls to the stored procedures that interface with the jokes in the database. You should have three stored procedures that deal with jokes:

- *sp\_manageJoke*
- *sp\_manageRating*
- *sp\_returnJokes*

The corresponding wrapping methods, part of file *JokesImplement.cs*, are shown in detail in Figure 8.32 (*createSqlManageJoke*), Figure 8.33 (*createSqlManageRating*),

and Figure 8.34 (*createSqlReturnJokes*). The code for all three Figures is also available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.32** *createSqlManageJoke* Method (*jokesImplement.cs*)

```

/// <summary>
///     The createSqlManageJoke method sets up the SQL command object
///     for the stored procedure sp_manageJoke, which deals with
///     adding, updating, and deleting jokes
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user (zero length if N/A)'>
/// </param>
/// <param name='joke'
///     type='string'
///     desc='the joke (zero length if N/A)'>
/// </param>
/// <param name='isModerated'
///     type='string'
///     desc='true/false if this is/is not a moderated joke
///     (zero length if N/A)'>
/// </param>
/// <param name='jokeID'
///     type='string'
///     desc='the joke ID for the joke (zero length if N/A)'>
/// </param>
/// <param name='action'
///     type='string'
///     desc='the action the SQL stored procedure should take
///     (see the stored procedure definition for allowed action
///     keywords)'>
/// </param>
/// <param name='sqlCommand'
///     type='SqlCommand'
///     desc='a reference to a SQL command object'>
/// </param>
/// <returns>the prepared SQL command object</returns>

```

Continued

## Figure 8.32 Continued

---

```
protected internal void createSqlManageJoke(
    string userName, string joke, string isModerated,
    string jokeID, string action, SqlCommand sqlCommand) {

    sqlCommand.CommandType = CommandType.StoredProcedure;
    sqlCommand.CommandText = "sp_manageJoke" ;

    SqlParameter argUserName =
        new SqlParameter("@@userName", SqlDbType.NVarChar, 20);
    if(userName.Length > 0) {
        argUserName.Value = userName;
    } else {
        argUserName.Value = DBNull.Value;
    }
    sqlCommand.Parameters.Add(argUserName);

    SqlParameter argJoke =
        new SqlParameter("@@joke", SqlDbType.NVarChar, 3500);
    if(joke.Length > 0) {
        argJoke.Value = joke;
    } else {
        argJoke.Value = DBNull.Value;
    }
    sqlCommand.Parameters.Add(argJoke);

    SqlParameter argIsModerated =
        new SqlParameter("@@isModerated", SqlDbType.Bit);
    if(isModerated.Length > 0) {
        argIsModerated.Value = bool.Parse(isModerated);
    } else {
        argIsModerated.Value = DBNull.Value;
    }
    sqlCommand.Parameters.Add(argIsModerated);

    SqlParameter argJokeID =
```

---

Continued



**Figure 8.32 Continued**


---

```

        new SqlParameter("@@jokeID", SqlDbType.Int);
    if(jokeID.Length > 0) {
        argJokeID.Value = Int32.Parse(jokeID);
    } else {
        argJokeID.Value = DBNull.Value;
    }
    sqlCommand.Parameters.Add(argJokeID);

    SqlParameter argAction =
        new SqlParameter("@@action", SqlDbType.NVarChar, 20);
    argAction.Value = action;
    sqlCommand.Parameters.Add(argAction);

    SqlParameter argReturn =
        new SqlParameter("@@return", SqlDbType.NVarChar, 20,
            ParameterDirection.Output, true, 0, 0, "",
            DataRowVersion.Current, "");
    sqlCommand.Parameters.Add(argReturn);
}

```

---

**Figure 8.33** *createSql/ManageRating* Method (jokesImplement.cs)

---

```

/// <summary>
///     The createSqlManageRating method sets up the SQL command
///     object for the stored procedure sp_manageRating, which
///     deals with adding and deleting user joke ratings
/// </summary>
/// <param name='jokeID'
///     type='string'
///     desc='the joke ID for the joke we would like to rate'>
/// </param>
/// <param name='rating'
///     type='string'
///     desc='the user rating for the joke (1-5)'>
/// </param>

```

---

Continued

## Figure 8.33 Continued

---

```
/// <param name='action'
///   type='string'
///   desc='the action the SQL stored procedure should take
///   (see the stored procedure definition for allowed action
///   keywords)'>
/// </param>
/// <param name='sqlCommand'
///   type='SqlCommand'
///   desc='a reference to a SQL command object'>
/// </param>
/// <returns>the prepared SQL command object</returns>
protected internal void createSqlManageRating(
    string jokeID, string rating, string action,
    SqlCommand sqlCommand) {

    sqlCommand.CommandType = CommandType.StoredProcedure;
    sqlCommand.CommandText = "sp_manageRating" ;

    SqlParameter argJokeID =
        new SqlParameter("@@jokeID", SqlDbType.Int);
    argJokeID.Value = Int32.Parse(jokeID);
    sqlCommand.Parameters.Add(argJokeID);

    SqlParameter argRating =
        new SqlParameter("@@rating", SqlDbType.TinyInt);
    argRating.Value = Int32.Parse(rating);
    sqlCommand.Parameters.Add(argRating);

    SqlParameter argAction =
        new SqlParameter("@@action", SqlDbType.NVarChar, 20);
    argAction.Value = action;
    sqlCommand.Parameters.Add(argAction);

    SqlParameter argReturn =
        new SqlParameter("@@return", SqlDbType.NVarChar, 20,
```

---

Continued

**Figure 8.33 Continued**


---

```

        ParameterDirection.Output, true, 0, 0, "",
        DataRowVersion.Current, "");
    sqlCommand.Parameters.Add(argReturn);
}

```

---

**Figure 8.34 createSqlReturnJokes Method (JokesImplement.cs)**


---

```

/// <summary>
///     The createSqlReturnJokes method sets up the SQL command object
///     for the stored procedure sp_returnJokes, which returns jokes
/// </summary>
/// <param name='howMany'
///     type='string'
///     desc='how many jokes we would like (zero length if N/A)'>
/// </param>
/// <param name='isModerated'
///     type='string'
///     desc='true/false if we are interested in (not) moderated
///     jokes (zero length if N/A)'>
/// </param>
/// <param name='returnRandom'
///     type='string'
///     desc='true/false if we are interested getting random jokes
///     (actually, only the starting position is random, from there
///     on we retrieve jokes in sequential order for practical
///     reasons)'>
/// </param>
/// <param name='sqlCommand'
///     type='SqlCommand'
///     desc='a reference to a SQL command object'>
/// </param>
/// <returns>the prepared SQL command object</returns>
protected internal void createSqlReturnJokes(
    string howMany, string isModerated, string returnRandom,
    SqlCommand sqlCommand) {

```

**Figure 8.34 Continued**

---

```
sqlCommand.CommandType = CommandType.StoredProcedure;
sqlCommand.CommandText = "sp_returnJokes" ;

SqlParameter argHowMany =
    new SqlParameter("@@howMany", SqlDbType.Int);
if(howMany.Length > 0) {
    argHowMany.Value = Int32.Parse(howMany);
} else {
    argHowMany.Value = DBNull.Value;
}
sqlCommand.Parameters.Add(argHowMany);

SqlParameter argIsModerated =
    new SqlParameter("@@isModerated", SqlDbType.Bit);
if(isModerated.Length > 0) {
    argIsModerated.Value = bool.Parse(isModerated);
} else {
    argIsModerated.Value = DBNull.Value;
}
sqlCommand.Parameters.Add(argIsModerated);

SqlParameter argReturnRandom =
    new SqlParameter("@@returnRandom", SqlDbType.Bit);
argReturnRandom.Value = bool.Parse(returnRandom);
sqlCommand.Parameters.Add(argReturnRandom);
}
```

---

## Setting Up Internal Methods to Manage Jokes and Ratings

Now that you can call the stored procedures that deal with jokes in the database, you want to implement the business logic that deals with jokes. There are four methods that either add or delete jokes and ratings.

- ***addJoke()*** Checks that a user is registered, and then adds the passed joke as an unmoderated joke to the system.
- ***addRating()*** Checks that a user is registered, and then adds the passed rating to the joke having the passed joke identifier to the system.
- ***addModerated()*** Checks that a user is a moderator, and then changes the *isModerated* flag of the joke having the passed joke identifier to the system.
- ***deleteUnmoderated()*** Checks that a user is a moderator, and then removes the joke having the passed joke identifier, along with all its user ratings, from the system.

Figure 8.35 shows the business logic for the *addJoke* method, while Figures 8.36, 8.37 and 8.37 deal with the *addRating*, *addModerated*, and *deleteUnmoderated* methods respectively. Here's the code for those methods, (still part of file *JokesImplement.cs*) which is also available on on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.35** *addJoke* Method (*JokesImplement.cs*)

```

/// <summary>
///     The addJoke method lets registered users add a joke
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of registered user'>
/// </param>
/// <param name='joke'
///     type='string'
///     desc='the joke we are adding'>
/// </param>
/// <returns>true</returns>
protected internal bool addJoke(
    string userName, string password, string joke) {
    string retCode;

```

**Figure 8.35 Continued**

---

```
try {
    // check if user is registered
    userAdminImplement myUser = new userAdminImplement();
    SqlCommand sqlCommand = new SqlCommand();

    myUser.createSqlCheckUser(userName, password, "", sqlCommand);

    databaseAccess myDatabase = new databaseAccess();
    sqlCommand.Connection = myDatabase.getConnection();
    sqlCommand.Connection.Open();

    sqlCommand.ExecuteNonQuery();
    retCode = sqlCommand.Parameters["@return"].Value.ToString();

    // exit, if user not registered
    if (retCode != "S_OK") {
        sqlCommand.Connection.Close();
        throw new jokeException(retCode);
    }

    // add the joke (unmoderated, at this point)
    sqlCommand.Parameters.Clear();
    createSqlManageJoke(
        userName, joke, "false", "", "add", sqlCommand);

    sqlCommand.ExecuteNonQuery();
    sqlCommand.Connection.Close();

    retCode = sqlCommand.Parameters["@return"].Value.ToString();

    // catch problems within the stored procedure
    if (retCode == "S_OK") {
        return true;
    } else {
```

---

Continued

**Figure 8.35 Continued**


---

```

        throw new jokeException(retCode);
    }
    // catch problems with the database
} catch (Exception e) {
    throw e;
}
}

```

---

**Figure 8.36 addRating Method (JokesImplement.cs)**


---

```

/// <summary>
///     The addRating method lets registered users rate a joke
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of registered user'>
/// </param>
/// <param name='rating'
///     type='int'
///     desc='the rating of the joke to rate (1-5)'>
/// </param>
/// <param name='jokeID'
///     type='int'
///     desc='the ID of the joke to rate'>
/// </param>
/// <returns>true</returns>
protected internal bool addRating(
    string userName, string password, int rating, int jokeID) {
    string retCode;

    try {
        // check if user is registered

```

---

**Figure 8.36 Continued**

```
userAdminImplement myUser = new userAdminImplement();
SqlCommand sqlCommand = new SqlCommand();

myUser.createSqlCheckUser(userName, password, "", sqlCommand);

databaseAccess myDatabase = new databaseAccess();
sqlCommand.Connection = myDatabase.getConnection();
sqlCommand.Connection.Open();

sqlCommand.ExecuteNonQuery();
retCode = sqlCommand.Parameters["@return"].Value.ToString();

// exit, if user not registered
if (retCode != "S_OK") {
    sqlCommand.Connection.Close();
    throw new jokeException(retCode);
}

// add the joke rating
sqlCommand.Parameters.Clear();
createSqlManageRating(
    jokeID.ToString(), rating.ToString(), "add", sqlCommand);

sqlCommand.ExecuteNonQuery();
sqlCommand.Connection.Close();

retCode = sqlCommand.Parameters["@return"].Value.ToString();

// catch problems within the stored procedure
if (retCode == "S_OK") {
    return true;
} else {
    throw new jokeException(retCode);
}

// catch problems with the database
} catch (Exception e) {
```

**Continued****www.syngress.com**



**Figure 8.36 Continued**


---

```

        throw e;
    }
}

```

---

**Figure 8.37 addModerated Method (JokesImplement.cs)**


---

```

/// <summary>
///     The addModerated method sets a previously submitted joke
///     to become a moderated joke
///     (for moderators only)
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator'>
/// </param>
/// <param name='jokeID'
///     type='int'
///     desc='joke ID of joke'>
/// </param>
/// <returns>an XML representation (xmlJokesReturn)
/// of a single joke</returns>
protected internal bool addModerated(
    string userName, string password, int jokeID) {
    string retCode;

    try {
        // check if user is a moderator
        userAdminImplement myUser = new userAdminImplement();
        SqlCommand sqlCommand = new SqlCommand();

        myUser.createSqlCheckUser(
            userName, password, "true", sqlCommand);

```

**Figure 8.37** Continued

```
databaseAccess myDatabase = new databaseAccess();
sqlCommand.Connection = myDatabase.getConnection();
sqlCommand.Connection.Open();

sqlCommand.ExecuteNonQuery();
retCode = sqlCommand.Parameters["@return"].Value.ToString();

// exit, if user not a moderator
if (retCode != "S_OK") {
    sqlCommand.Connection.Close();
    throw new jokeException(retCode);
}

// make the joke a moderated one
sqlCommand.Parameters.Clear();
createSqlManageJoke(userName, "", "true", jokeID.ToString(),
    "modify", sqlCommand);

sqlCommand.ExecuteNonQuery();
sqlCommand.Connection.Close();

retCode = sqlCommand.Parameters["@return"].Value.ToString();

// catch problems within the stored procedure
if (retCode == "S_OK") {
    return true;
} else {
    throw new jokeException(retCode);
}
// catch problems with the database
} catch (Exception e) {
    throw e;
}
}
```

**Figure 8.38** *deleteUnmoderated* Method (JokesImplement.cs)

```

/// <summary>
///     The deleteUnmoderated method deletes a previously
///     submitted joke (unmoderated) joke
///     (for moderators only)
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator'>
/// </param>
/// <param name='jokeID'
///     type='int'
///     desc='joke ID of joke'>
/// </param>
/// <returns>true</returns>
protected internal bool deleteUnmoderated(
    string userName, string password, int jokeID) {
    string retCode;

    try {
        // check if user is a moderator
        userAdminImplement myUser = new userAdminImplement();
        SqlCommand sqlCommand = new SqlCommand();

        myUser.createSqlCheckUser(
            userName, password, "true", sqlCommand);

        databaseAccess myDatabase = new databaseAccess();
        sqlCommand.Connection = myDatabase.getConnection();
        sqlCommand.Connection.Open();

        sqlCommand.ExecuteNonQuery();
    }
}

```

**Figure 8.38 Continued**

---

```
retCode = sqlCommand.Parameters["@return"].Value.ToString();

// exit, if user not a moderator
if (retCode != "S_OK") {
    sqlCommand.Connection.Close();
    throw new jokeException(retCode);
}

// delete the joke
sqlCommand.Parameters.Clear();
createSqlManageJoke(
    userName, "", "", jokeID.ToString(), "delete", sqlCommand);

sqlCommand.ExecuteNonQuery();
sqlCommand.Connection.Close();

retCode = sqlCommand.Parameters["@return"].Value.ToString();

// catch problems within the stored procedure
if (retCode == "S_OK") {
    return true;
} else {
    throw new jokeException(retCode);
}
// catch problems with the database
} catch (Exception e) {
    throw e;
}
}
```

---

## Setting Up Internal Methods to Return Jokes

Finally, there are two methods that return joke data.

- ***getJokes()*** Checks that a user is registered, and then returns one or more moderated jokes, depending on the argument passed.

- ***getUnmoderated()*** Checks that user is a moderator, and then returns one or more moderated jokes, depending on the argument passed.

As mentioned above, we forgo returning *DataSets*, and return instead an array of type *xmlJokesReturn*. Figure 8.39 shows the code for the *getJokes* method, while Figure 8.40 details method *getUnmoderated*. The corresponding code for both Figures is available on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.39** *getJokes* Method (JokesImplement.cs)

```

/// <summary>
///   The getJokes method returns howMany new jokes from
///   the database
/// </summary>
/// <param name='userName'
///   type='string'
///   desc='name of registered user'>
/// </param>
/// <param name='password'
///   type='string'
///   desc='password of registered user'>
/// </param>
/// <param name='howMany'
///   type='int'
///   desc='number of jokes to return (1-10)'>
/// </param>
/// <returns>an XML representation (xmlJokesReturn) of a
///   single joke</returns>
protected internal xmlJokesReturn[] getJokes(
    string userName, string password, int howMany) {
    string retCode;

    try {
        // check if user is registered
        userAdminImplement myUser = new userAdminImplement();
        SqlCommand sqlCommand = new SqlCommand();

```

Continued

**Figure 8.39 Continued**

---

```
myUser.createSqlCheckUser(userName, password, "", sqlCommand);

databaseAccess myDatabase = new databaseAccess();
sqlCommand.Connection = myDatabase.getConnection();
sqlCommand.Connection.Open();

sqlCommand.ExecuteNonQuery();

retCode = sqlCommand.Parameters["@return"].Value.ToString();

// exit, if user not registered
if (retCode != "S_OK") {
    sqlCommand.Connection.Close();
    throw new jokeException(retCode);
}

// retrieve a random joke

// maximum is 10 jokes
if((howMany < 1) || (howMany > 10)) {
    throw new jokeException("F_10JokesMax");
}

sqlCommand.Parameters.Clear();
createSqlReturnJokes(
    howMany.ToString(), "true", "true", sqlCommand);

sqlCommand.ExecuteNonQuery();

sqlCommand.Connection.Close();

SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(sqlCommand);
DataTable dataTable = new DataTable("sqlReturn");
sqlDataAdapter.Fill(dataTable);
```

---

Continued

**Figure 8.39** Continued

---

```

// convert SQL table into xmlJokesReturn class
int rowCount = dataTable.Rows.Count;
xmlJokesReturn[] myJokes = new xmlJokesReturn[rowCount];
for(int i = 0; i < rowCount; i++) {
    myJokes[i] = new xmlJokesReturn();
    myJokes[i].jokeID = dataTable.Rows[i][0].ToString();
    myJokes[i].joke    = dataTable.Rows[i][1].ToString();
    myJokes[i].rating = dataTable.Rows[i][2].ToString();
}
// catch problems within the stored procedure
if(rowCount > 0) {
    return myJokes;
} else {
    throw new jokeException("F_noJokes");
}
// catch problems with the database
} catch (Exception e) {
    throw e;
}
}

```

---

**Figure 8.40** *getUnmoderated* Method (JokesImplement.cs)

---

```

/// <summary>
///     The getUnmoderated method retrieves howMany jokes from
///     the database
///     (for moderators only)
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator'>
/// </param>

```

---

## Figure 8.40 Continued

---

```
/// <param name='howMany'
///   type='int'
///   desc='number of jokes to return'>
/// </param>
/// <returns>an XML representation (xmlJokesReturn)
/// of a single joke</returns>
protected internal xmlJokesReturn[] getUnmoderated(
    string userName, string password, int howMany) {
    string retCode;

    try {
        // check if user is a moderator
        userAdminImplement myUser = new userAdminImplement();
        SqlCommand sqlCommand = new SqlCommand();

        myUser.createSqlCheckUser(
            userName, password, "true", sqlCommand);

        databaseAccess myDatabase = new databaseAccess();
        sqlCommand.Connection = myDatabase.getConnection();
        sqlCommand.Connection.Open();

        sqlCommand.ExecuteNonQuery();

        retCode = sqlCommand.Parameters["@@return"].Value.ToString();

        // exit, if user not a moderator
        if (retCode != "S_OK") {
            sqlCommand.Connection.Close();
            throw new jokeException(retCode);
        }

        // retrieve the first <howMany> unmoderated jokes

        // maximum is 10 jokes
```

---

Continued



**Figure 8.40** Continued

---

```

    if((howMany < 1) || (howMany > 10)) {
        throw new jokeException("F_10JokesMax");
    }

    sqlCommand.Parameters.Clear();
    createSqlReturnJokes(
        howMany.ToString(), "false", "false", sqlCommand);

    sqlCommand.ExecuteNonQuery();

    sqlCommand.Connection.Close();

    SqlDataAdapter sqlDataAdapter = new SqlDataAdapter(sqlCommand);
    DataTable dataTable = new DataTable("sqlReturn");
    sqlDataAdapter.Fill(dataTable);

    // convert SQL table into xmlJokesReturn class
    int rowCount = dataTable.Rows.Count;
    xmlJokesReturn[] myJokes = new xmlJokesReturn[rowCount];
    for(int i = 0; i < rowCount; i++) {
        myJokes[i] = new xmlJokesReturn();
        myJokes[i].jokeID = dataTable.Rows[i][0].ToString();
        myJokes[i].joke    = dataTable.Rows[i][1].ToString();
        myJokes[i].rating = dataTable.Rows[i][2].ToString();
    }
    // catch problems within the stored procedure
    if(rowCount > 0) {
        return myJokes;
    } else {
        throw new jokeException("F_noJokes");
    }
    // catch problems with the database
} catch (Exception e) {
    throw e;
}
}

```

---

## Creating the Public Web Methods

You are now finished with the internal methods, and can now go about implementing the public Web methods for the jokes Web Service. Remember that we put all of those Web methods in the *jokes* class (the file on the Solutions Web site for the book is *jokes.asmx.cs*). Figures 8.41 through 8.46 detail the code for those public Web methods.



**Figure 8.41** *addJoke* Web Method (*jokes.asmx.cs*)

```

/// <summary>
///     The addJoke method adds a new joke to the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of registered user'>
/// </param>
/// <param name='joke'
///     type='string'
///     desc='the joke'>
/// </param>
/// <returns>nothing</returns>
[SoapDocumentMethodAttribute(Action="addJoke",
    RequestNamespace="urn:schemas-syngress-com-soap:jokes",
    RequestElementName="addJoke",
    ResponseNamespace="urn:schemas-syngress-com-soap:jokes",
    ResponseElementName="addJokeResponse")]
[WebMethod(Description="The addJoke method adds a new joke " +
    "to the database")]
public void addJoke(
    string userName, string password, string joke) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        jokesObj.addJoke(userName, password, joke);
    }
}

```

Continued

**Figure 8.41 Continued**


---

```

    // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
    }

    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
    }
}

```

---

**Figure 8.42 getJokes Web Method (jokes.asmx.cs)**


---

```

/// <summary>
///     The getJokes method gets howMany (moderated) jokes
///     from the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of registered user'>
/// </param>
/// <param name='howMany'
///     type='int'
///     desc='how many jokes we would like'>
/// </param>
/// <returns>an XML representation (xmlJokesReturn)
/// of howMany jokes</returns>
[SoapDocumentMethodAttribute(Action="getJokes",
    RequestNamespace="urn:schemas-syngress-com-soap:jokes",
    RequestElementName="getJokes",
    ResponseNamespace="urn:schemas-syngress-com-soap:jokes",
    ResponseElementName="getJokesResponse")]

```

---

**Continued**

**Figure 8.42 Continued**


---

```
[WebMethod(Description="The getJokes method gets <howMany> " +
    "(moderated) jokes from the database")]
[return: XmlElementAttribute("jokeData", IsNullable=false)]
public xmlJokesReturn[] getJokes(
    string userName, string password, int howMany) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        xmlJokesReturn[] myJokes =
            jokesObj.getJokes(userName, password, howMany);
        return myJokes;
    // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
        return null; // code never reached, but needed by compiler
    }
    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
        return null; // code never reached, but needed by compiler
    }
}
```

---

**Figure 8.43 addRating Web Method (jokes.asmx.cs)**


---

```
/// <summary>
///     The addRating method lets a user add a rating
///     for a joke to the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of registered user'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of registered user'>
```

---

Continued

**Figure 8.43 Continued**

---

```

/// </param>
/// <param name='rating'
///   type='int'
///   desc='rating of the joke (1-5)'>
/// </param>
/// <param name='jokeID'
///   type='int'
///   desc='ID of the joke'>
/// </param>
/// <returns>nothing</returns>
[SoapDocumentMethodAttribute(Action="addRating",
    RequestNamespace="urn:schemas-syngress-com-soap:jokes",
    RequestElementName="addRating",
    ResponseNamespace="urn:schemas-syngress-com-soap:jokes",
    ResponseElementName="addRatingResponse")]
[WebMethod(Description="The addRating method lets a user add a " +
    "rating for a joke to the database")]
public void addRating(
    string userName, string password, int rating, int jokeID) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        if((rating < 1) && (rating > 5)) {
            throwFault("Fault occurred", "F_ratingInvalid", userName);
        } else {
            jokesObj.addRating(userName, password, rating, jokeID);
        }
        // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
    }
    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
    }
}
}

```

---

**Figure 8.44** *getUnmoderated* Web Method (jokes.asmx.cs)

```
/// <summary>
///     The getUnmoderated method lets a moderator retrieve
///     howMany unmoderated jokes from the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator'>
/// </param>
/// <param name='howMany'
///     type='int'
///     desc='how many jokes we would like'>
/// </param>
/// <returns>an XML representation (xmlJokesReturn)
/// of howMany jokes</returns>
[SoapDocumentMethodAttribute(Action="getUnmoderated",
    RequestNamespace="urn:schemas-syngress-com-soap:jokes",
    RequestElementName="getUnmoderated",
    ResponseNamespace="urn:schemas-syngress-com-soap:jokes",
    ResponseElementName="getUnmoderatedResponse")]
[WebMethod(Description="The getUnmoderated method lets a " +
    "moderator retrieve <howMany> unmoderated jokes from " +
    "the database")]
[return: XmlElementAttribute("jokeData", IsNullable=false)]
public xmlJokesReturn[] getUnmoderated(
    string userName, string password, int howMany) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        xmlJokesReturn[] myJokes =
            jokesObj.getUnmoderated(userName, password, howMany);
        return myJokes;
    }
    // catch jokeExceptions
```

---

**Continued**

**Figure 8.44 Continued**


---

```

    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
        return null; // code never reached, but needed by compiler
    }

    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
        return null; // code never reached, but needed by compiler
    }
}

```

---

**Figure 8.45 addModerated Web Method (jokes.asmx.cs)**


---

```

/// <summary>
///     The addModerated method lets a moderator set a joke to be
///     'moderated', i.e. accessible to regular users
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator'>
/// </param>
/// <param name='jokeID'
///     type='int'
///     desc='ID of joke'>
/// </param>
/// <returns>nothing</returns>
[SoapDocumentMethodAttribute(Action="addModerated",
    RequestNamespace="urn:schemas-syngress-com-soap:jokes",
    RequestElementName="addModerated",
    ResponseNamespace="urn:schemas-syngress-com-soap:jokes",
    ResponseElementName="addModeratedResponse")]

```

---

Continued

**Figure 8.45 Continued**


---

```
[WebMethod(Description="The addModerated method lets a " +
    "moderator set a joke to be 'moderated', i.e. accessible " +
    "to regular users")]
public void addModerated(
    string userName, string password, int jokeID) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        jokesObj.addModerated(userName, password, jokeID);
        // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
    }
    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
    }
}
```

---

**Figure 8.46** *deleteUnmoderated* Web Method (jokes.asmx.cs)

---

```
/// <summary>
///     The deleteUnmoderated method lets a moderator delete a
///     (unmoderated) joke from the database
/// </summary>
/// <param name='userName'
///     type='string'
///     desc='name of moderator'>
/// </param>
/// <param name='password'
///     type='string'
///     desc='password of moderator'>
/// </param>
/// <param name='jokeID'
///     type='int'
///     desc='ID of joke'>
```

---

Continued



**Figure 8.46 Continued**


---

```

/// </param>
/// <returns>nothing</returns>
[SoapDocumentMethodAttribute(Action="deleteUnmoderated",
    RequestNamespace="urn:schemas-syngress-com-soap:jokes",
    RequestElementName="deleteUnmoderated",
    ResponseNamespace="urn:schemas-syngress-com-soap:jokes",
    ResponseElementName="deleteUnmoderatedResponse")]
[WebMethod(Description="The deleteUnmoderated method lets a " +
    "moderator delete a (unmoderated) joke from the database")]
public void deleteUnmoderated(
    string userName, string password, int jokeID) {
    jokesImplement jokesObj = new jokesImplement();
    try {
        jokesObj.deleteUnmoderated(userName, password, jokeID);
    // catch jokeExceptions
    } catch (jokeException e) {
        throwFault("Fault occurred", e.failReason, userName);
    }
    // then, catch general System Exceptions
    catch (Exception e) {
        throwFault(e.Message, "F_System", userName);
    }
}

```

---

Remember you need to either add the same error handling routine, *throwFault*, as you did for the *userAdmin* Web Service, or reference that method (in which case you need to modify its access scope).

This completes the Web Service section of the jokes Web Service. As mentioned before, you can find the complete code for the Jokes Web Service in the directory *jokesService* on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

Let's quickly review what you have done so far: You have implemented two out of three tiers of a complex Web application that delivers jokes to users using Web Services technology. You have set up a database back-end system to hold user and joke information, you have created business logic components to manage users and jokes, and you have implemented a data access mechanism

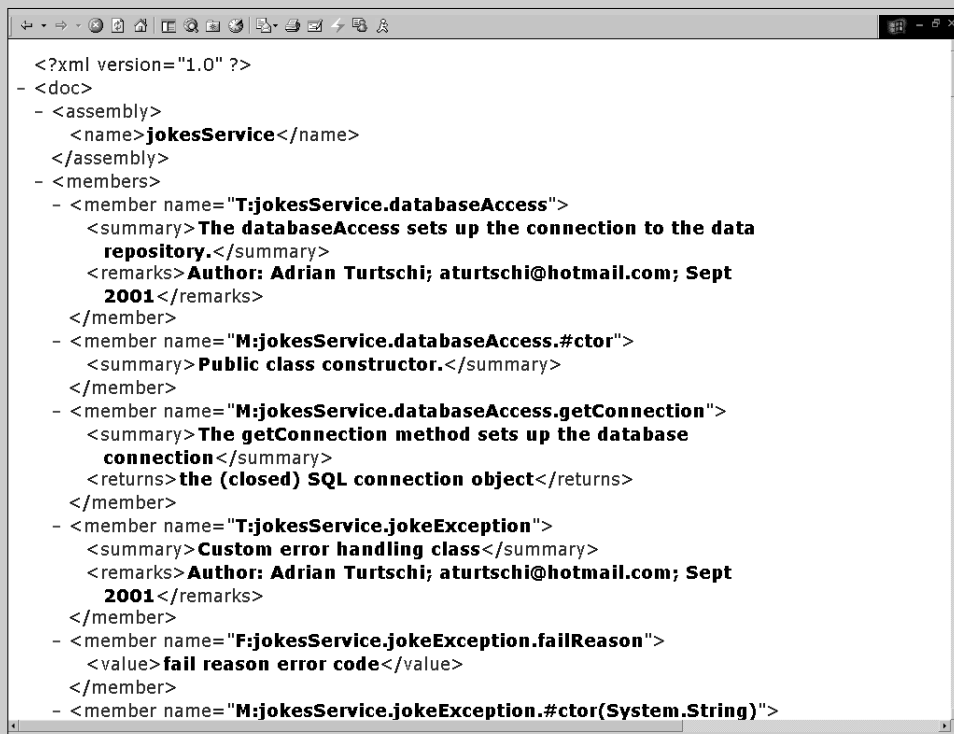
using Web Services. Although you could now go ahead and publish your Web Service in a UDDI registry and wait for clients out there to consume our Web Service, you should consider implementing an additional step and build a portal application that lets users interface with the jokes application through Windows forms.

## Developing & Deploying...

### Creating Human Readable Documentation

As you set up the Web Service project, you instructed the C# compiler to automatically create an XML documentation output file (refer back to Figure 8.12). If you now look at that file, you should see something similar to Figure 8.47.

**Figure 8.47** XML Documentation Generated by the C# Compiler (Excerpt)



```
<?xml version="1.0" ?>
- <doc>
- <assembly>
  <name>jokesService</name>
</assembly>
- <members>
- <member name="T:jokesService.databaseAccess">
  <summary>The databaseAccess sets up the connection to the data repository.</summary>
  <remarks>Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001</remarks>
</member>
- <member name="M:jokesService.databaseAccess.#ctor">
  <summary>Public class constructor.</summary>
</member>
- <member name="M:jokesService.databaseAccess.getConnection">
  <summary>The getConnection method sets up the database connection</summary>
  <returns>the (closed) SQL connection object</returns>
</member>
- <member name="T:jokesService.jokeException">
  <summary>Custom error handling class</summary>
  <remarks>Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001</remarks>
</member>
- <member name="F:jokesService.jokeException.failReason">
  <value>fail reason error code</value>
</member>
- <member name="M:jokesService.jokeException.#ctor(System.String)">
```

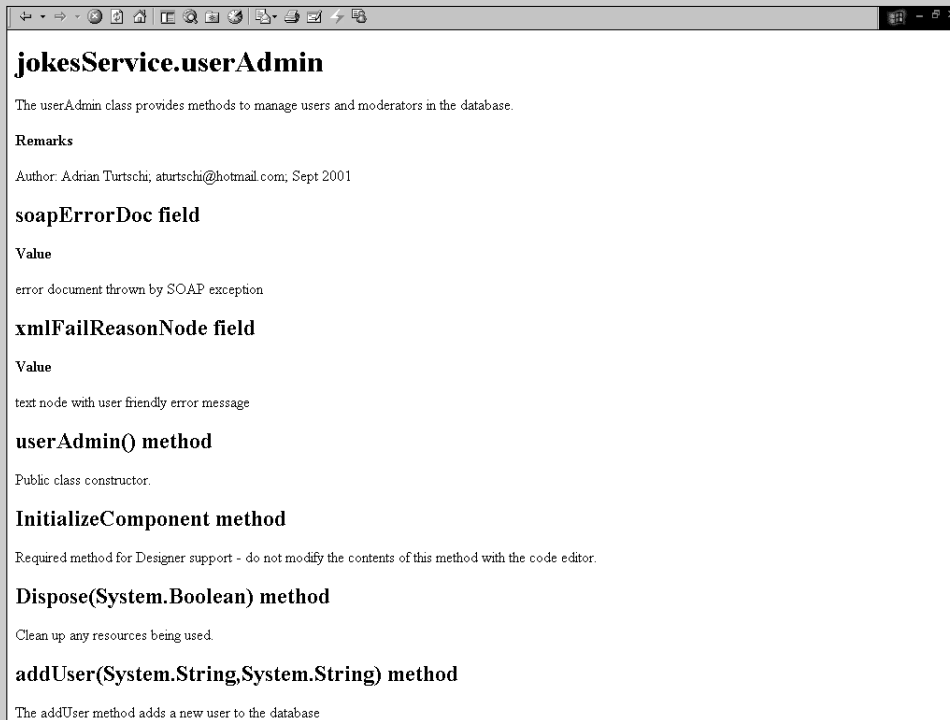
Continued

Although the comments appear as they should, this document needs some improvement to be truly useful for human consumption.

Dan Vallejo was kind enough to make an XSLT style sheet ([www.conted.bcc.ctc.edu/users/danval/CSharp/CSharp\\_Code\\_Files/doc.xsl](http://www.conted.bcc.ctc.edu/users/danval/CSharp/CSharp_Code_Files/doc.xsl)) publicly available on his C# Web site at [www.conted.bcc.ctc.edu/users/danval](http://www.conted.bcc.ctc.edu/users/danval) that generates a nice looking HTML documentation file. Although not quite as functionally rich as the documentation generated by, say, the *javadoc* tool in the Java world, it is a first step in the right direction. The XSLT file was originally conceived by Anders Hejlsberg. We use it by permission of the author.

After applying that style sheet, our documentation looks as in Figure 8.48:

**Figure 8.48** HTML Documentation after Applying a Style Sheet (Excerpt)

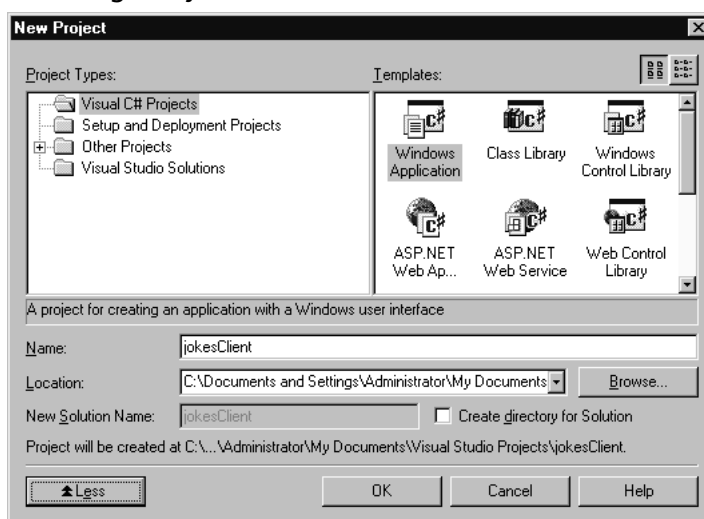


## Creating a Client Application

Let's go ahead and develop a simple Windows Forms-based client for our Jokes Web Service. The complete code for this application is on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)) in the directory `jokesClient`.

Start by opening up Visual Studio.NET. Go to **File | New | Project**, choose the entry **Windows Application** under the **Visual C# Projects** folder, keep the default Location, and enter **jokesClient** as the Name of the project, as indicated in Figure 8.49.

**Figure 8.49** Creating The *jokesService* Client as a Windows Forms Application



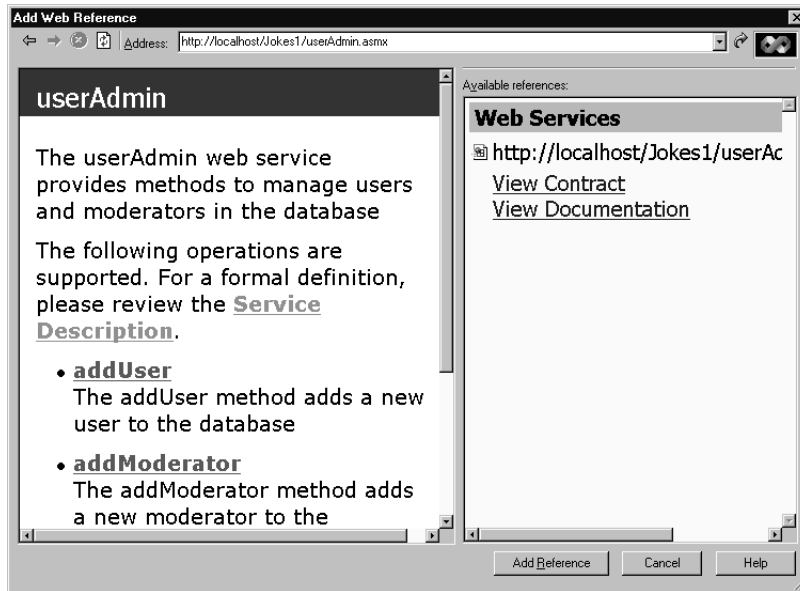
This will set up a new form, and create supporting project files.

Next, add a reference to the Jokes Web Server. Right click on the **jokesClient** project in the Solutions Explorer and select **Add Web Reference**. At the **Address** input box, enter **http://localhost/Jokes1/userAdmin.asmx**, as shown in Figure 8.50.

Once you verify that everything is fine, click **Add Reference**. Do the same for the Jokes Web Service, which is at the URL <http://localhost/jokesService/jokes.asmx>.

These references create the necessary proxy classes for your client to access our Jokes application. Keep in mind that those references are static, and as you change the Web Service public Web methods, you need to manually refresh the Web references. (You don't need to do this if you change the internal implementation classes, which was, after all, one of the reason we created them in the first place.)

Figure 8.50 Adding a Web Reference to the Web Service



The rest is simply an exercise in Windows Forms programming. Things to keep in mind are:

- Even though Web Services are stateless, you can let your users “log on” by asking for their credentials once, checking them against the user database with the *checkUser* Web method, and then cache them locally on the client
- The Web Service throws SOAP exceptions if things go wrong. You can extract a user-friendly message by looking at the *failReason* custom XML element in the SOAP exception return envelope.

Look at Figure 8.51, Figure 8.52, and Figure 8.53 to see how our client application looks.

Figure 8.51 The Web Service Client at Start Up

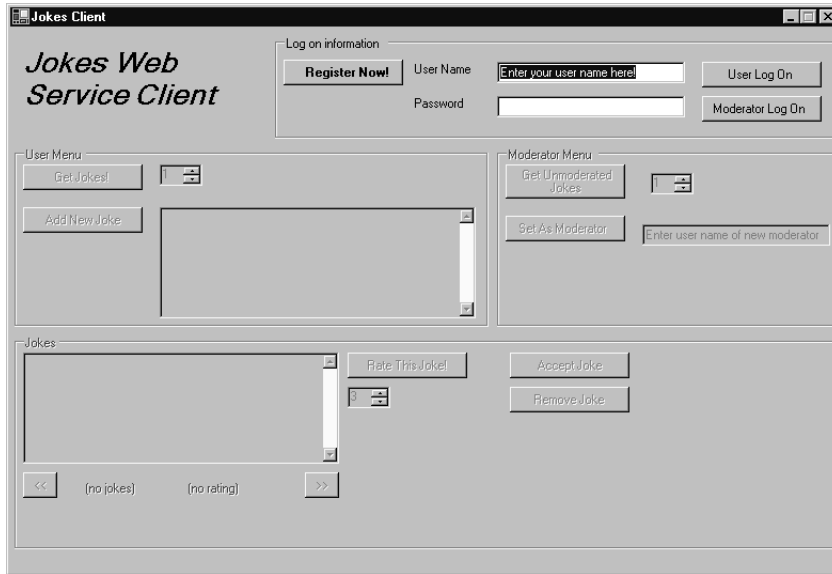
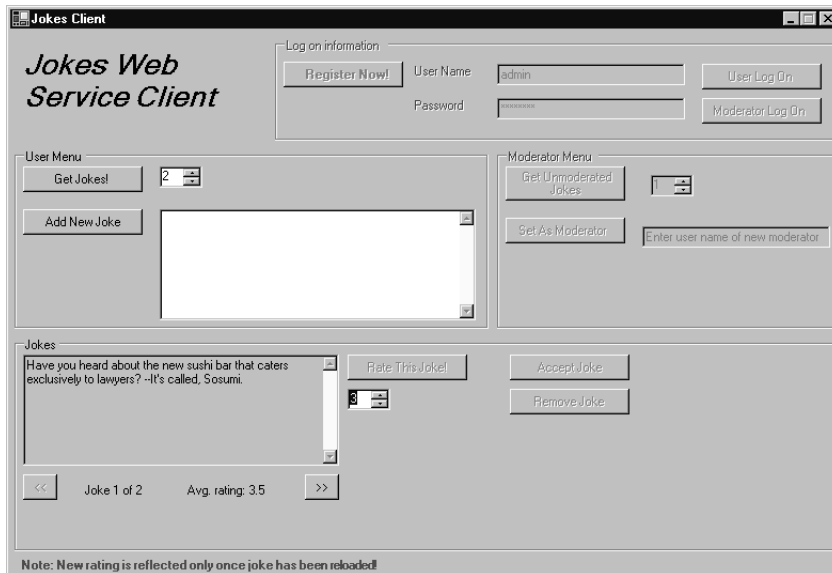


Figure 8.52 The Web Service Client after Logging On as a User, Retrieving Some Jokes, and Adding a User Joke Rating



**Figure 8.53** The Web Service Client after Logging On as a Moderator, Retrieving One Unmoderated Joke, and Accepting it to become Moderated

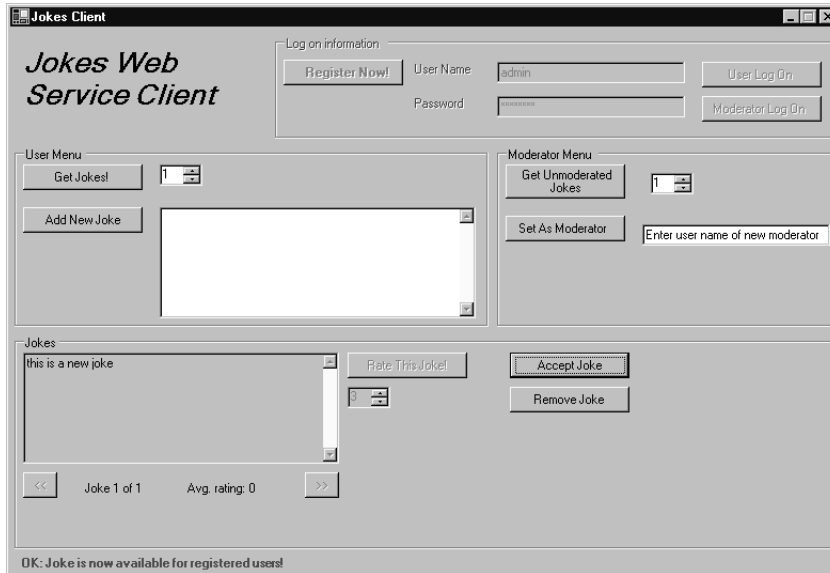


Figure 8.54 shows the code for the Jokes Client, ignoring code generated through the form designer. For the complete code see file `jokesClient.cs` on the Solutions Web site for the book ([www.syngress.com/solutions](http://www.syngress.com/solutions)).

**Figure 8.54** Jokes Client Application (`jokesClient.cs`)

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
using System.Web.Services.Protocols;
using System.Xml;

namespace jokesClient
{
    /// <summary>
    ///     Form f_jokeClient.
    /// </summary>
```

Continued

**Figure 8.54 Continued**

---

```
/// <remarks>
///   Author: Adrian Turtschi; aturtschi@hotmail.com; Sept 2001
/// </remarks>
public class f_jokeClient : System.Windows.Forms.Form
{
    // placeholders for Web Service objects
    private userAdmin.userAdmin userAdminObj ;
    private jokes.jokes jokesObj;
    // remembe if objects have been created
    private bool userAdminObjCreated = false;
    private bool jokesObjCreated = false;

    // remember user name and password, and moderator status
    private string userName = "";
    private string password = "";
    private bool isModerator = false;
    // hold jokes
    private jokes.xmlJokesReturn[] myJokes;
    private int jokesReturned = 0;
    private int currentJoke = 0;
    // are we looking at moderated jokes or not?
    private bool moderatedJokes = false;

    // IGNORE setting up of form elements

    public f_jokeClient() {
        InitializeComponent();
    }

    public void InitializeComponent() {
        // IGNORE
    }

    protected override void Dispose( bool disposing ) {
```

---

Continued



**Figure 8.54 Continued**


---

```

    if( disposing )
    {
        if (components != null)
        {
            components.Dispose();
        }
    }
    base.Dispose( disposing );
}

[STAThread]
static void Main() {
    Application.Run(new f_jokeClient());
}

private void displayJoke(
    string joke, int jokeNumber, int totalJokes, decimal rating,
    bool moderatedJokes) {
    this.l_statusMessage.Text = "";

    if(totalJokes == 0) {
        this.gb_jokes.Enabled = false;
        this.tb_jokesJoke.Text = "";
        this.nud_jokesRating.Value = 3;
        this.l_jokesNumber.Text = "(no jokes)";
        this.l_jokesRating.Text = "(no rating)";
        return;
    }

    if(totalJokes > 0) {
        this.gb_jokes.Enabled = true;
        if (!moderatedJokes) {
            this.b_jokesAddModerated.Enabled = true;
            this.b_jokesRemove.Enabled = true;

```

---

Continued

Figure 8.54 Continued

```
        this.nud_jokesRating.Enabled = false;
        this.b_jokesAddRating.Enabled = false;
    } else {
        this.b_jokesAddModerated.Enabled = false;
        this.b_jokesRemove.Enabled = false;
        this.nud_jokesRating.Enabled = true;
        this.b_jokesAddRating.Enabled = true;
    }
}

if(totalJokes > 1) {
    if(jokeNumber == 1) {
        this.b_jokesNext.Enabled = true;
        this.b_jokesPrev.Enabled = false;
    } else {
        if(jokeNumber == totalJokes) {
            this.b_jokesNext.Enabled = false;
            this.b_jokesPrev.Enabled = true;
        } else {
            this.b_jokesNext.Enabled = true;
            this.b_jokesPrev.Enabled = true;
        }
    }
} else {
    this.b_jokesNext.Enabled = false;
    this.b_jokesPrev.Enabled = false;
}

this.tb_jokesJoke.Text = joke;
this.l_jokesNumber.Text = "Joke " + jokeNumber.ToString()
    + " of " + totalJokes.ToString();
this.l_jokesRating.Text = "Avg. rating: " + rating.ToString();
}

private void logon(bool isModerator, bool register) {
```

Continued

**Figure 8.54 Continued**


---

```

string userName = this.tb_logonUserName.Text;
string password = this.tb_logonPassword.Text;
if((userName.Length > 0) && (password.Length > 0)
    && (userName.Length <= 20) && (password.Length <= 20)) {
    if(!this.userAdminObjCreated) {
        this.userAdminObj = new userAdmin.userAdmin();
        this.userAdminObjCreated = true;
    }
    try {
        // register new user?
        if(register) {
            // Call our Web Service method addUser
            this.userAdminObj.addUser(userName, password);
        } else {
            // Call our Web Service method checkUser
            this.userAdminObj.checkUser(
                userName.Substring(0,Math.Min(userName.Length, 20)),
                password.Substring(0,Math.Min(password.Length, 20)),
                isModerator);
        }
        // OK
        this.userName = userName;
        this.password = password;
        this.isModerator = isModerator;
        if(isModerator) {
            this.gb_moderatorMenu.Enabled = true;
        } else {
            this.gb_moderatorMenu.Enabled = false;
        }
        this.gb_userMenu.Enabled = true;
        this.gb_userInfo.Enabled = false;
        this.l_statusMessage.Text = "";
        displayJoke("", 0, 0, 0, this.isModerator);
        if(register) {
            this.l_statusMessage.Text = "OK: you have successfully " +

```

---

**Continued**

Figure 8.54 Continued

```
        "registered with the system!";
    }
} catch (SoapException ex) {
    XmlNode[] customErrorMsgs = ex.OtherElements;
    if(customErrorMsgs.Length > 0) {
        XmlNode customErrorMsg = customErrorMsgs[0];
        if (customErrorMsg.InnerText.Length > 0) {
            this.l_statusMessage.Text = "Error: " +
                customErrorMsg.InnerText;
            return;
        }
    }
} catch (Exception ex) {
    this.l_statusMessage.Text = "Error: " + ex.Message;
}
}
}

private void getJokes(int howMany, bool moderatedJokes) {
    try {
        if(!this.jokesObjCreated) {
            this.jokesObj = new jokes.jokes();
            this.jokesObjCreated = true;
        }
        // Call our Web Service method getJokes
        if(moderatedJokes) {
            myJokes = this.jokesObj.getJokes(
                userName, password, howMany);
        } else {
            myJokes = this.jokesObj.getUnmoderated(
                userName, password, howMany);
        }
        // OK
        this.jokesReturned = myJokes.Length;
        if(this.jokesReturned == 0) {
```

Continued

**Figure 8.54 Continued**


---

```

        displayJoke("", 0, 0, 0, this.isModerator);
    } else {
        this.currentJoke = 1;
        displayJoke(
            myJokes[this.currentJoke - 1].joke,
            this.currentJoke,
            this.jokesReturned,
            // need leading zero in case NULL is returned from
            // the database, i.e. joke unrated (which
            // will come back as zero length string)
            Decimal.Parse(
                "0" + myJokes[this.currentJoke - 1].rating),
            moderatedJokes);
    }
} catch (SoapException ex) {
    XmlNode[] customErrorMsgs = ex.OtherElements;
    if(customErrorMsgs.Length > 0) {
        XmlNode customErrorMsg = customErrorMsgs[0];
        if (customErrorMsg.InnerText.Length > 0) {
            this.l_statusMessage.Text =
                "Error: " + customErrorMsg.InnerText;
            return;
        }
    }
} catch (Exception ex) {
    this.l_statusMessage.Text = "Error: " + ex.Message;
}
}

private void b_logonUserLogOn_Click(
    object sender, System.EventArgs e) {
    logon(false, false);
}

private void b_logonModeratorLogOn_Click(

```

---

**Continued**

**Figure 8.54 Continued**

```
        object sender, System.EventArgs e) {
            logon(true, false);
        }

private void b_logonRegisterNow_Click(
    object sender, System.EventArgs e) {
    logon(false, true);
}

private void b_moderatorMakeModerator_Click(
    object sender, System.EventArgs e) {
    displayJoke("", 0, 0, 0, this.isModerator);
    string newModeratorUserName =
        this.tb_moderatorNewModeratorUserName.Text;
    if(newModeratorUserName.Length > 0) {
        newModeratorUserName = newModeratorUserName.Substring(
            0, Math.Min(newModeratorUserName.Length, 20));
        if(!this.userAdminObjCreated) {
            this.userAdminObj = new userAdmin.userAdmin();
            this.userAdminObjCreated = true;
        }
        try {
            // Call our Web Service method addModerator
            this.userAdminObj.addModerator(
                this.userName, this.password, newModeratorUserName);
            // OK
            this.l_statusMessage.Text =
                "OK: " + newModeratorUserName + " is now a moderator";
        } catch (SoapException ex) {
            XmlNode[] customErrorMsgs = ex.OtherElements;
            if(customErrorMsgs.Length > 0) {
                XmlNode customErrorMsg = customErrorMsgs[0];
                if (customErrorMsg.InnerText.Length > 0) {
                    this.l_statusMessage.Text =
                        "Error: " + customErrorMsg.InnerText;
                }
            }
        }
    }
}
```

---

**Continued**

**Figure 8.54 Continued**


---

```

        return;
    }
}
} catch (Exception ex) {
    this.l_statusMessage.Text = "Error: " + ex.Message;
}
}
}

private void b_userGetJokes_Click(
    object sender, System.EventArgs e) {
    displayJoke("", 0, 0, 0, this.isModerator);
    this.moderatedJokes = true;
    getJokes((int)this.nud_userHowMany.Value, this.moderatedJokes);
}

private void b_moderatorGetUnmoderated_Click(
    object sender, System.EventArgs e) {
    displayJoke("", 0, 0, 0, this.isModerator);
    this.moderatedJokes = false;
    getJokes(
        (int)this.nud_moderatorHowMany.Value, this.moderatedJokes);
}

private void b_jokesPrev_Click(object sender, System.EventArgs e) {
    // displayJoke() ONLY enables this button if there are jokes
    // to display, so we don't need a sanity check here.
    this.currentJoke = this.currentJoke - 1;
    displayJoke(
        myJokes[this.currentJoke - 1].joke,
        this.currentJoke,
        this.jokesReturned,
        Decimal.Parse("0" + myJokes[this.currentJoke - 1].rating),
        this.moderatedJokes);
}
}

```

---

**Continued**

**Figure 8.54 Continued**

```
private void b_jokesNext_Click(object sender, System.EventArgs e) {
    // displayJoke() ONLY enables this button if there are jokes
    // to display, so we don't need a sanity check here.
    this.currentJoke = this.currentJoke + 1;
    displayJoke(
        myJokes[this.currentJoke - 1].joke,
        this.currentJoke,
        this.jokesReturned,
        Decimal.Parse("0" + myJokes[this.currentJoke - 1].rating),
        this.moderatedJokes);
}

private void b_jokesAddRating_Click(
    object sender, System.EventArgs e) {
    try {
        if(!this.jokesObjCreated) {
            this.jokesObj = new jokes.jokes();
            this.jokesObjCreated = true;
        }
        // Call our Web Service method addRating
        this.jokesObj.addRating(
            userName,
            password,
            (int)this.nud_jokesRating.Value,
            Int32.Parse(this.myJokes[this.currentJoke-1].jokeID));
        // OK
        // try to tell user not to rate the joke again...
        this.b_jokesAddRating.Enabled = false;
        this.l_statusMessage.Text = "Note: New rating is " +
            "reflected only once joke has been reloaded!";
    } catch (SoapException ex) {
        XmlNode[] customErrorMsgs = ex.OtherElements;
        if(customErrorMsgs.Length > 0) {
            XmlNode customErrorMsg = customErrorMsgs[0];
            if (customErrorMsg.InnerText.Length > 0) {
```

---

Continued



**Figure 8.54 Continued**


---

```

        this.l_statusMessage.Text =
            "Error: " + customErrorMsg.InnerText;
        return;
    }
}
} catch (Exception ex) {
    this.l_statusMessage.Text = "Error: " + ex.Message;
}
}

private void b_jokesAddModerated_Click(object sender, System.EventArgs
e) {
    try {
        if(!this.jokesObjCreated) {
            this.jokesObj = new jokes.jokes();
            this.jokesObjCreated = true;
        }
        // Call our Web Service method addRating
        this.jokesObj.addModerated(
            userName,
            password,
            Int32.Parse(this.myJokes[this.currentJoke-1].jokeID));
        // OK
        this.l_statusMessage.Text =
            "OK: Joke is now available for registered users!";
    } catch (SoapException ex) {
        XmlNode[] customErrorMsgs = ex.OtherElements;
        if(customErrorMsgs.Length > 0) {
            XmlNode customErrorMsg = customErrorMsgs[0];
            if (customErrorMsg.InnerText.Length > 0) {
                this.l_statusMessage.Text =
                    "Error: " + customErrorMsg.InnerText;
                return;
            }
        }
    } catch (Exception ex) {

```

---

Continued

**Figure 8.54 Continued**

```
        this.l_statusMessage.Text = "Error: " + ex.Message;
    }
}

private void b_jokesRemove_Click(
    object sender, System.EventArgs e) {
    try {
        if(!this.jokesObjCreated) {
            this.jokesObj = new jokes.jokes();
            this.jokesObjCreated = true;;
        }
        // Call our Web Service method addRating
        this.jokesObj.deleteUnmoderated(
            userName,
            password,
            Int32.Parse(this.myJokes[this.currentJoke-1].jokeID));
        // OK
        this.l_statusMessage.Text = "OK: Joke has been removed!";
    } catch (SoapException ex) {
        XmlNode[] customErrorMsgs = ex.OtherElements;
        if(customErrorMsgs.Length > 0) {
            XmlNode customErrorMsg = customErrorMsgs[0];
            if (customErrorMsg.InnerText.Length > 0) {
                this.l_statusMessage.Text =
                    "Error: " + customErrorMsg.InnerText;
                return;
            }
        }
    } catch (Exception ex) {
        this.l_statusMessage.Text = "Error: " + ex.Message;
    }
}

private void b_userAddJoke_Click(
    object sender, System.EventArgs e) {
```

---

Continued

**Figure 8.54 Continued**


---

```

displayJoke("", 0, 0, 0, this.isModerator);
string newJoke = this.tb_userJoke.Text;
if(newJoke.Length > 0) {
    newJoke = newJoke.Substring(
        0,Math.Min(newJoke.Length, 3500));
    try {
        if(!this.jokesObjCreated) {
            this.jokesObj = new jokes.jokes();
            this.jokesObjCreated = true;
        }
        // Call our Web Service method addRating
        this.jokesObj.addJoke(
            userName,
            password,
            newJoke);
        // OK
        this.l_statusMessage.Text = "OK: Joke has been " +
            "submitted for consideration to the system!";
        this.tb_userJoke.Text = "";
    } catch (SoapException ex) {
        XmlNode[] customErrorMsgs = ex.OtherElements;
        if(customErrorMsgs.Length > 0) {
            XmlNode customErrorMsg = customErrorMsgs[0];
            if (customErrorMsg.InnerText.Length > 0) {
                this.l_statusMessage.Text =
                    "Error: " + customErrorMsg.InnerText;
                return;
            }
        }
    } catch (Exception ex) {
        this.l_statusMessage.Text = "Error: " + ex.Message;
    }
}
}
}
}

```

---

## Some Ideas to Improve the Jokes Web Service

If you like the idea of the Jokes application you may want to think about expanding it a little bit. It would be nice, for example, to get to know the users and to have a logging and reporting subsystem to identify who submits jokes, and which jokes are the most popular. Another idea would be to add additional meta-data to the jokes. For instance you could add joke categories that describe the joke subject matter. Along those lines you may want to have an additional Web Service that lets moderators manage those categories and add new ones. You could also delve into the internationalization classes that the .NET Framework has built in and localize status and error messages.

## Summary

In this chapter we have set out to develop a real-world Web Service application, namely a service that delivers jokes to the Internet community. We started out by gathering requirements, such that we want to know our users, that our users should be able to submit their own jokes and rate other user's jokes, and that there should be an administrative module in place to manage both users and jokes.

Our choice of developing this application as a Web Service was reinforced by the fact that Web Services make our application universally accessible, even for users behind corporate firewalls, and that Web Services give us support for non-English languages for free because they are based on XML and Unicode.

We started out our design by using a visual modeling tool in order to get a clear road map for our back-end and middle tier application architecture. We designed the various components of our application in such a way that we had a clear separation between a thin Web Service “front end” layer, and implementation classes where the business logic of our application sits. We abstracted access to the Microsoft SQL Server database by providing for wrapper methods for the SQL stored procedures and by creating a separate data access class. We also designed a security and error handling mechanism, and we made the first steps in implementing an application logging system based on interaction with the machine Event Log.

Once we had the database schema and the middle tier object model firmly in place, we started implementing the various pieces in a methodical way, starting at the back end. Because the various layers of our application are clearly separated, it would have been possible to create our project in a team of developers, say one person writing the back end infrastructure, one person writing the business logic, and a third person writing the actual Web Service itself.

Apart from encountering a very methodical way towards application development in general, we have seen a number of best practices in the area of Web Services:

- Don't put a lot of business logic into your Web Service classes! Have implementation classes do the heavy lifting. This way, you also don't limit yourself to Web Services as the only way to access your application; there may be instances where you want Internet users to access your application through Web Services, whereas it may be better for intranet users to use COM/DCOM or .NET Remoting.

- Put special emphasis on how the XML should look like between Web Service client and server. But don't limit yourself to the best case, rather decide from the very start how error information should be communicated to the client, particularly if the error can be corrected by the client. The SOAP Fault mechanism is a good start, but it has the disadvantage that it is an all-or-nothing mechanism. You may want to think about a scheme where the server can communicate to the client that *part* of the information it received was all right, but not all of it.
- There are alternatives to sending relation data through SOAP using .NET DataSets. If you think your clients will not all be running on Microsoft's .NET platform you may want to create an alternative (and simpler!) schema to bring such data to your clients.
- Because of inherent the limitations of state management in Web Services there are currently probably few alternatives than to send user authentication information to the server with every single Web Service request.
- Pay special attention to add documentation comments in our code throughout the project from the very start. You can then utilize the .NET feature to automatically generate project documentation files in XML format for you. You can use those files to generate your API documentation as a set of, say, HTML pages.

Lastly, we developed a client application based on Windows Forms to use our service.

## Solutions Fast Track

### Motivations and Requirements for the Jokes Web Service

- ☑ Internet based applications have to be *universally accessible*; on a technical level, this means they should work well with corporate firewalls, and on a user level, they have to support an international audience. Both can be achieved by employing Web Service technology.

## Functional Application Design

- ☑ Security, state management, and error handling are critical elements of application architecture that need to be considered first.

## Implementing the Jokes Data Repository

- ☑ Visual Studio.NET includes a fully working copy of Microsoft's SQL Server Desktop Engine.
- ☑ Visual Studio.NET's Server Explorer lets you interface with data repositories such as Microsoft SQL Server, including both reading and writing database schemas and data.
- ☑ Starting the application development process by implementing the back-end first is usually a good idea.

## Implementing the Jokes Middle Tier

- ☑ Visual Studio.NET continues in the tradition of the Visual Studio product line in being a very comfortable and efficient environment for application development.
- ☑ It is often a good idea to extend the *System.Exception* class to add custom error handling mechanisms, such as additional logging functionality.
- ☑ When throwing a new exception in a Web Service context the .NET runtime will automatically send a SOAP Fault back to the client application.
- ☑ The .NET Framework allows you to extend SOAP Faults to include custom XML elements, such as user friendly status or error information.
- ☑ Web Service security can either be implemented using the standard ASP.NET security mechanisms, or using a custom authentication and authorization scheme. We have chosen the latter method and implemented a stateless security system for the Jokes Web Service.

## Creating a Client Application

- ☑ Web Service clients that run on the .NET Framework can be very easily created through employing Web References.
- ☑ Caching user credentials on the client is one way to address state management and security.

## Frequently Asked Questions

The following Frequently Asked Questions, answered by the authors of this book, are designed to both measure your understanding of the concepts presented in this chapter and to assist you with real-life implementation of these concepts. To have your questions about this chapter answered by the author, browse to [www.syngress.com/solutions](http://www.syngress.com/solutions) and click on the “Ask the Author” form.

**Q:** My back-end data repository is not Microsoft SQL Server. How do I go about accessing my data?

**A:** One solution is to use the data access classes provided in the *System.Data.OleDb* namespace, that allow you to open data connections to essentially all the data sources that have OLEDB providers, such as Microsoft Office files or Oracle databases. However, because the .NET Framework is still very new, you may run into problems if you stray too far from the main stream. For instance, those classes don't currently work well with Microsoft's own Exchange 2000 Web Storage System, particularly if you are dealing with multi valued fields. Your last recourse is to use straight OLEDB or straight ADO through the .NET COM Interoperability layer.

**Q:** What options do I have to have the Jokes Web Service perform better?

**A:** As a first step, you want to build the application in Release configuration. Because Jokes is an ASP.NET application, you then have all the performance options of ASP.NET at your disposal. Particularly, you may want to look into the ASP.NET caching system.



**Q:** How do I deploy a Web Service?

**A:** Deploying an ASP.NET application is as easy as creating a new IIS virtual directory on a production machine and copying all the application files into the new location. Be sure, though, to compile your application in Release mode.

**Q:** Do I need .NET on the client to use the Jokes application?

**A:** No, not at all. While the client application we created in this chapter does in fact run only on a machine that has the .NET Framework installed, this is not a prerequisite. Any client that allows you to call a Web Service will do; specifically, all you need is a client that can send data over HTTP—so you can certainly go ahead and write a client that runs in a Web browser.

## A

- Absolute positioning, 268
  - Access 2000, 234, 238–242, 250, 252
  - Active Data Objects (ADO), 300, 302
  - Active Server Page (ASP), 10
  - Ad hoc queries, 305
  - Adding data, 272
  - Adding new users, 371–375
  - Administration
    - customer, 275–278
    - user, 371–390
    - Web sites, 266–267
  - Administration login (adminLogin.aspx), 266–267
  - Administration page (adminPage.aspx), 268–274
  - ADO. *See* Active Data Objects (ADO)
  - ADOCatalog, 278–284
  - ADO.NET
    - based on XML, 68, 81
    - namespace of classes, 49
    - .NET framework, 56–57
    - XML Schema Definition (XSD), 76
  - AirportWeather example, 225–227
  - Aliasing, 248
  - Antechinus C# editor, 44
  - Application design
    - adding new users, 371–375
    - administration login (adminLogin.aspx), 266–267
    - administration page (adminPage.aspx), 268–274
  - ADOCatalog, 278–284
  - business logic layer, implementing, 361
  - catalog, rendering, 289–290
  - checking existing user information, 376–379
  - client application, creating, 423–438
  - code, creating, 290–292
  - customer administration, 275–278
  - database-access component, 369–371
  - error handling, 345, 349, 366–369, 384–386
  - improvement ideas, 439
  - moderators, 379–381
  - motivation and requirements, 338–339
  - returning highly structured data, 390–393
  - security considerations, 344–345
  - start.aspx page, 288–289
  - state management, 345
  - stored procedures, creating, 244–250, 348–361
  - user administration service, 371–390
  - user interface, 287–292
  - Visual Studio project, 361–365
  - Web Service architecture, defining, 342–345
  - Web Services, creating, 250–259
  - Web Services, implementing, 361
  - Web Services, testing, 259–263
  - Web site, administration of, 266–275
  - Web site, designing, 264–266
  - wrapping stored procedure calls, 393–399
  - WSDL Web references, 263–264
  - XMLCart, 284–287
  - See also* Data; Databases; Public Web methods
- Application development, 43–55
  - Application event log, 366
  - Argument types, wrong, 165
  - Arithmetic, floating-point, 161
  - Arrays, 20–21
  - .asmx files, 10–11

- ASP. *See* Active Server Page
- asp controls, 288
- ASP.NET
  - .asmx extension, 9–11
  - building Internet applications, 30
  - DataSet* class, 21
  - Hello World* class, 3
  - IIS 5.0 prerequisite, 65
  - installation, 36
  - message transmission over HTTP, 18
  - .NET framework, 55–56
  - premium, 65
  - primary user interfaces, 60
  - Service1 default Web Service, 362
  - Web Forms, 30
  - Web Services, 61
  - Web Services, testing, 259–263
  - Windows 95/98 not supported, 32
  - XML knowledge requirement, 68
- ASP.NET/ADO.NET shopping cart. *See* Shopping-cart example
- .aspx files, 11–15, 55
- Assemblies, 39, 42
- Authentication schemes, 191–192, 306
- Authorization, 203
- Auto-generated code, customizing, 15
- autonumber* data type, 240, 243
- Available physical memory, 177
  
- B**
- Base Class Libraries, 30, 49–55
- Binary data, 207
- Building applications, 7
- Business logic layer, 361
  
- C**
- C# language, 43–45, 59–60
- Caching, smart, 56
- Catalog, rendering, 289–290
- Checking existing user information, 376–379
- CIL. *See* Intermediate Language (IL)
- Classes
  - loading in CLR, 37
  - transparent, 17
- Cleanup of objects, 37–38
- Client application
  - creating, 423–438
  - writing, 167–174
- Client proxy class, 158
- CLR. *See* Common Language Runtime
- Code
  - creating, 290–292
  - documenting, 365
- Code-behind, 55, 253, 267
- codebehind* attribute, 146
- COM. *See* Component Object Model
- Command-line compilation, 45–46
- Command-line utilities
  - osql* command line utility, 346
  - in Web Services Description Language (WSDL), 15, 21
- Common Intermediate Language (CIL). *See* Intermediate Language (IL)
- Common Language Runtime (CLR)
  - assemblies, 39
  - classes, loading, 37
  - cross-language interoperability, 38–39
  - deployment support, 41
  - description, 30, 36
  - exception handling, 39, 52–55
  - Intermediate Language (IL), 7
  - just-in-time (JIT) compilation, 38
  - managed code, 41–42
  - metadata, 40, 384
  - namespaces, 42

- object-lifetime management, 37–38
  - safety and security checks, 37
  - versioning support, 41
  - Common Object Request Broker Architecture (CORBA), 3, 8, 138, 202
  - Common Type System (CTS), 38
  - Compilers, 45–46
  - Compiling applications, 7, 40
  - Complex data types, 159–162
  - Component Object Model (COM), 138, 202, 212
  - Components
    - .NET framework, 55–61
    - self-describing, 40
  - Connection string, 253
  - Conventions, naming, 149, 238
  - Converting legacy data, 82
  - Cookies, 189, 191–194
  - CORBA. *See* Common Object Request Broker Architecture
  - CorDbg debugging tool, 48
  - Counters, hit, 189, 191–196, 199
  - CPU requirements, 32
  - Create Stored Procedure Wizard, 245
  - Credentials, checking, 376–379
  - Cross-browser Dynamic HTML (DHTML), 268
  - Cross-language interoperability, 38–39
  - Cryptography, Public Key (PKC), 203
  - CTS. *See* Common Type System
  - Currency data type, 240
  - Customer administration, 275–278
  - Customizing auto-generated code, 15
- D**
- Data
    - adding, 272
    - binary, 207
    - deleting, 272–273
    - displaying, 272
    - retrieving, 270–272
    - returning, 390–393
    - updating, 273–275
    - See also* Legacy systems and data
  - Data types, 19–21, 159–162, 165, 239–240
  - Database-access component, 369–371
  - databaseAccess.cs* class, 370
  - Databases
    - creating in Access, 238–242
    - creating in SQL Server, 242–250, 302–303
    - denormalizing, 241
    - designing, 234–237
    - entities, 236
    - Extensible Markup Language (XML), 124–125
    - gateway to, 369
    - installation script, 346–348
    - installing, 346–348
    - many-to-many relationships, 236
    - Northwind, 179
    - one-to-many relationships, 236
    - queries, 125–127
    - relational, 235–236
    - schema, defining, 302, 341
    - splitter tables, 236
    - stored procedures, creating, 244–250
  - DataGrid* control, 104, 107, 272–274, 278, 317
  - DataReader* classes, 57
  - DataSet* class
    - methods, 124
    - passing over SOAP, 179–182
    - reading XML documents, 127–128

- representation, 20
  - update ability, 57
  - using, 21–23
  - Date/time data type, 240
  - DbgClr debugging tool, 48
  - DCOM. *See* Distributed Component Object Model
  - Debugging
    - Start page, 5
    - System.Diagnostics* namespace, 50
    - tools, 48–49
  - Deleting data, 272–273
  - Denormalizing databases, 241
  - Deploying Web Services, 159
  - Deployment projects, 41
  - Description* namespace in
    - System.Web.Services*, 17
  - Deserializing, 164–165, 170
  - Design
    - databases, 234–237
    - separated from coding, 145–146, 168
    - Web sites, 264–266
    - See also* Application design
  - Design Time Controls (DTCs), 278
  - Designing Web sites, 264–266
  - Developing applications, 43–55
  - Development platforms, 43–44
  - DHTML. *See* Dynamic HTML
  - Diagnostics* namespace, 50
  - Digital Signature, 203
  - DISCO
    - description, 210, 217–219
    - .disco* files, 17, 217
    - disco.exe tool, 212, 219
    - failure, 170
    - generating DISCO files, 218–219
    - Visual Studio .NET (VS.NET), 218
  - Discovery* namespace in
    - System.Web.Services*, 17–18
  - Displaying data, 272
  - Distributed Component Object Model (DCOM), 3, 8, 138
  - div*, 289
  - Document Object Model (DOM), 68, 93–94, 133
  - Documenting code, 365–365, 421–422
  - Documents, XML
    - components, 72–75
    - creating, 70–72
    - database queries, 125–127
    - DataSet* class, 127–128
    - generating, 90–93, 365
    - loading, 99–100
    - navigating, 87–89, 94–95
    - parsing, 85–87, 95–98
    - processing in .NET framework, 81–84
    - reading, 82–89
    - sending, 186
    - storing, 83–84
    - structure, 69, 80–81
    - transforming to HTML, 116–118
    - transforming with XSLT, 115–124
    - valid, 76–80
    - well-formed, 75–76
    - writing, 82–83
    - See also* Extensible Markup Language (XML)
  - DOM. *See* Document Object Model
  - Drill-down functionality, 107
  - DTCs. *See* Design Time Controls
  - Dynamic HTML (DHTML), cross-browser, 268
- ## E
- Echo Web Service example, 140–143, 151–152, 213–217
  - EDI, 138
  - EM. *See* Enterprise Manager

Enterprise Manager (EM), 242  
 Entities, database, 236  
 Enumeration types, 20  
 Envelope, SOAP, 153–155, 390–393  
 Error handling, 162–167, 345, 366–369, 384–386  
   *See also* Exception handling  
 Event log, 366  
 Exception handling, 39, 52–55  
 Exceptions in server code, 165–167  
 Extensible Markup Language (XML)  
   description, 68  
   future, 70  
   schema, 16, 76–79  
   updateagrams, 308  
   validation against schema, 391  
   validation in VS.NET, 79–80  
 VS.NET XML Designer, 71–72  
 Web Services, 16  
   *XmlDocument* class, 98–107  
   *XmlTextReader* class, 84–89  
   *XmlTextWriter* class, 90–93  
   *XPathDocument* class, 107–110, 112–115  
   *XPathNavigator* class, 107, 110–115  
   *See also* Documents, XML  
 Extensible Stylesheet Language  
   Transformations (XSLT), 3, 115–124, 422

## F

Faults, Simple Object Access Protocol (SOAP), 164–165, 167, 384–386  
 File extensions, mapping, 143  
 Floating-point arithmetic, 161  
 Foreign keys, 236  
 FTP, 41  
 Future of XML, 70

## G

Garbage collection, 37–38  
 Gateway to databases, 369  
 Generating XML documents, 90–93, 365  
*GET* method, 152–153, 161

## H

Handling exceptions, 39, 52–55  
 Hard-disk requirements, 32  
 Hardware requirements for SDK, 32  
 Headers  
   HTTP, 191–194  
   Simple Object Access Protocol (SOAP), 186–187, 194–202  
 Hejlsberg, Anders, 422  
 Hit counters, 189, 191–196, 199  
 HTML, transforming XML documents to, 116–118  
 HTTP  
   ASP.NET message transmission, 18  
   cookies, 191–194  
   inherently stateless, 188  
   messaging, 3, 8, 16  
   SOAP headers, 194–202  
   state management, 191–202

## I

IDE, 348  
 IIS. *See* Internet Information Server  
 IL. *See* Intermediate Language  
 ILDasm. *See* Intermediate Language Disassembler  
 Improvement ideas, 439  
 Industry standards, nonproprietary, 16  
 Installation  
   ASP.NET, 36  
   database, 346–348

- .NET framework, 34–36
  - software development kit (SDK), 31, 33–34
- Installing .NET framework, 34–36
- Integrating different systems, 139
- IntelliSense, 171, 177, 297
- Intermediate Language Disassembler (ILDasm), 46–47
- Intermediate Language (IL)
  - building and compiling, 7
  - just-in-time (JIT) compilation, 38
- Internet Information Server (IIS), 65, 140
- Interoperability
  - cross-language, 38–39
  - unmanaged code, 42
- Interoperability Web site, 208

**J**

- Java packages, 42
- Javadoc, 422
- JIT. *See* Just-in-time compilation
- Jokes Web Service example
  - adding new users, 371–375
  - business logic layer, implementing, 361
  - checking existing user information, 376–379
  - client application, creating, 423–438
  - database, installing, 346–348
  - database-access component, 369–371
  - database schema, defining, 341
  - error handling, 345, 349, 366–369, 384–386
  - improvement ideas, 439
  - jokes and ratings, managing, 399–407
  - middle tier, implementing, 361
  - moderators, 379–381
  - motivation and requirements, 338–339

- public methods, defining, 340
- public Web methods, creating, 413–422
- public Web methods, testing, 389–390
- public Web methods for administrators, 386–388
- public Web methods for users, 381–384
- returning highly structured data, 390–393
- returning jokes, 407–412
- security considerations, 344–345
- state management, 345
- stored procedures, creating, 348–361
- user administration service, 371–390
- Visual Studio project, 361–365
- Web Service, implementing, 361
- Web Service architecture, defining, 342–345
- wrapping stored procedure calls, 393–399

- Jsc.exe (Jscript.NET) compiler, 45
- Just-in-time (JIT) compilation, 38

**L**

- Language choice, 45
- Legacy systems and data
  - converting, 82
  - Extensible Markup Language (XML), 82
  - migrating, 42, 300–301
  - SQLXML Web Services, 335
  - Web Services use, 8, 16
  - wrappers, 159, 342
- Lifetime management for objects, 37–38
- Loading XML documents, 99–100
- Localhost, 278
- Logging application events, 366

**M**

Maintaining state, 187–188  
 Malformed SOAP requests, 163–165  
 Managed code, 36, 38–39, 41–42  
 Managing object lifetimes, 37–38  
 Many-to-many relationships, 236  
 Marshalling of data, 21  
 Marshalling of data types, 19–21  
 MDAC. *See* Microsoft Data Access Components  
 Memo data type, 240  
 Memory available, 177  
 Messaging, 3  
 Metadata, 40, 280, 384  
 Method mapping, 311  
 Method messaging, 3  
 Microsoft. *See* Access 2000; Databases, Northwind; DISCO; IntelliSense; Internet Information Server (IIS); .NET framework; Passport service; SQL Server 2000; Windows 95/98; Windows Forms; Windows Installer; Windows NT  
 Microsoft Data Access Components (MDAC), 33–34  
 Microsoft Developer Network (MSDN), 34  
 Microsoft Intermediate Language (MSIL). *See* Intermediate Language (IL)  
 Minimum requirements. *See* Requirements  
 Moderators, 346, 379–381, 386  
 Moore, M., 203  
 Motivation, 338–339  
 Mozilla project, 268  
 MSDN. *See* Microsoft Developer Network  
 MSIL. *See* Intermediate Language (IL)

Multiple tables of *XmlDataDocument* objects, 103–107

**N**

Namespaces, 42  
 Naming conventions, 149, 238  
 Native Image Cache, 46  
 Navigating XML documents, 87–89, 94–95  
 .NET framework  
   ADO.NET, 56–57  
   ASP.NET, 55–56  
   Base Class Libraries, 30, 49–55  
   C# language, 59–60  
   client proxy class, 158  
   compilers, 45–46  
   components, 55–61  
   debugging tools, 48–49  
   description, 30  
   developing applications, 43–55  
   development platforms, 43–44  
   development tool, 43  
   documents, processing, 81–84  
   file extensions, mapping, 143  
   installation, 34–36  
   Intermediate Language Disassembler (ILDasm), 46  
   language choice, 45  
   NGEN.exe, 46–48  
   redistributable package (runtime), 33  
   tools, 46–49  
   VB.NET, 57–59  
   Web Services, 61  
   Windows Forms, 60–61  
   *See also* Common Language Runtime (CLR); Software development kit (SDK)  
 Netscape, 268



New users, adding, 371–375  
 NGEN.exe tool, 46–48  
 Nonproprietary industry standards, 16  
 Northwind database, 179  
 Number data type, 240

## O

Object lifetime management, 37–38  
 Object messaging, 3  
 OLE objects, 240  
 OleDb connections, 241  
 OLEDB managed provider, 56  
*OleDbDataReader* class, 57  
 One-to-many relationships, 236  
 Open-source Mozilla project, 268  
 Operating system requirements for SDK, 33  
 Order entries, multiple, 241  
 Organization, project, 143–146  
*osql* command line utility, 346

## P

Parameterized queries, 250  
 Parsing XML documents, 85–87, 95–98  
 Passing over SOAP  
   headers, 186–187  
   objects, 174–179  
   relational data (*DataSets*), 179–182  
   XML documents, 182–186  
 Passport service, 189  
 Performance Monitor tool, 174  
*PerformanceCounter* class, 174–177, 179  
 Physical memory available, 177  
 PKC. *See* Public Key Cryptography  
 Platforms for development, 43–44  
 Port 80, 3, 8  
 Positioning, absolute, 268  
 POST method, 152–153

Primitive types, 19–20  
 Procedures, stored. *See* Stored procedures  
 Processor requirements, 32  
 Project Explorer. *See* 297  
 Project organization, 143–146  
*Protocols* namespace in  
   *System.Web.Services*, 18–19  
 Proxy classes, 158, 170–171, 223  
 Public Key Cryptography (PKC), 203  
 Public Web methods  
   for administrators, 386–388  
   creating, 413–422  
   defining, 340  
   error handling, 384–386  
   testing, 389–390  
   for users, 381–384

## Q

Queries, ad hoc, 305  
 Queries, parameterized, 250

## R

Random access memory (RAM), 32  
 Reading XML documents, 82–89  
 Redistributable package (runtime), 33  
 Relational databases, 235–236  
 Relational view of *XmlDataDocument* objects, 100–103  
 Relationships, database, 236, 241  
 Remote procedure calls (RPC), 8, 202  
*RequiredFieldValidator* server control, 266  
 Requirements  
   hardware for software development kit (SDK), 32  
   jokes Web Service example, 338–339  
   operating system for software development kit (SDK), 33

XML knowledge for ASP.NET, 68

Retrieving data, 270–272

Returning highly structured data,  
390–393

RPC. *See* Remote procedure calls

Runtime (redistributable package), 33

## S

Safety and security checks in CLR, 37

Schema

defining, 302, 341

XML, 76–80

SDK. *See* Software Development Kit

SDL. *See* Service Definition Language

Security considerations, 335, 344–345

Self-describing components, 40

Serializing, 170, 174–175, 178–179,  
391–393

Server application event log, 366

Server Explorer tool, 346, 362–364

Servers

.asmx files, 10–11

.aspx files, 11–15

Simple Object Access Protocol  
(SOAP), 8

Service Definition Language (SDL), 17

Service1 default Web Service, 362

Shopping-cart example

administration login

(adminLogin.aspx), 266–267

administration page (adminPage.aspx),  
268–274

ADOCatalog, 278–284

Book Shop Web Services, 250–253

cart, rendering, 290

catalog, rendering, 289–290

code, creating, 290–292

customer administration, 275–278

data, 270–275

database, creating in Access, 238–242

database, creating in SQL Server,  
242–250

database, designing, 234–237

start.aspx page, 288–289

stored procedures, creating, 244–250

user interface, 287–292

Web Services, creating, 250–263,  
253–259

Web Services, testing, 259–263

Web site, administration, 266–275

Web site, designing, 264–266

Web site, overview, 265

WSDL Web references, 263–264

XMLCart, 284–287

Simple Object Access Protocol (SOAP)

client application, writing, 167–174

creating Web Services, 139–145

*DataSet* class, 179–182

definition, 137–138

deserializing, 164–165, 170

Digital Signature, 203

envelope, 153–155

exceptions, 424

faults, 164–165, 167, 384–386

headers, 186–187, 194–202

HTTP body, 194–202

HTTP header, 191–194

implementations, list of, 207

implementations, Microsoft variety of,  
211

interoperability Web site, 208

malformed SOAP requests, 163–165

message patterns, 137

messaging over HTTP, 3, 8, 16

namespaces, 384

non-Microsoft systems, 339

- passing objects, 138, 174–179
  - passing relational data (*DataSets*), 179–182
  - project organization, 143–146
  - rationale, 138
  - return envelopes, 390–393
  - security considerations, 202–203
  - sending XML documents, 186
  - serializing, 170, 174–175, 178–179, 391–393
  - standardization, 137
  - state, maintaining, 187–188
  - type marshalling, 19
  - URL mangling, 189–191
  - Web Services, running, 146–147
  - Web Services, testing, 147–159
  - Web sites, 137
  - wrong argument types, 165
  - XML documents, 182–186
- Smart caching, 56
- SOAP. *See* Simple Object Access Protocol
- “SOAP Messages with Attachments” Web site, 186
- SOAPBuilders Interoperability Lab, 208
- Software Development Kit (SDK)
- description, 31
  - hardware requirements, 32
  - installation, 31, 33–34
  - obtaining, 31–34
  - operating system requirements, 33
  - Visual Studio .NET (VS.NET), 31
  - Web sites for downloading, 34
- Solution Explorer, 10, 146–147, 265, 364
- Splitter tables, 236
- SQL Managed Provider, 372
- SQL record set, fetching, 390
- SQL Server 2000
- alternative, 346
  - limitations on row size, 341
  - service packs, 300
  - virtual directory, creating, 305–310
  - Web Services Toolkit, 301, 305
- SQL Server client tools, 348
- SQL Server Desktop Engine, 346, 348
- SQL Template queries, 309
- SqlDataReader* class, 57
- SQLServer 7 managed provider, 56
- SQLXML Web Services
- description, 300–301
  - legacy systems and data, 335
  - version 3.0, 305, 309
  - virtual directory, 305–307
- Stack trace information, 52–53
- Standard primitive types, 19
- Standards, nonproprietary, 16
- Standards for Web Services, 210–211
- Start page, 5
- Start.aspx page, 288–289
- State management
- cookies, 191–194
  - HTTP, adding state to, 188, 191, 202
  - HTTP body, 194–202
  - HTTP header, 191–194
  - jokes Web Service example, 345
  - maintaining, 187–188
  - SOAP header, 194–202
  - URL mangling, 189–191
- Stored procedures
- creating, 244–250, 303–305, 348–361
  - enabling for SOAP, 310–312
  - limitations, 241
  - using, 301–302
  - wizard, 245
  - wrapping, 250, 393–399

Structs, 20  
 Structured data, returning, 390–393  
 Structured exception handling. *See*  
   Exception handling  
 System namespace, 49–51  
 System.Web.Services namespace, 17–19

## T

T-SQL, 351  
 Tables of *XmlDataDocument* objects,  
   103–107  
 TcpTunnelGui program, 189  
 Testing  
   public Web methods, 389–390  
   Start page, 5  
   Web Services, 147–159, 259–263  
 Text data type, 240  
 TimeTrack example  
   client application, creating, 313–317  
   database, creating, 302–303  
   description, 301–302  
   SQL Server virtual directory, creating,  
     305–310  
   stored procedures, creating, 303–305  
   stored procedures, enabling, 310–312  
   Web Services, consuming, 317–318  
 tModel, 221  
 Tools, 46–49  
 Tracking users' paths, 276  
 Transactions, 250  
 Transforming documents with XSLT,  
   115–124  
 Transparent classes, 17  
 Tunneling, 173, 189  
 Turtschi, A., 203  
 Type marshalling, 19–21  
 Types. *See* Data types

## U

UDDI. *See* Universal Description,  
 Discovery, and Integration  
 Uniform Resource Identifier (URI),  
   149  
 Universal Description, Discovery, and  
 Integration (UDDI), 210, 219–227  
 Unmanaged code. *See* Managed code  
 Updategrams, 308  
 Updating data, 273–275  
 Upsizing Wizard, 242  
 URI. *See* Uniform Resource Identifier  
 URL mangling, 189–191  
 URLs. *See* Web sites  
 User administration service, 371–390  
 User information  
   checking, 376–379  
   keeping, 202  
   paths, tracking, 276  
 User interface, 287–292

## V

Vallejo, Dan, 422  
 Vbc.exe (VB.NET) compiler, 45  
 VB.NET, 5, 57–59  
 Versioning support, 41  
 Video requirements, 32  
 Virtual directory, creating, 305–310  
 Visual Studio .NET (VS.NET)  
   converting legacy data, 82  
   creating .disco files, 17  
   creating Web Services, 3–5  
   IDE, 348  
   marshalling of data, 21  
   .NET Software Development Kit  
     (SDK), 31  
   project, 361–365  
   project organization, 143–146

- Server Explorer tool, 346, 362–364
- Solution Explorer, 10
- tool for .NET development, 43
- transparent classes, 17
- Universal Description, Discovery, and Integration (UDDI), 223
- WSDL command line utility, 15
- WSDL proxy, 9
- XML Designer, 70–72
- XML validation, 79–80
- xmlns* attribute, 102
- XSD specifications, 76
- .vsdisco* files, 17
- VS.NET. *See* Visual Studio .NET

## W

- Web farms, 202
- Web Forms
  - adding to .aspx page, 11–12
  - ASP.NET central display mechanism, 55–56
  - ASP.NET primary user interfaces, 60
  - building Internet applications, 30
  - drill-down functionality, 107
  - getting read-only data, 57
  - limited by browsers, 60
  - Visual Studio default, 55
- Web gardens, 202
- Web Services
  - ADOCatalog, 278–284
  - architecture, defining, 342–345
  - calling, 152
  - COM components, 26–27
  - complex data types, 159–162
  - creating in VS.NET, 3–5
  - creating with SOAP, 139–145
  - customer administration, 275–278
  - definition, 136

- deploying, 159
- description, 3–7
- error handling, 162–167
- Extensible Markup Language (XML), 16
- implementing, 361
- .NET framework, 55, 61
- rationale, 139
- running, 146–147
- Service1 default Web Service, 362
- shopping-cart example, 250–263
- standards, 210–211
- System.Web.Services* namespace, 17–19
- testing, 147–159, 259–263
- Web Services Description Language (WSDL), 210–217
- XMLCart, 284–287
- See also* SQLXML Web Services
- Web Services Description Language (WSDL)
  - command line utility, 15, 21
  - customizing auto-generated code, 15
  - description, 15, 210–217
  - generating Web Service descriptions, 212
  - Internet user-page request process, 9
  - proxy, 9, 297
  - specification Web site, 212
- Web Services Toolkit, 301, 305
- Web site
  - administration, 266–275
  - AirportWeather, 225
  - designing, 264–266
  - Extensible Stylesheet Language Transformations (XSLT), 292
  - interoperability, 208
  - Microsoft Data Access Components (MDAC), 33

- Simple Object Access Protocol (SOAP), 137
  - SOAP Digital Signature, 203
  - SOAP implementations, 207
  - SOAP Messages with Attachments, 186
  - SOAPBuilders Interoperability Lab, 208
  - software development kit (SDK), 34
  - Uniform Resource Identifiers (URIs), 150
  - Universal Description, Discovery, and Integration (UDDI), 219–220
  - WSDL specification, 212
  - XML-RPC, 138
  - XSLT style sheet, 422
  - Well-formed XML documents, 75–76
  - Windows 95/98, 32
  - Windows 2000, 174, 203, 300
  - Windows Forms, 60–61, 176, 424
  - Windows Installer, 41
  - Windows Integrated Authentication, 306
  - Windows NT, 34
  - Windows Scripting Host, 156
  - Winer, David, 138
  - Wiring protocol, 139
  - Wrapping stored procedure calls, 393–399
  - writeEventLogEntry()* method, 366
  - Writing XML documents, 82–83
  - Wrong argument types, 165
  - WSDL. *See* Web Services Description Language
- X**
- XCOPY, 41
  - XMethods, 208
  - XML. *See* Extensible Markup Language
  - XML Designer, 70–72, 71–72
  - XML-Journal*, 203
  - XML-RPC, 138
  - XML Schema Definition (XSD), 16, 76
  - XmlAttributeAttribute* class, 179
  - XMLCart, 278, 284–287
  - XmlDataDocument* class
    - description, 98–99
    - loading, 99–100
    - multiple tables, 103–107
    - relational view, 100–103
  - XmlDocument* class, 94–98
  - XmlElementAttribute* class, 179
  - XmlNode* class, 20–21, 83–84
  - XmlNodeReader* class, 82
  - xmlns* attribute, 102
  - XmlTextReader* class, 84–89
  - XmlTextWriter* class, 90–93
  - XPath, 134
  - XPathDocument* class
    - document navigation, 112–115
    - querying XML data, 107–110
  - XPathNavigator* class
    - document navigation, 112–115
    - purpose, 107
    - using, 110–111
  - XSD. *See* XML Schema Definition
  - XSL Patterns, 134
  - XSLT. *See* Extensible Stylesheet Language Transformations
- Y**
- Yes/no data type, 240







# SYNGRESS SOLUTIONS...



AVAILABLE NOW!  
ORDER at  
[www.syngress.com](http://www.syngress.com)

## **.NET Developer's Kit, Including ASP, C#, and Visual Basic**

This 3-book box set will help developers build solutions for the .NET platform. The set includes: *ASP .NET Web Developer's Guide*, *C# .NET Web Developer's Guide*, and *VB .NET Developer's Guide*.

ISBN: 1-928994-61-X

Price: \$119.95 USA, \$185.95 CAN

AVAILABLE NOW!  
ORDER at  
[www.syngress.com](http://www.syngress.com)

## **XML .NET Developer's Guide**

XML is one of the cornerstones of the .NET Framework. .NET aims to bridge the gap between desktop applications and online applications, and facilitate the communication of objects between the two. *XML .NET Developer's Guide* will show you how to develop XML documents and applications for use within the .NET Framework.

ISBN: 1-928994-47-4

Price: \$49.95 USA, \$77.95, CAN



AVAILABLE JULY 2002!  
ORDER at  
[www.syngress.com](http://www.syngress.com)

## **Hack Proofing XML**

*Hack Proofing XML* will allow Web developers and database administrators to take advantage of the limitless possibilities of XML without sacrificing the integrity, confidentiality, and security of their information. Readers will be given hands-on instruction on how to encrypt and authenticate their XML data using prescribed standards, digital signatures, and various vendors' software.

ISBN: 1-931836-50-7

Price: \$49.95 USA, \$77.95 CAN

[solutions@syngress.com](mailto:solutions@syngress.com)

SYNGRESS®

