SYNGRESS®

1 YEAR UPGRADE
BUYER PROTECTION PLAN

XML .NET Developer's Guide

Develop and Deliver Enterprise-Critical Applications with XML .NET

- Complete Case Studies with Ready-to-Run Source Code and Full Explanations
- Hundreds of Developing & Deploying, and Debugging Sidebars, Security Alerts, and FAQs
- Complete Coverage of Web Services and the VS.NET Integrated Development Environment (IDE)

Adam Sills
Mesbah Ahmed
Dotthatcom.com
Frank Boumphrey
Jonothon Ortiz Technical Editor

solutions@syngress.com

With more than 1,500,000 copies of our MCSE, MCSD, 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**, 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

SYNGRESS®

SYNGRESS®

1 YEAR UPGRADE
BUYER PROTECTION PLAN

XML .NET Developer's Guide

Adam Sills
Mesbah Ahmed
Dotthatcom.com
Frank Boumphrey
Jonothon Ortiz Technical Editor

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	PKH4T67VT5
002	ESTRT45RF4
003	BHER6W354N
004	9HD34B3QAN
005	ZR88JN6NVH
006	NTG4R54RM4
007	CG8VHTR46T
008	D6Y9R565MR
009	22N5M4BX6S
010	SD6YH2Y7FC

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

XML .NET Developer's Guide

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: 1-928994-47-4

Technical Editor: Jonothon Ortiz
Acquisitions Editor: Catherine B. Nolan
Indexer: Robert Saigh
Copy Editor: Beth A. Roberts
Cover Designer: Michael Kavish
Page Layout and Art by: Reuben Kantor and Shannon Tozier

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, 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, Bill Getz, Jon Mayes, John Mesjak, Peg O'Donnell, Sandra Patterson, Betty Redmond, Roy Remer, Ron Shapiro, Patricia Kelly, Andrea Tetrick, Jennifer Pascal, Doug Reil, and David Dahl 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.

Annabel Dent and Paul Barry of Elsevier Science/Harcourt Australia for all their help.

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.

Ethan Atkin at Cranbury International for his help in expanding the Syngress program.

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.

Contributors

Adam Sills is an Internet Programmer at GreatLand Insurance, a small insurance company parented by Kemper Insurance. He works in a small IT department that focuses on creating applications to expedite business processes and manage data from a multitude of locations. Previously, he had a small stint in consulting and also worked at a leading B2B eCommerce company designing and building user interfaces to interact with a large-scale enterprise eCommerce application. Adam's current duties include building and maintaining Web applications, as well as helping to architect, build, and deploy new Microsoft .NET technologies into production use. Adam has contributed to the writing of a number of books for Syngress including *ASP .NET Developer's Guide* (ISBN: 1-928994-51-2) and is an active member of a handful of ASP and ASP.NET mailing lists, providing support and insight whenever he can.

Todd Carrico (MCDBA, MCSE) is a Senior Database Engineer for Match.com. Match.com is a singles portal for the digital age. In addition to its primary Web site, Match.com provides back-end services to AOL, MSN, and many other Web sites in its affiliate program. Todd specializes in design and development of high-performance, high-availability data architectures primarily on the Microsoft technology. His background includes designing, developing, consulting, and project management for companies such as Fujitsu, Accenture, International Paper, and GroceryWorks.com. In addition to his contribution to *C# .NET Web Developer's Guide* (ISBN: 1-928994-50-4), Todd has also contributed chapters to other books in the Syngress .NET Series including the *ASP .NET Web Developer's Guide (*ISBN: 1-928994-51-2) and the *VB .NET Developer's Guide* (ISBN: 1-928994-48-2). Todd resides in Sachse, Texas with his wife and two children.

Greg Hack is a Senior Software Engineer with Allscripts Healthcare Solutions. Greg has over 15 years of experience developing software on platforms ranging from the mainframe to the desktop, using a wide

variety of languages and technologies. Recent work includes a Web-based application that allows patients to view their medical records and a Pocket PC application that delivers clinical information to physicians at the point of care. Greg has also contributed to *C# .NET Web Developer's Guide* (ISBN: 1-928994-50-4).

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 of science degree from the University of Washington, Bothell. He lives in Puyallup, Washington with his wife, Angela. Patrick is a contributor to Syngress Publishing's *C# .NET Web Developer's Guide* (ISBN: 1-928994-50-4) and the *ASP .NET Web Developer's Guide (*ISBN: 1-928994-51-2).

David Jorgensen (MCP) is an instructor at North Seattle Community College, University of Washington Extension campus, and Puget Sound Centers. He is also developing courses for Seattle Vocational Institute, which teach .NET and Web development to the underprivileged in the Seattle area. David also provides internship opportunities through his company, DotThatCom.com, which does online sample classes and chapters of books. David holds a bachelor's degree in Computer Science from St. Martin's College and resides in Puyallup, Washington, 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 *ASP .NET Web Developer's Guide (*ISBN: 1-928994-51-2).

Joe Dulay (MCSD) is the Vice-President of Technology for the IT Age Corporation. IT Age Corporation is a project management and software development firm specializing in customer-oriented business enterprise and e-commerce solutions located in Atlanta, Georgia. His current

responsibilities include managing the IT department, heading the technology steering committee, software architecture, e-commerce product management, and refining development processes and methodologies. Though most of his responsibilities lay in the role of manager and architect, he is still an active participant of the research and development team. Joe holds a bachelor's degree from the University of Wisconsin in Computer Science. His background includes positions as a Senior Developer at Siemens Energy and Automation, and as an independent contractor specializing in e-commerce development. Joe is also co-author of Syngress Publishing's *Hack Proofing Your Web Applications* (ISBN: 1-928994-31-8). Joe would like to thank his family for always being there to help him.

Henk-Evert Sonder (CCNA) has over 15 years of experience as an Information and Communication Technologies (ICT) professional, building and maintaining ICT infrastructures. In recent years, he has specialized in integrating ICT infrastructures with secure business applications. Henk's company, IT Selective, works with small businesses to help them develop high-quality, low cost solutions. Henk has contributed to several Syngress books, including the *E-Mail Virus Protection Handbook* (ISBN: 1-928994-23-7), *Designing SQL Server 2000 Databases for .NET Enterprise Servers* (ISBN: 1-928994-19-9), *VB .NET Developer's Guide* (ISBN: 1-928994-48-2), and *BizTalk Server 2000 Developers Guide for .NET* (ISBN: 1-928994-40-7). Henk lives in Hingham, Massachusetts with his wife, Jude and daughter, Lilly.

Chris Garrett is the Technical Manager for a large European Web agency. He has been working with Internet technologies since 1994 and has provided technical and new media expertise for some of the world's biggest brands. Chris is a co-author of Syngress Publishing's *ASP .NET Web Developer's Guide (*ISBN: 1-928994-51-2). Chris lives in Yorkshire, England with his wife, Clare and his daughter, Amy.

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 also a co-author of Syngress Publishing's *ASP .NET Web Developer's Guide (*ISBN: 1-928994-51-2).

Dreamtech Software India, Inc., is a leading provider of corporate software solutions. Based in New Delhi, the company is a successful pioneer of innovative solutions in e-learning technologies. The Dreamtech Software team, which authored all the books in the *Cracking the Code* series has over 50 years of combined software-engineering experience in areas such as Java, wireless application, XML, voice-based solutions, .NET, COM/COM+ technologies, distributed computing, DirectX, Windows Media technologies, and security solutions. For more information, log on to www.dreamtechsoftware.com.

Frank Boumphrey is a retired professor of surgery who now specializes in Internet applications and medical documentation. As well as numerous medical papers, he has authored several books on XML, the Internet and on other related subjects. Frank is the president of the HTML Writers Guild, a 125,000 member strong, not-for-profit, International organization of Web page Writers, and was a participant in various working groups of the World Wide Web Consortium (W3C). Presently his main objective is to help XML to become the language of choice in Web documents.

Technical Editor and Reviewer

Jonothon Ortiz is Vice President of Xnext, Inc. in Winter Haven, Florida. Xnext, Inc. is a small, privately owned company that develops Web sites and applications for prestigious companies, such as the New York Times. Jonothon is the head of the programming department and works together with the CEO on all company projects to ensure the best possible solution. His primary field of experience is database backend for Web applications and occasionally programming the GUI of a Web application. He has developed over 30 databases, ranging from small e-commerce sites to client identification and storage. Many of these databases incorporated XML in some fashion, from a small footprint file to the generation of smaller XML files to increase performance time for often-used queries and results. The majority of these applications were coded in either PHP, Perl, or ASP 3.x / .NET. Johothon has been a contributor to a variety of title from Syngress Publishing, including *ASP .NET Web Developer's Guide (*ISBN: 1-928994-51-2), the *VB .NET Developer's Guide* (ISBN: 1-928994-48-2), and the *Ruby Developer's Guide* (ISBN: 1-928994-64-4). Jonothon lives with his wife, Carla in Lakeland, Florida.

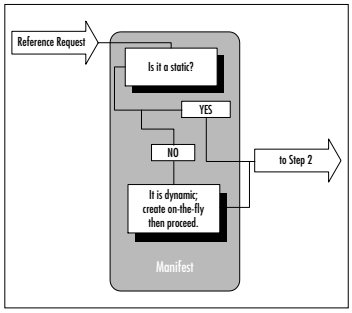
Contents

Foreword **xxi**

Chapter 1 Introducing the Microsoft .NET Framework **1**

Introduction 2
What Is the .NET Framework? 3
Introduction to the Common Language Runtime 3
Using .NET-Compliant Programming Languages 5
Creating Assemblies 5
Using the Manifest 8
Assembly Cache 11
Locating an Assembly 12
Private Assembly Files 17
Shared Assembly Files 17
Understanding Metadata 17
The Benefits of Metadata 18
Identifying an Assembly with Metadata 18
Types 19
Defining Members 19
Using Contracts 20
Assembly Dependencies 21
Unmanaged Assembly Code 21
Reflection 21
Attributes 22
Ending DLL Hell 22
Side-by-Side Deployment 23
Versioning Support 23
Using System Services 24
Exception Handling 24

Step 1 of the Location Process

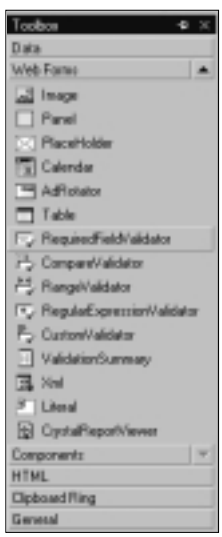


StackTrace 25
InnerException 26
Message 26
HelpLink 26
Garbage Collection 26
Console I/O 26
Microsoft Intermediate Language 27
The Just-in-Time Compiler 27
Using the Namespace System to Organize Classes 28
The Common Type System 29
Type Safety 32
Relying on Automatic Resource Management 32
The Managed Heap 33
Garbage Collection and the Managed Heap 35
Assigning Generations 40
Using Weak References 41
Security Services 41
Framework Security 43
Granting Permissions 43
Gaining Representation through a Principal 45
Security Policy 46
Summary 48
Solutions Fast Track 49
Frequently Asked Questions 52

Chapter 2 Visual Studio.NET IDE 55

Introduction 56
Introducing Visual Studio.NET 56
Components of VS.NET 58
Design Window 59
Code Window 59
Server Explorer 60
Toolbox 61
Docking Windows 62
Properties Explorer 63
Solution Explorer 64

The Toolbox Window



Class View 65
Dynamic Help 66
Task List Explorer 67
Features of VS.NET 68
IntelliSense 68
XML Editor 70
Documentation Generation (XML Embedded Commenting) 73
Adding XML Document Comments to C# Pages 74
Customizing the IDE 75
Creating a Project 76
Projects 76
Creating a Project 76
Add Reference 77
Build the Project 77
Debugging a Project 77
Summary 78
Solutions Fast Track 79
Frequently Asked Questions 81

Chapter 3 Reviewing the Fundamentals of XML 83

Introduction 84
An Overview of XML 84
The Goals of XML 85
What Does an XML Document Look Like? 85
Creating an XML Document 86
Creating an XML Document in VS.NET XML Designer 87
Components of an XML Document 88
Structure of an XML Document 91
Well-Formed XML Documents 92
Schema and Valid XML Documents 93
XML Schema Data Types 97
Transforming XML through XSLT 98
XSL Use of Patterns 102

The Goals of XML

- XML shall be compatible with SGML.
- It shall be easy to write programs that process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum; ideally, zero.
- XML documents should be human-legible and reasonably clear.
- The XML design should be prepared quickly.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.
- Terseness in XML markup is of minimal importance.
- XML shall be straightforwardly usable over the Internet.
- XML shall support a variety of applications.

XPath 105
Summary 107
Solutions Fast Track 107
Frequently Asked Questions 109

Chapter 4 Using XML in the .NET Framework 111

Introduction 112
Explaining the XML Document Object Model 112
The Different XML DOM Levels 113
XML DOM Core Interfaces 114
DOM Structure Model 115
DOM Traversal 118
NodeIterator 119
TreeWalker 120
NodeFilter 121
DOM Range 122
DOM XPath 123
Introduction to the *System.Xml* Namespace 124
Overview of *System.Xml.Schema* Classes 124
Mapping XML DOM on the *System.Xml* Namespace 130
Explaining a Selection of *System.Xml* Classes 132
Using the *System.Xml* Namespace 145
Building the XML Address Book 145
Loading the XML Address Book 145
Creating and Deleting Categories 149
Creating, Editing, and Deleting Entries 151
Summary 156
Solutions Fast Track 157
Frequently Asked Questions 158

Chapter 5 Understanding .NET and XML Security 159

Introduction 160
The Risks Associated with Using XML in the .NET Framework 160

The Document Interface Attributes

Attribute	Description
doctype	The Document Type Declaration (DTD) with this document. If DTD is not present, this returns null.
documentElement	The root element in the document.

Confidentiality Concerns 161
.NET Internal Security as a Viable Alternative 162
Permissions 163
Principal 164
Authentication 165
Authorization 165
Security Policy 165
Type Safety 165
Code Access Security 166
.NET Code Access Security Model 166
Stack Walking 167
Code Identity 168
Code Groups 169
Declarative and Imperative Security 172
Requesting Permissions 173
Demanding Permissions 177
Overriding Security Checks 179
Custom Permissions 184
Role-Based Security 185
Principals 186
WindowsPrincipal 186
GenericPrincipal 187
Manipulating Identity 188
Role-Based Security Checks 190
Security Policies 192
Creating a New Permission Set 195
Modifying the Code Group Structure 200
Remoting Security 207
Cryptography 207
Security Tools 210
Securing XML—Best Practices 212
XML Encryption 212
XML Digital Signatures 218
Summary 222
Solutions Fast Track 223
Frequently Asked Questions 228

You can determine the permission set of a code group by performing these steps:

1. Run Microsoft Management Console (MMC) by choosing **Start | Run** and typing **mmc**.
2. Open the .NET Management snap-in, via **Console | Add/Remove Snap-in**.
3. Expand the **Console Root | .NET Configuration | My Computer**.
4. Expand **Runtime Security Policy | Enterprise |Code Groups**.
5. Select the code group **All_Code**.
6. Right-click **All_Code** and select **Properties**.
7. Select the **Permission Set** tab.
8. The **Permission Set** field lists the current value.

Answers to Your Frequently asked Questions

Q: Why so much emphasis on the Web? Can't I use XML on the desktop as well?

A: Yes, you can use XML on the desktop. However, one of the main goals of .NET is to properly connect the desktop with the Internet and not suffer any setback due to server type, programming language, and so on. As you might have noticed as well, ASP.NET can be thought of as a Web wrapper for desktop code. This helps ensure that what you see online will be mostly reproducible offline.

Chapter 6 XML and the Web with ASP.NET 231

Introduction 232
Reviewing the Basics of the ASP.NET Platform 232
Reading and Parsing XML 234
Parsing an XML Document 235
Navigating through an XML Document to Retrieve Data 236
Writing an XML Document Using the *XmlTextWriter* Class 239
Generating an XML Document Using *XmlTextWriter* 239
Exploring the XML Document Object Model 242
Navigating through an *XmlDocument* Object 243
Parsing an XML Document Using the *XmlDocument* Object 244
Using the *XmlDataDocument* Class 247
Loading an *XmlDocument* and Retrieving the Values of Certain Nodes 248
Using the Relational View of an *XmlDataDocument* Object 249
Viewing Multiple Tables of an *XmlDataDocument* Object 252
Querying XML Data Using *XPathDocument* and *XPathNavigator* 256
Using *XPathDocument* and *XPathNavigator* Objects 259
Using *XPathDocument* and *XPathNavigator* Objects for Document Navigation 261
Transforming an XML Document Using XSLT 264
Transforming an XML Document to an HTML Document 266
Transforming an XML Document into Another XML Document 268
Working with XML and Databases Online 274

Creating an XML Document from a Database Query 275
Reading an XML Document into a DataSet 278
Summary 280
Solutions Fast Track 280
Frequently Asked Questions 283

Chapter 7 Creating an XML.NET Guestbook 283
Introduction 284
Functional Design Requirements of the XML Guestbook 285
Constructing the XML 286
Adding Records to the Guestbook 288
Understanding the *pnlAdd* Panel 292
Adding a Thank-You Panel with *PnlThank* 294
Exploring the Submit Button Handler Code 294
Viewing the Guestbook 298
Displaying Messages 298
Advanced Options for the Guestbook Interface 301
Manipulating Colors and Images 301
Modifying the Page Output 305
Summary 308
Solutions Fast Track 308
Frequently Asked Questions 310

Chapter 8 Creating a Message Board with ADO and XML 311
Introduction 312
Setting Up the Database 312
MS Access Database 313
SQL Server Database 317
Designing Your Application 321
Designing Your Objects 323
Creating Your *Data Access* Object 323
Designing the *User* Class 325
Designing the *Board* Class 335

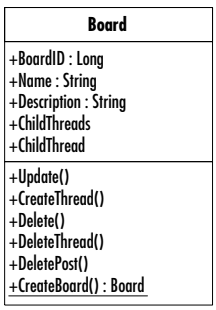
Migrating…

Online Forms

As you have noticed and learned throughout this book, ASP.NET enables programmers to use Web forms, which can be described as the VB6.0 desktop form. In this particular example, your "AddClick" sub would be placed within the *OnClick()* event for whatever button you wanted to use as your trigger for this action. One other little trick is to view each "panel" as a small form within an mdi, namely the browser window, with their own "hide" and "show" features.

Designing the *ThreadList* Class 344
Designing the *Thread* Class 347
Designing the *PostList* Class 350
Designing the *Post* Class 353
Designing the *MessageBoard* Class 356
Designing the User Interface 357
Setting Up General Functions 358
Building the Log-In Interface 366
Designing the Browsing Interface 373
Board Browsing 373
Thread Browsing 376
Message Browsing 379
Creating the User Functions 382
Editing the Member Profile 383
Creating Threads and Posts 385
Building the Administrative Interface 389
Summary 403
Solutions Fast Track 403
Frequently Asked Questions 405

The Board Class



Chapter 9 Building a Remote Database Viewer 407
Introduction 408
Understanding ADO.NET 408
The ADO.NET Architecture 410
Using .NET Data Provider 410
Connection 411
Command 411
Data Reader 411
Data Adapter 412
Using *DataSets* and *DataTables* 413
A Quick Comparison of ADO and ADO.NET 414
Accessing Data from a Database Using ADO.NET 414
Database Design 415
Navigating between Records 415
Add Record Form 419
Delete/Update Form 422

Understanding ADO.NET

The ADO.NET also sports these features:

- Interoperability
- DataSet
- Performance
- Scalability
- Maintainability

Converting Binary Data Using Base64 428
How Base64 Works 429
Converting Binary Data into Base64 Format 431
Database Design 431
Reading Base64 Encoded Data from an XML File 436
Designing and Implementing a Simple Remote Database Viewer 440
What Is a Remote Database? 441
Advantages and Disadvantages of Remote Data Access 442
Implementing a Simple Remote Database Viewer 445
Summary 448
Solutions Fast Track 448
Frequently Asked Questions 449

Chapter 10 Building a Wholesale Catalog 451

Introduction 452
Basic Design Considerations 453
Storage: XML versus Traditional Databases 453
Information Transport Methods 454
XML and EDI 455
XML Vocabularies 456
Implementation of the Agora Markets Catalog 456
Data Store 456
Transport Protocols 457
Vocabularies 457
Requirements 457
Analysis 458
Data Store 458
Catalog Updating 459
Business-to-Business E-Communications 459
The XML Files 460
Data Typing Entries 461
Catalog 462

Developing & Deploying…

DataReader versus DataSet

DataReader for the most part works pretty much like the old recordset with which ASP programmers are familiar. DataSet will create a virtual database (preserved in XML) that we can work with even while disconnected to the database. It requires a complete new subset of objects and methods to work with it.

Coding the Project 462
Database Design 463
OLTP versus OLAP 463
XML Packages Design 465
Supplier Interface and B2B Design 469
Fatal Errors versus Nonfatal Errors 469
Coding updatecat1.aspx 470
Analysis of Code Listing updatecat1.aspx 482
Customer Interface Design 490
GUI: The Catalog Page 490
Analysis of Code 493
Analysis of Code 498
GUI: The Shopping Cart Page(s) 502
Analysis of Code Listing 'shopcartadd.aspx' 508
Business and Web Services 514
Business versus Web Services 514
Coding a Business Service 514
Analysis of Code 516
Creating a Web Service 519
An Overview of Web Services 519
Coding a Web Service 519
Analysis of Code 521
Testing the Web Service 522
Using Web Services 524
Universal Description, Discovery, and Integration 524
Web Service Description Language 524
Installation: Migrating to SQL Server 529
Changing the Connection String 530
Compatible Data Types 530
SQL Strings 531
Converting to SQL Server 531
Summary 533
Solutions Fast Track 533
Frequently Asked Questions 535

Index 537

Foreword

Welcome to the *XML.NET Developer's Guide*! We have taken great care to create a quality reference book for XML programmers who want to enhance their coding skills to include applications for the .NET platform. This book assumes that you do have previous exposure to XML and are familiar with VB.NET, C#, and ASP.NET. In other words, this book is not for a novice or beginner.

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 the potential of what XML could be—a way to provide data between parties without needing to rely on proprietary solutions.

Developers began to incorporate snippets of XML into their desktop applications, maybe to store configuration data or maybe as an export file. As time passed, they began to transfer XML to the Internet. Databases began to communicate to each other via XML and companies were finding that they had an easier time coping with external database data thanks to XML.

Developers, however, were not the only ones to notice the potential of XML: Microsoft 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.

The XML .NET Developer's Guide was created and organized using the following principal: XML, in the real world, lives up to its flexibility. You are just as likely to stumble across a desktop application running XML as you are to find an online e-commerce shop that uses XML to transfer data.

As you work through this book you'll find that we will be jumping around from VB.NET to C# or maybe use a little bit of both. This flexibility within .NET allows you to use the right code always to optimize your XML code; if you thought C# provides faster queries than VB.NET but VB.NET delivers the better front-end you'll find; it's not an issue, as within .NET you can use both.

If this sounds confusing to you it may mean that you are still a beginner with .NET in general; if this is the case we suggest you pick up a copy of *VB.NET Developer's Guide* (ISBN: 1-928994-48-2) and *C# Web Developer's Guide* (ISBN: 1-928994-50-4) from Syngress Publishing. These books contain greater detail on the .NET framework including how it works, and how to work with it using the programming language of your choice. As new .NET languages become available Syngress' set of .NET programming books will increase and so will your choices in programming for XML.

If you have read any of Syngress' .NET books in the past you'll be familiar with the layout. We introduce either introductory material (or, in this case, refresher material) in the first couple of chapters, move on to the meat of the book with in-depth views on specific points in the programming language, and finish off with a set of case studies that enhance the skills and ideas you've learned throughout the book. All in all, the *XML.NET Developer's Guide* has a total of ten chapters.

Chapter 1 (*Introducing the .NET Framework*) will bring you up to speed with a refresher in how .NET works internally, with Chapter 2 (*Visual Studio .NET IDS*) providing an introductory look into VS.NET, Microsoft's IDE for .NET programming. This new IDE can work more with XML than its predecessor, so even if you are familiar with the VS.NET IDE it may be a good idea to browse through this chapter.

Chapter 3 (*Reviewing the Fundamentals of XML*) kicks off by giving you a quick refresher for XML basics. This is followed up by Chapter 4 (*Using XML in the .NET Framework*), in which we start to look at how you can work with XML through .NET. Both Chapter 3 and 4 cover basic XML items such as proper XML syntax and validation through Schemas. You will also learn about many of the major namespaces and how they work, and begin to familiarize yourself with the appropriate classes you need to complete your projects. You will be surprised at the flexibility that XML offers and how even some other classes that do not directly revolve around XML can work with XML as well.

Chapter 5 (*Understanding .NET and XML Security*) introduces a major issue in the XML user community—security. While XML is unable to provide proper security by

itself, a thorough understanding of what XML can do combined with an understanding of .NET security is vital.

Chapter 6 (*Web Development Using XML and ASP.NET*) will introduce you to the online aspect of XML using ASP .NET and teach you how XML is a vital part of online applications through the use of multiple examples, including an online catalog.

Chapters 7, 8, 9, and 10 are the hands-on case studies (*Creating an XML.NET Guestbook*; *Creating a Message Board with ADO and XML*; *Building a Simple Remote Database Viewer*; and *Building a Wholesale Catalog*). These applications, with the exception of Chapter 7, are fairly large, complex, and require an understanding of basic .NET concepts as well as .NET programming.

You wanted XML? You got it!

—*Jonothon Ortiz, Technical Editor*

Chapter 1

Introducing the Microsoft .NET Framework

Solutions in this chapter:

- What Is the .NET Framework?
- Introduction to the Common Language Runtime
- Using .NET-Compliant Programming Languages
- Creating Assemblies
- Understanding Metadata
- Using System Services
- Microsoft Intermediate Language
- Using the Namespace System to Organize Classes
- The Common Type System
- Relying on Auto Resource Management
- Security Services

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

With the introduction of the .NET architecture, Microsoft presents a solution that must provide us with a solid base for distributed applications. Although Microsoft has a long way to go, they are off to a good start with the .NET Framework. Visual Studio .NET is the first real .NET application that will see daylight.

To leverage the communication of distributed .NET applications, the .NET architecture makes heavy use of XML. In fact, XML is the default encoding language of the framework, not only to encapsulate data to send back and forth between applications, but also in configuration files. This is a logical choice as you learn what the .NET Framework is all about.

This chapter will cover all the basics of this framework that you need to understand to use the information in the rest of this book. If you need more in-depth information on the .NET framework, you can find it in the other .NET books in this series.

The .NET framework includes a number of *base classes* to get you started. The Framework includes abstract base classes to inherit from, as well as implementations of these classes to use. You can even derive your own classes for custom modifications. All the classes are derived from the *system object*, which gives you great power and flexibility. All applications will share a common runtime environment called the Common Language Runtime (CLR). The .NET Framework now includes a Common Type System (CTS) that allows all the languages to share data using the same types. These features facilitate cross-language interoperability.

To use .NET, you need to learn some new concepts, which we discuss throughout this chapter. A .NET application is wrapped up in an *assembly*. An assembly includes all the information you need about your application. It includes information that you would find currently in a type library, as well as information you need to use the application or component. This makes your application or component completely self-describing. When you compile your application, it is compiled to an intermediate language called the Microsoft Intermediate Language (MSIL). When a program is executed, it is then converted to machine code by the CLR's just-in-time (JIT) compiler. The MSIL allows an application to run on any platform that supports the CLR without changing your development code.

Once the code has been prepared, .NET's work is still not done. It continues to monitor the application and perform *automatic resource management* on the

application to clear up any unused memory resources and provide security measures to prevent anyone from accessing your assembly.

In these few paragraphs, we've introduced the major new concepts found within .NET: the CLR, the assembly unit (and its contents), what makes .NET interoperable, and how .NET is "smart" in terms of automatic memory management and security. Let's now look in-depth at how .NET works so we can get a better grasp of what it can do for our XML applications, both desktop and online.

What Is the .NET Framework?

The .NET Framework is Microsoft's latest offering in the world of cross-development (developing both desktop and Web applications), interoperability, and, soon, cross-platform development. As you go through this chapter, you'll see just how .NET meets these developmental requirements. However, Microsoft's developers did not stop there; they wanted to completely revamp the way we program.

In addition to the more technical changes, .NET strives to be as simple as possible. .NET contains functionality that a developer can easily access. This same functionality operates within the confines of standardized data types and naming conventions. This internal functionality also encompasses the creation of special data within an assembly file that is vital for interoperability, .NET's built-in security, and .NET's automatic resource management.

Another part of the "keep it simple" philosophy is that .NET applications are geared to be copy-only installations; in other words, a special installation package for your application is no longer required. The majority of .NET applications work if you simply copy them into a directory, which definitely eases the burden on the programmer.

The CLR changes the way in which programs are written, in the sense that developers won't be limited to the Windows platform. Just as with ISO C/C++, programmers are now able to see their programs work on any platform with the .NET runtime installed.

Introduction to the Common Language Runtime

The CLR controls the .NET code execution. CLR is the step above COM, MTS, and COM+.

The CLR is the runtime environment for .NET. It manages code execution and the services that .NET provides. The CLR "knows" what to do through

special data (referred to as *metadata*) that is contained within the applications. The special data within the applications store a map of where to find classes, when to load classes, and when to set up runtime context boundaries, generate native code, enforce security, determine which classes use which methods, and load classes when needed. Since the CLR is privy to this information, it can also determine when an object is used and when it is released. This is known as *managed code*.

Managed code is what we want to aim for in order to create fully CLR-compliant code. Code that's compiled with COM and Win32API declarations falls under the category of *unmanaged code*. Managed code keeps us from depending on obstinate dynamic link library (DLL) files. In fact, thanks to the CLR, we don't have to deal with the Registry, graphical user identifications (GUIDs), AddRef, HRESULTS, and all the macros and application programming interfaces (APIs) we depended on in the past. They aren't even an available option in .NET.

Removing all the excess also provides a more consistent programming model. Since the CLR encapsulates all the functions that we had with unmanaged code, we won't have to depend on any preexisting DLL files residing on the hard drive. This does not mean that we have seen the last of DLL; it simply means that the .NET Framework contains a system within it that can "map out" the location of all the resources we are using.

To help CLR-based code execute properly, CLR-compliant code is also Common Language Specification (CLS)-compliant code. CLS is a subset of CLR types defined in the Common Type System (also discussed later in the chapter), and its features are instrumental in the interoperability process by containing the basic types required for CLR operability. These little things put together allow .NET to handle multiple programming languages. The CLR manages the mapping; all that you need is a compiler that can generate the code and the special data needed within the application for the CLR to operate. This ensures that any dependencies your application might have are always met and not broken.

When you set your compiler to generate the .NET code, it runs through the CTS and inserts the appropriate data within the application for the CLR to read. Once the CLR finds the data, it proceeds to run through it and lay out everything it needs within memory, declaring any objects when they are called (but not before). Any application interaction, such as passing values from classes, is also mapped within the special data and handled by the CLR.

Using .NET-Compliant Programming Languages

.NET isn't stuck in the rut of a single, solitary programming language taking advantage of a multiplatform system. In fact, when you get right down to it, what's the point of having a runtime that promises portability when you have to use a singular programming model to do it? You have to rely on just that one programming language to fill your needs, but what happens if the language doesn't lend itself to your needs? All of a sudden, portability takes a back seat to necessity—for something to be truly "portable," you require not only a portable runtime, but also the ability to code in *what* you need, *when* you need it. .NET offers us the solution of allowing any programming language that is compliant with .NET to run. Can't get that bug in your class worked out in VB, but you know that you can work around it in C? Use C# to create a class that can be easily used with your VB application. Third-party programming language users don't need to fret for long, either; several companies plan to create .NET-compliant versions of their languages.

Currently, the only .NET-compliant languages are all of the Microsoft flavor; for more information, check these out at http://msdn.microsoft.com/net:

- C#
- XML
- C++ with Managed Extensions
- VB.NET
- ASP.NET (although this one is more a subset of VB.NET)
- Jscript.NET

Other programming languages are preparing their .NET – compliant versions, such as Perl (http://aspn.activestate.com/ASPN/NET/index) and even Cobol (www.adtools.com/info/whitepaper/net.html).

Creating Assemblies

So, just how do you get a bunch of languages to "play nice" together? Most other programming languages do not use the Portable Executable (PE) format for their executables, which was a primary reason that prevented portability to

Microsoft, and vice versa. With the .NET environment comes something new: a logical approach to executables called *assemblies*. The CLR handles the entire *executing* of an assembly. The assembly "owns" a collection of files that are referred to as *static assemblies*, which the CLR uses. Static assemblies can be resources used by the assembly, such as image files or text files that the application will use. The actual code that executes is found within the assembly in MSIL format. In other words, an assembly is roughly the equivalent of a VB 6.0 COM component. An assembly has three options that need to be set when you create it:

- Loader optimization
- Naming
- Location

The *loader optimization* option has three settings: *single domain*, *multidomain*, and *multidomain host*. The single-domain setting is the default and is used most in client-side situations. The JIT code is generally smaller when the single-domain setting is used, compared with the other two settings, and there is no real noticeable difference between memory resources. The exception is if the application winds up being used as part of a multidomain or multidomain host setup, where it will actually hurt more than it'll help—such as within a client/server solution.

The multidomain and multidomain host settings apply to the same concept of multidomain usage. The only difference between the two is how the CLR will react with the code. In multidomain, the code is assumed to be the same across the domain. In multidomain host, however, each domain hosts different code. Let's say that you have an application development in which all the domains have the assembly filename, but each has different code hosted to see how it can still interact; you would get the best performance using the multidomain host optimization routine.

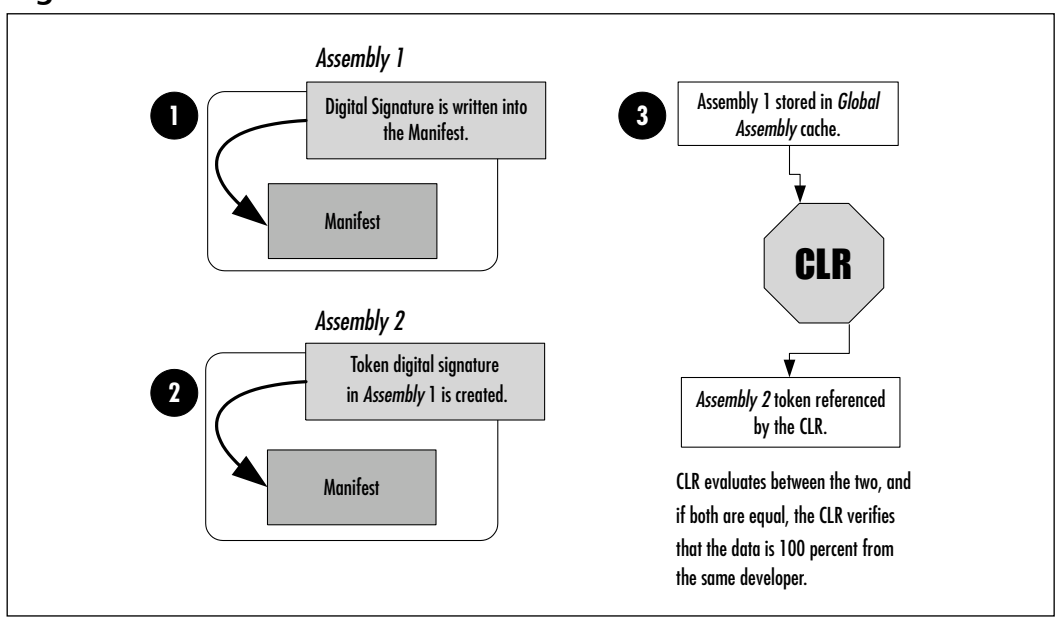
There are some benefits to setting the assembly as useable by multiple applications. Fewer resources will be consumed, since the type (object) will be loaded and mapped already; it won't need to be recreated each time it's needed. However, the end result of the JIT code is increased some, and access to static items are slower, since the static references are referenced indirectly.

The *name* of the assembly can impact the scope and usage by multiple applications. A single-client use application uses the name given to it when created, but there is no prevention for name collision. Therefore, in order to help prevent

name collisions in an assembly in a multi-assembly scenario, you can also give the assembly a *shared name*. Having a shared name means that the assembly can be deployed in the *global assembly cache*, which you can think of as a global repository of assemblies.

A shared name is made up of the textual name of the assembly (the name you created for it) and a digital signature. Shared names are unique names due to the pairing of the text name and digital signature. This system, in turn, helps prevent name collision and keeps anyone using the same textual name from writing over your file, since the shared name is different. A shared name also provides the required information that's needed for versioning support by the CLR; this same information is used to provide integrity checks to give a decent level of trust. (For full trust, you should include a full digital signature with certificates.) Figure 1.1 illustrates how the shared-name process works.

Figure 1.1 The Shared-Name Process



From the shared-name diagram in Figure 1.1, you can see that the shared name is first created into the primary assembly (*Assembly 1*), then the reference of the primary assembly is stored as a token of the version within the referencing assembly's (*Assembly 2*'s) metadata, and it is finally verified through the CLR.

An assembly, once created, has the following characteristics:

- **Contains code that the runtime executes** PE MSIL code is not executed without the manifest present. In other words, if the file is not formatted correctly, it will not run.
- **Only one entry point** An assembly cannot have more than one starting point for execution by the runtime; you cannot, for example, use both WinMain and Main.
- **Unit of side-by-side execution** An assembly provides the basic unit needed for side-by-side execution.
- **Type boundary** Each type declared within an assembly is recognized as a type of the assembly, not as a solitary type initiated into memory.
- **Security boundary** The assembly evaluates permission requests.
- **Basic deployment unit** An application comprised of assemblies requires only the assemblies that make up its core functions. Any other assemblies that are needed can be provided on demand, which keeps applications from having the bloated setup files commonly associated with VB 6.0 runtime files.
- **Reference scope boundary** The manifest within the assembly dictates what can and cannot go on, in order to resolve types and resources; it also enumerates assembly dependency.
- **Version boundary** Being the smallest versionable unit in the CLR, all the types and resources that it has are also versioned as a unit. The manifest describes any version dependencies.

Figure 1.2 displays a typical assembly. The assembly has been dissected to display the code, the manifest area, the metadata within the manifest, and the information stored within the metadata.

As you can see, all the benefits that CLR gives us are located within the assembly, but reside within the manifest.

Using the Manifest

Apart from the MSIL, an assembly contains metadata within its manifest. We will go into detail about metadata and its uses in upcoming sections, but for now, just remember that the metadata is all the relevant information that the CLR needs to properly run the file, and the manifest stores the metadata. Thanks to the manifest, assemblies are freed from depending on the Registry and breaking DLLs (the cause of DLL Hell). Basic metadata includes the items listed in Table 1.1.

Figure 1.2 A Typical Assembly

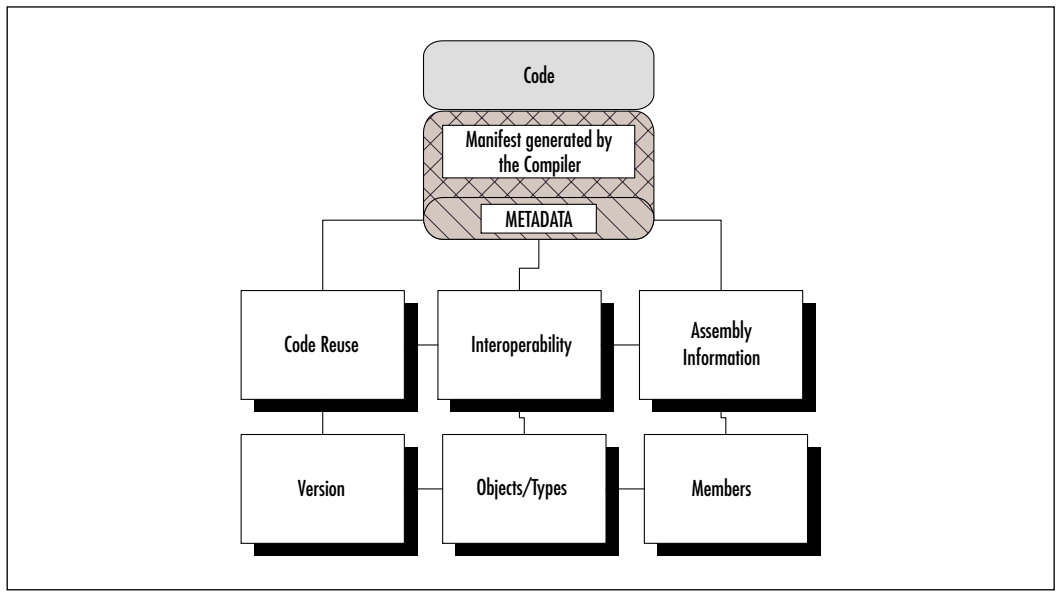


Table 1.1 Basic Attribute Classes

Basic Attribute Class	Description
AssemblyCompanyAttribute	Contains a string with the company name and product information.
AssemblyConfigurationAttribute	Contains current build information, as in Alpha stage.
AssemblyCopyrightAttribute	Copyright information that is stored as a string.
AssemblyDefaultAliasAttribute	Name information and alias information.
AssemblyDescriptionAttribute	Provides a description of the modules included within the assembly.
AssemblyInformational VersionAttribute	Any extra version information; this is not used by the CLR for versioning purposes.
AssemblyProductAttribute	Product information.
AssemblyTitleAttribute	Title of the assembly.
AssemblyTrademarkAttribute	Any trademarks of the assembly.

Table 1.2 lists custom attributes that you can set into the manifest.

Table 1.2 Custom Attributes

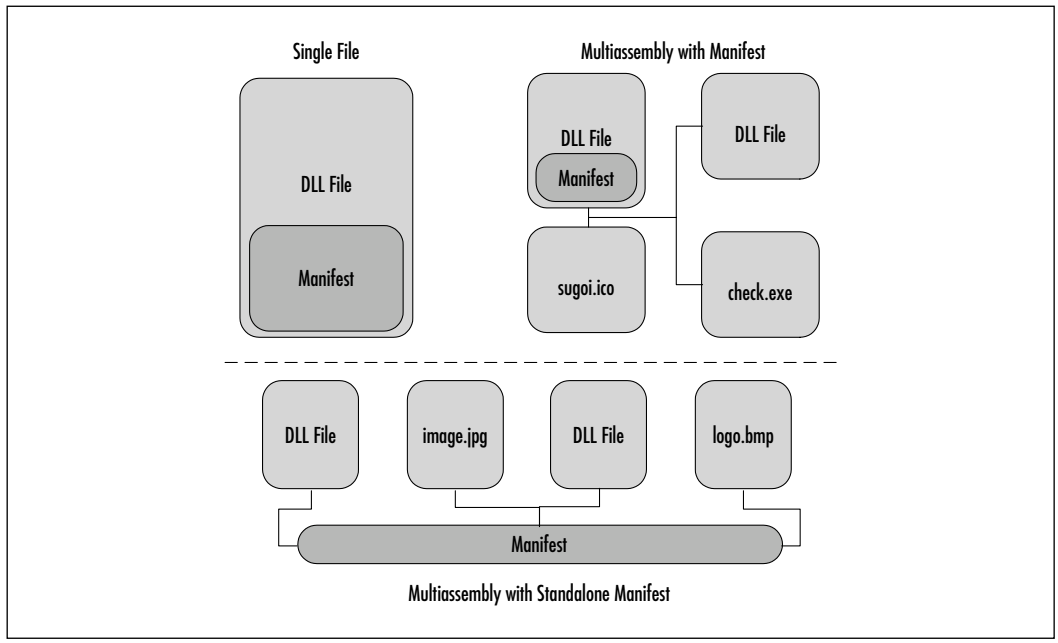
Custom Attributes	Description
AssemblyCultureAttribute	Contains information on the *cultural* settings, such as base language or time zone.
AssemblyDelaySignAttribute	Tells the CLR that there is some extra space that might be empty to reserve space for a future digital signature.
AssemblyKeyFileAttribute	Contains the name of the file that contains the key pair for a shared name.
AssemblyKeyNameAttribute	If you use the CSP option, the key will be stored within a key container. This attribute returns the name of the key container.
AssemblyOperatingSystem Attribute	Information on the operating system(s) supported by the assembly.
AssemblyProcessAttribute	Information on the CPU(s) supported by the assembly.
AssemblyVersionAttribute	Returns the version of the assembly in the standard *major.minor.build.revision* form.

In regard to the third assembly option, *location*, a manifest's location on the assembly can also be altered, based on the type of assembly deployment. An assembly can be deployed as either a single file or multiple files. A single file assembly is much like a standard DLL file; its manifest is placed directly within the application. Again, the assembly is not that different from the standard executable or DLL; what changes is how it's run. In a multifile assembly, the manifest is either incorporated into the main file (such as the main DLL file), or as a standalone (Figure 1.3).

NOTE

Depending on what you are doing, you might want to use a standalone manifest for any multifile assembly. A standalone manifest provides a consistent access location for the manifest and ensures that it will be there when needed. However, constantly referencing the assembly can carry a small memory overhead, so its advantage shines with larger, multifile assemblies.

Figure 1.3 Manifest Location within an Assembly



Assembly Cache

The cache on which the CLR depends is called the *machinewide code cache*. This cache is further divided into two subsections: the *global assembly cache* and the *download cache*. The download cache simply handles all the online codebases that the assembly requires. The global download cache stores and deals with the assemblies that are required for use within the local machine; namely, those that came from an installer or an SDK. Only assemblies that have a shared name can be entered into the global assembly cache, since the CLR assumes that these files will be used frequently and between programs.

Even though a file will be used often, however, it can still be sluggish. Since the CLR knows that to enter the global assembly cache, the assembly must be verified, it assumes that it is already verified and does not go through the verification process, thus increasing the time it takes to reference the assembly within the global assembly cache. One integrity check is performed on it prior to entry into the global assembly cache; this integrity check consists of verifying the hash code and algorithms located within the manifest. Furthermore, if multiple files attempt to reference the assembly, a single dedicated instance of the assembly is

created to handle all the references, which allows the assemblies to load faster and reference faster across multi-assembly situations.

A file that's located in the global assembly also experiences a higher degree of end-user security, since only an administrator can delete files located within the global assembly cache. In addition, the integrity checks ensure that an assembly has not been tampered with, since assemblies within the global assembly cache can be accessed directly from the file system.

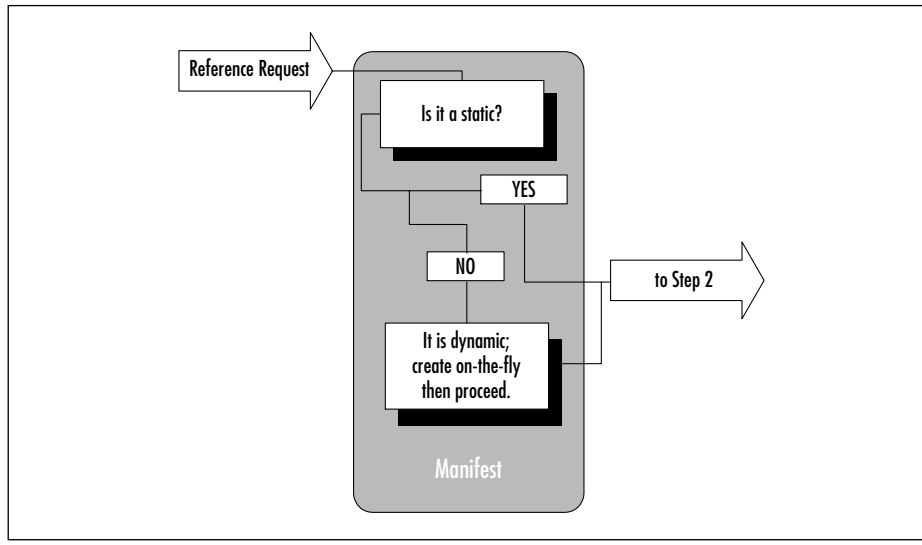
Locating an Assembly

Once the assembly is created, finished, and deployed, its scope is basically private; in other words, the assembly will *not* in any way, shape, or form interfere with any other assemblies, DLL files, or settings that are *not* declared in the assembly's manifest. It's all part of CLR's automation; it used to be that only VB coders had protection from memory leaks or other types of problems by inadvertently creating a program that went too far out of its area, but now, the CLR handles all that.

Now, a single assembly is easy to run, and easy for the CLR to locate. However, when dealing with multiple files, you might ask yourself, "Wait—if the assembly is so tightly locked, how can multiple assemblies interact with each other?" It's a good question to ask; most programmers working with .NET create multifile assemblies, and so we need to understand the process the CLR takes to locate an assembly. It goes like this:

1. **Locate the reference and begin to bind the assembly(ies).** Once the request has been made (through *AssemblyRef*) by an assembly in a multi-assembly to reference *another* assembly within the multi-assembly, the runtime attempts to resolve a reference in the manifest that tells the CLR where to go. The reference within the manifest is either a static reference or a dynamic reference. A *static reference* is a reference created at build time by the compiler; a *dynamic reference* is created as an on-the-fly call is made. Figure 1.4 illustrates Step 1 of the location process.
2. **Check the version policy in the configuration file.** The CLR checks to see if there's a configuration file; for client-side executables, it usually resides in the same directory with the same name, but has a *.CFG extension. For Internet-based applications, the application must be explicitly declared in the HTML file. A standard configuration file can look like the following:

Figure 1.4 Step 1 of the Location Process




```
<?xml version = "1.0">
<Configuration>
  <AppDomain
    PrivatePath="bin;etc;etc;code"
    ShadowCOpy="true"/>
  <BindingMode>
     <AppBindingMode Mode="normal"/>
  </BindingMode>
  <BindingPolicy>
    <BindingRedir Name="TestBoy"
                  Originator="45asdf879er423"
                  Version="*" VersionNew="7.77"
                  UseLatestBuildRevision="yes"/>
  </BindingPolicy>
  <Assemblies>
    <CodeBaseHit Name="s_test_mod.dll"
                 Originator="12d57w8d9r6g7a3r"
                 Version="7.77"
                 CodeBase=http://thisisan/hreflink/test.dll/>
  </Assemblies>
</Configuration>
```

The document element of this XML file is Configuration. All this node does is tell the CLR that it's found a configuration file type, and that it should look through it to see if this type is the one it needs. The first node contains the *AppDomain* element that has the *PrivatePath* and *ShadowCopy* attributes. *PrivatePath* points to a shared and private path to the bin(s) directory(ies). The path is the location of the assemblies that you need and the location of the global assembly cache.

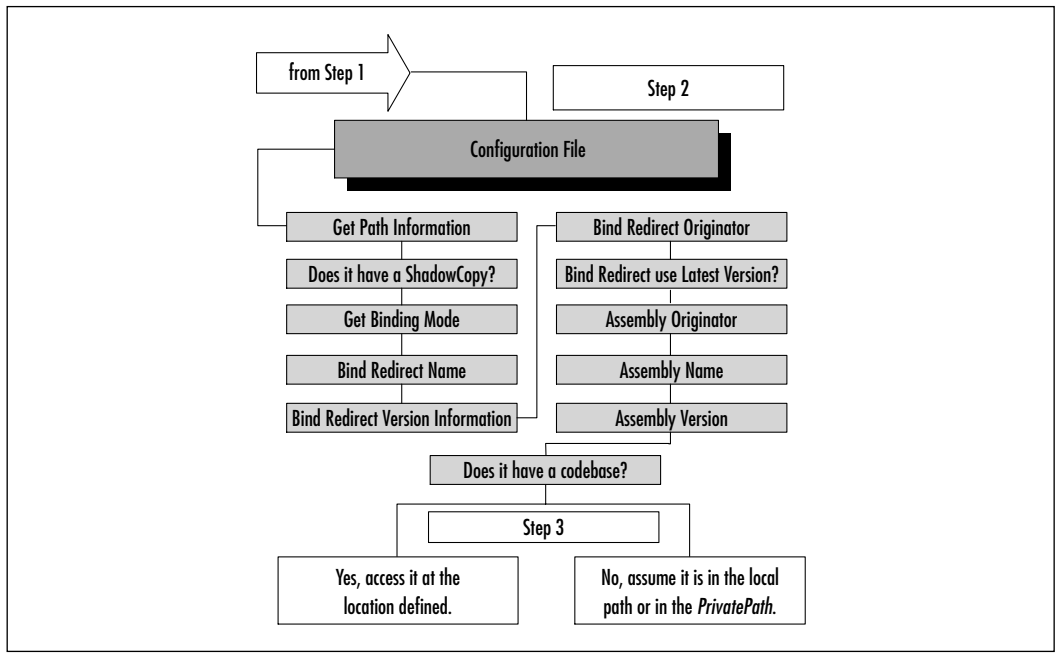
Keep in mind that the *PrivatePath* attribute is relative to the assembly's root directory and/or subdirectories thereof; anything outside of that needs to be either in the global assembly cache or linked to using the *CodeBase* attribute of the *Assemblies* attribute. *ShadowCopy* is used to determine whether an assembly should be copied into the local download cache, even if it can be run remotely.

The next node contains *BindingMode*. Binding mode refers to how the assemblies within the application should "bind" to their exact versions. *BindingMode* contains the *AppBindingMode* element, which declares the *BindingMode* to be *safe* or *normal*. A *safe* binding mode indicates that this assembly is of the same assembly version as the others when the application is deployed. No Quick Fix Engineering (QFE) methods are applied, and any version policies are ignored; these characteristics apply to the entire application. *Normal* mode is simply the normal binding process in which the QFE is used and version policies are applied.

BindingPolicy stores the *BindingRedir* element, which deals with the attributes that tell the CLR which version to look for. This type of element applies only to assemblies that are shared. The *Name* attribute is the assembly's name; *Originator* contains an 8-byte public key of the assembly; and *Version* can either explicitly state which version the assembly should be redirected to, or uses a wildcard (*) to signify that all versions should be redirected. *VersionNew* contains the version to which the CLR should be redirected, and *UseLatestBuildVersion* contains a yes/no value that states whether the QFE will automatically update it.

Assemblies stores the tags that the CLR can use to locate an assembly. The tags in this element are always attempted before a thorough search. *Name* and *Originator* contain the same information that they contain in the *BindingPolicy*. *Version* contains only the current version of the assembly; *CodeBase* contains the URL at which the assembly can be located. Figure 1.5 illustrates Steps 2 and 3.

Figure 1.5 Steps 2 and 3 of the Location Process



NOTE

The reference that's checked against from the *AssemblyRef* contains the following information from the assembly it's asking for: text name, version, culture, and originator if it has a shared name. Of the references listed, the location process can work without all of them except the name. If it can't find culture, version, or originator (which only shows up on shared names), it will try to match the filename and then the newest version.

WARNING

Even though you *can* use partial references, doing so kills the whole concept of version support, and cancause you to use the *wrong* file at times. For example, let's say that you've created a whole new set of classes and need to benchmark the differences. If you are using partial references, it's more than likely that the new version will be selected over the old version. Be precise, even if it's tedious to do so!

3. **Locate the assembly via probing or codebase.** When the information stored in the Configuration file is retrieved, it is then checked against the information it has in the reference and determines whether it should locate the file at the specified URL codebase or via location probe. In the case of a codebase, the URL is referenced and the file's version, name, culture, and originator are retrieved to determine a match. If any of these fails, the location process stops. The only exception is if the version is equal to or greater than the version needed; if it is greater or equal to and all the other references check out, the location process proceeds to Step 4. If no URL is listed for a codebase, the CLR will probe for the needed assembly under the root directory.

   Probing is a bit different and more thorough than looking at the URL, but definitely more lax in verifying references. When probing begins, it checks within the root directory for a file with the assembly name ending with *.MCL, *.DLL, or *.EXE. If it's not found in the root, it continues to check all the paths listed in the *PrivatePath* attribute of *AppDomain* of the configuration file. The CLR also checks a path with the name of the assembly in it. Again, if an error is found, the location process stops; if it's found and verified, it proceeds to Step 4.

4. **Use the global assembly cache and QFE.** The global assembly cache is where global assemblies that are used throughout multiple programs are found. All global assemblies have a shared name so that they can be located through a probe. QFE, refers to a method in which the latest build and revision are used; it's done this way to allow greater ease for software vendors to provide patches by recreating just one assembly instead of the entire program. If the assembly is found and the QFE is off, the runtime double-checks in the global assembly cache with a QFE for the particular assembly; if a greater revision/build is found, that version takes the place of the one found while probing.

5. **Apply the administrator policy.** At this point, any versioning policies are applied (versioning policies are stored in the admin.cfg file of the Windows directory), and the program is run with the policies applied. The only major impact on this policy occurs if an administrator policy initiates a redirect to a version; if this happens, the version must be located in the global assembly cache before the redirect occurs. The runtime assumes that since the redirect is administrative, the user manu-

ally and consciously set it, and has supplied the needed file in the global assembly cache.

Private Assembly Files

Private assembly files are normally single applications and reside in its own directory without needing to retrieve any information or use resources from an assembly that is located outside its own folder. This does not mean that the private assembly can't access the standard namespaces; it simply means that they do not use or require any other external applications to properly function. These types of assemblies are useful if the assembly will be constantly reused and does not rely on any other assembly. Private assembly files are not affected by versioning constraints.

Shared Assembly Files

Shared assembly files are generally reserved for multi-assembly applications and store commonly used components, such as the graphical user interface (GUI) and/or frequently used low-end components. These assemblies are stored in the global assembly cache, and the CLR does enforce versioning constraints. Examples of a shared assembly are the built-in .NET Framework classes.

A shared assembly, as you might have guessed, is the exact opposite of a private assembly. A shared assembly does stretch outside the bounds of its directories and requires resources that are found within other assemblies. Shared assemblies come into play heavily when dealing with modular applications. For example, a GUI that is used between several applications can be stored as a shared assembly or as a commonly used database routine.

Understanding Metadata

Two things happen when you create your assembly: Your code is transformed into MSIL, and all the relevant information contained in the code—types, references, and so on—are noted within the manifest as metadata. When the CLR kicks in, it inserts the metadata into in-memory data and uses it as a reference in locating what it needs according to the program. This road map provides a large part of interoperability, since the CLR doesn't actually need to know what code it's programmed in; it simply looks at the metadata to find out what it needs and where it's going.

The metadata is responsible for conveying the following information to the CLR:

- Security permissions
- Types exported
- Identity
- External assembly references
- Interface name
- Interface visibility
- Local assembly members

The Benefits of Metadata

The items in metadata are placed within in-memory data structures by the CLR when run. This allows metadata to be used more freely with faster access time. This system enhances the self-describing functions of .NET assemblies by having readily available all of the items that the assembly needs to interact. This also allows for other objects (per the metadata, of course) to interact with the assembly.

Metadata also allows interoperability by creating a layer between the assembly's code and what the CLR sees. The CLR uses the metadata extensively, thus removing the burden of operability from the CPU/language. The CLR reads, stores, and uses the metadata through a set of APIs, most notably the managed *reflection* and *reflection emit* services. The layer abstraction causes the runtime to continue optimizing in-memory manifest items without needing to reference any of the original compilers, and enables a snap-in type of persistence that allows CLR binary representations, interfacing with unmanaged types, and any other format needed to be placed in memory.

You might have been surprised when you saw that the metadata allows unmanaged types to show up; however, this does not impact the CLR in any way. Unmanaged metadata APIs are not checked, nor do they enforce the constraints present. However, the burden of verifying unmanaged metadata APIs is placed solely on the compiler.

Identifying an Assembly with Metadata

Metadata identifies each assembly with the following: Name, culture, version, and public key. The *name* used is the textual name of the assembly or the name you gave it when you created it. The *culture* simply references the cultural settings

used, such as language, time zone, country/region, and other localization items. The *public key* used is the same one generated by the assembly.

Types

In unmanaged code we referred to types as *objects*. Types, like objects, contain data and logic that are exposed as methods, properties, and fields. The big differences between the two lie in the properties and fields; *properties* contain logic in order to verify or construct data, whereas *fields* act like public variables. *Methods* are unchanged. Types also provide a way to create two different representations with different types by looking at the two different types as part of the same interface—in other words, they have similar responses to events.

Currently, two types are available to .NET users: value types and reference types. *Reference* types describe the values as the location of bits, and can be described as an object, interface, or pointer type. An *object* type references a self-describing value, an *interface* type is a partial description that is supported by other object types, and the *pointer* type is a compile-time description of a machine-address location value.

When dealing with classes, the CLR uses any method it deems fit, according to the CTS. Metadata has a special mark for each class that describes to the CLR which method it should use. Table 1.3 lists the layout rules that metadata marks for each class.

Table 1.3 Class Layout Rules

Class	Layout Rules
AutoLayout	CLR has free reign over how the class is laid out; this shows up more often on the inconsequential classes.
LayoutSequential	CLR guides the loader to preserve field order as defined, but offsets are based on the field's CLR type.
ExplicitLayout	CLR ignores field sequence and uses the rules the user provides.

Defining Members

Members are the methods, fields, properties, events, and nested types that are found within a type. These items are descriptions of the types themselves and are

defined within the metadata. This is one of the reasons that access of items through metadata is so efficient.

Fields, arrays, and values are subvalues of a value representation. Field subvalues are named, but when accessed through an index they are treated as array elements; a type that describes the values composed of array elements creates a true array type with values of a single type. Finally, the compound type is a value of a set of fields that can hold fields of different types.

Methods are operations that are associated with a particular type or a value within the type. For security purposes, methods are named and signed with the allowed types of arguments and return values. Static methods are methods that are tied directly to the type; virtual methods are tied to the value of the type. The CLR also allows the *this* keyword to be null within a virtual method.

Using Contracts

The signature that methods use is part of a set of signatures referred to as a *contract*. The contract brings together sets of shared assumptions from the signatures between all implementers and users of the contract, providing a level of check and enforcement. They aren't real types; rather, they are the requirements that a type needs to be properly implemented. Contract information is defined within the class definition.

Class contracts are one of the most common; they are specified within a class definition, and in this case are defined as the class type along with the class definition. The contract represents the values and other contracts supported by the type, and allows inheritance of other contracts within other types.

An *interface contract* is defined within an interface. Just like the class definition, an interface definition defines both the interface contract and the interface type. It can perform the functions that a class contract can, but it cannot describe the representation of a value, nor can it support a class contract.

A *method contract* is defined within a method definition. Just like a normal method, it's an operation that's named and specifies the contract between the method and the callers of the method. It exerts the most control over parameters, specifying the contract for each parameter in the method that it must support and the contracts for each return value, if there is one.

A *property contract* is defined within a property definition. The property contract specifies the method contract used for the subset of operations that handle a named value, including the read/change operations. Each property contract can be used only with a single type, but a type can use multiple property contracts.

An *event contract* is defined in an event definition. It specifies method contracts for the basic event operations (such as the activation of an event), and for any operations implemented by any type that uses the event contract. Like the property contract, each event contract can be used only with a single type, but a type can use multiple event contracts.

Assembly Dependencies

An assembly can depend on another assembly by referencing the resources that are within the scope of another assembly from the current assembly scope. The assembly that made the reference has control over how the reference is resolved; this gives the assembly mapping control over the reference onto a particular version of the referenced assembly. When you depend on an external assembly, you can choose to let the CLR assume that the files are present in the deployed environment or will be deployed with the corresponding assemblies. Such an assumption can be pretty large or problematic, but the CLR is smart enough to know what to do if it's not there.

Unmanaged Assembly Code

Unmanaged code is not left out of the manifest. To allow interoperability with unmanaged COM / code, a GUID is included with each type, adding a version number to classes, and includes a collection of type references that are raised as events. An unmanaged assembly attempts to receive data from an assembly through reformatting via the Virtual Execution System.

Reflection

The concept of *reflection* is available to the user via the *System.Reflection* namespace. In essence, reflection reflects the composition of other .NET code back to us. It can discover everything that is vital within the assembly, such as the classes, events, properties, and methods exposed by the assembly. We can then use this information to "clone" an instance of that assembly so that we can use the classes and methods defined there.

Using reflection can theoretically provide access to nonpublic information such as code, data, and other information that is normally restricted due to isolation. .NET provides a built-in check system of rules to determine just what you can get using reflection. If you really have to use nonpublic information, you need to use *ReflectionPermission*. *ReflectionPermission* is a class located within the

Object.CodeAccessPermission namespace and gives access to all the nonpublic information when requested by a reflection. This class can theoretically also give someone the ability to view your code, so *do not use this class* if you can avoid it! You definitely will not want to use this on Internet applications. By default and without needing permission, reflection can access or do the following:

- Public types
- Public members
- Module/assembly location
- Enumerate assemblies and modules
- Enumerate nonpublic types (have to be in the same location as the assembly using reflection)
- Enumerate public types
- Invoke public, family access (of calling code class), and assembly access (of calling-code class) members

Attributes

An attribute allows you to add descriptive declarations that behave similarly to keywords. You can use attributes to annotate types, methods, fields, properties, and other programming elements. They are stored within the metadata and can help the CLR understand the description of your code. Attributes can describe the way in which data is serialized, describe security characteristics, or limit JIT compilation for debugging purposes. Perhaps one of the most versatile of the metadata items, attributes can even add descriptive elements to your code to affect its runtime behavior. A simple attribute might be used like this:

```
Public Class <attribute()> ClassName
```

In this example, the class *ClassName* is described by the attribute *attribute()*. This means that when the CLR hits this class, it will alter its behavior according to what *attribute()* says.

Ending DLL Hell

Everyone knows what DLL Hell is: It's that situation that happens when an older or newer DLL file overwrites the previous copy after the installation of a new application (usually a newer DLL that is not backward compatible). Registry settings are changed; some are added, some are removed, and some are altered.

GUIDs could change and, in the blink of an eye, all these things accumulate into a situation in which one DLL file prevents your application from working.

In order to prevent DLL Hell, the .NET Framework performs the following:

- Application isolation is enforced.
- "Last known good" system from Windows NT systems is enforced.
- Side-by-side deployment is permitted and backed up by isolation.
- File version information is recorded and enforced.
- Applications are self-describing.

Side-by-Side Deployment

Side-by-side execution allows two different versions of the same assembly file to run simultaneously. This is an advantage of the isolation provided to each assembly. Side-by-side deployment removes the dependency on backward compatibility that often causes DLL Hell. Side-by-side execution can be running either on the same machine or in the same process.

Side-by-side deployment in the same process can be the most strenuous to code for; you have to write the code so that no process-wide resources are used. The extra work pays off in that you can run multiple components and objects in the same thread, allowing for greater process flexibility and usage.

Side-by-side deployment on the same machine puts less stress on the code writer, but still has its quirks. The biggest point to look out for when coding this way is to write in support for multiple applications attempting to use the same resource; you can work around this by removing the dependency on the resource and allowing each version to have its own cache.

Versioning Support

Versioning is the method .NET uses with assemblies that have a shared name; it tells the CLR the version of the particular assembly. Each assembly has two types of version information available: the compatibility version and the informational version. The *compatibility version* is the first number, which the CLR uses to determine identities. The *informational version* allows for an extra string description of the assembly that the CLR doesn't really need.

The version number looks like your typical version—a four-part number that describes, in order, the major build version, the minor build version, the build, and the revision. If there are any changes to the major or minor versions, the assembly is used as a separate entity and is isolated. The build and the revision

signify a build compatible to the present assembly, which means that this new version contains a bug fix or patch.

The major and minor numbers are used to perform incompatibility checks. In other words, compatibility is weighed against the major and minor numbers; any difference in either of these two numbers tells the runtime that it is a new release with many changes and should be treated accordingly. The build number tells the runtime that a change has been made, but does not carry a high incompatibility risk. It's been our experience that relying on the build number at times is very bad practice, especially if the minor change involves your types. In fact, whenever you change anything, such as how a class is referenced, you should treat it as a major/minor revision unless you *absolutely* take all the necessary steps to make the class backward compatible.

When you do create a backward-compatible class, try to create it as a bug fix or patch, and define the change in the QFE. That way, the runtime assumes that backward compatibility is in place, since there should be no major changes (again, such as class references), and uses it accordingly unless it is explicitly told not to by a configuration file.

Using System Services

System services combine everything that the runtime makes available, such as exception (error) handling, memory management, and console input/output (I/O). Some of the topics discussed here might not be new to some VB programmers, especially those who have had some exposure to Java or C/C++.

Memory management really hasn't changed a lot, only the way in which it's implemented. Instead of programmers having full control over object instantiation and destruction, the CLR takes over that task. However, we do have the ability now to create standard command-line programs.

Exception Handling

.NET introduces the implementation of a Try/Catch system through its new Exception object. Some of you might be familiar with this concept from previous Java work. A simple try/catch statement can look like the following:

```
Try
{
Thiswillcrash();
}
```

```
Catch(error_from_Thiswillcrash)
(
//react to the error thrown by Thiswillcrash()
}
```

In essence, a try/catch set will place the function or sub within a try "wrapper" that will monitor any error messages. If an error message matches "error_from_Thiswillcrash", then the catch "wrapper" generates the appropriate response to the error. This gives programmers more flexibility in determining errors and how they want to handle the error, instead of letting Windows do it and hoping for the best.

A perfect scenario is a DLL file. Within the DLL file, you have a standard file read and file write system. However, instead of just generating a failure error if the file that needs to be read is not found, you would just display a message that says "this file is being created" and then creates the file without the user even knowing that an error occurred. A simple way of doing a try/catch for this situation might appear as follows:

```
Try
{
FileReadDisplay();
}
Catch(File_not_found_error)
{
//display message "This file is being created"
//create file that matches needed defaults
//display message "A new default file has been generated.
//Please reset your defaults."
}
```

The try/catch system is part of the *Exception* class. The *Exception* class brings with it some extra goodies for debugging, including *StackTrace*, *InnerException*, *Message*, and *HelpLink*.

StackTrace

Stacks haven't changed over the years; a *stack* is still a special type of data structure in which items are removed in the reverse order in which they were added (last in, first out, or LIFO). This means that the most recently added item is the first to

be removed. *StackTrace*, quite simply, allows you trace the stack for errors. It is most useful when dealing with constant errors along loops and within a try/catch statement. *StackTrace* is useful when it is defined before a try statement and when it ends after the catch statement.

InnerException

An *InnerException* can store a series of exceptions that occur during error handling. You can then format the series of exceptions into a new exception that contains the series. It's almost like a waterfall view; an exception is thrown, which in turn throws another exception. Using *InnerException*, the first exception would be stored within the last exception and so on, giving the developer an ample road map to locate the starting point of an error.

Message

Message stores a more in-depth error description. This is extremely useful when used in conjunction with *InnerException*.

HelpLink

Using *HelpLink*, you can set a specific URL or URN within a try/catch block to point to an article or help file that has more details on the error generated.

Garbage Collection

Garbage Collection (GC) is .NET's method for handling object creation and destruction, as well as cleanup and preventive maintenance. GC does not rely on reference counting; it has its own unique system for detecting and determining which objects are no longer in use. In this sense, .NET is smart enough to know when a file is being used and when it needs to be removed. We delve into a full overview of GC in the *Relying on Automatic Resource Management* section later in this chapter.

Console I/O

Console applications are those little programs that pop up a DOS box and run from the command line. Command-line applications can be used in middle-tier situations, in testing a new class, or even for creating DOS-based functionality for a utility tool. We have this capability thanks to the *System.Console* namespace. (We discuss namespaces later in this chapter.)

Here's a brief example of a simple command-line VB application:

```
Import System.Console

Sub Main()
   Dim readIN as String
   WriteLine("This is a line!")
   ReadIN = ReadLine()
   WriteLine(ReadIN)
End Sub
```

The console would print "This is a line!" with a carriage return at the end automatically, giving us one line to write whatever we want. After a carriage return is detected, what we wrote is stored within the variable *ReadIN* and then displayed via *WriteLine*.

Microsoft Intermediate Language

Once your assembly is in managed code, the CLR in turn translates the code to the MSIL. MSIL is a type of bytecode that gives .NET developers the portability they need, but it is also essential to the system's interoperability, since it provides the JIT compiler with the information it needs to create the necessary native code. MSIL is platform independent.

MSIL also creates the metadata that is found within an assembly. Both the MSIL and metadata are stored within an extended and modified version of the PE (which is more a combination between PE's syntax and the Common Object File Format, or COFF, object system). MSIL's flexibility allows an assembly to properly define itself and declare all it needs for self-description.

The Just-in-Time Compiler

Without the JIT, we wouldn't have any functioning .NET programs. The JIT turns the MSIL code into the native code for the particular platform on which it's running. Each version of .NET for each individual platform also includes a JIT for that specific platform architecture. For example, an *x*86 version of .NET can compile .NET code from a non-*x*86 architecture, because the JIT on the *x*86 machine translates the MSIL into *x*86-specific code and contains no platform-specific code.

JIT's method of code compilation is literally "just in time"—it compiles the MSIL code as it's needed. This method guarantees faster program loading time

and less overhead in the long run, since JIT compiles what is needed when it's needed. MSIL, when created and referenced, creates a stub to mark the methods within the class being used. JIT compiles just the stubbed code and replaces the stubs within the MSIL to the location of the compiled code address.

There are currently two flavors of JIT: *Normal JIT* and *Economy JIT*. Economy JIT is geared toward intensive CPU/RAM usage systems, such as Windows CE platforms. Economy JIT differs from normal JIT in that, in order to make the best of the intensive CPU/RAM usage situation, it replaces the stubs in the MSIL with the actual compiled code, not a reference to its address. Microsoft currently claims that economy JIT is less efficient than normal JIT for this reason. However, a decent bench exam of these two compilers has yet to be done.

Using the Namespace System to Organize Classes

We've already seen an example of namespaces in the previous code example, but what are they? Namespaces are references that we place within the code that point to the location of the object or class that we need to use within the .NET Framework. In the previous code example, we used the *System.Console* namespace. This naming scheme is used only for organizational purposes, but it is vital that you understand it.

A namespace is basically a hierarchical system created to organize intrinsic classes that provide the basic functions that come with .NET. Each class is kept within a namespace that suits its use; for example, Web-related classes are kept within the *System.web* namespace. Each namespace can contain namespaces, providing more functionality for each namespace. The system namespace is the root namespace on all .NET machines.

.NET allows for multiple namespaces, classes, interfaces, and other valid types declared within it. The following example displays a sample namespace that contains multiple assemblies and an assembly that is stored within a namespace:

```
MyNamespace.namespace.class
MyNamespace.enum
MyNamespace.interface.class
MyNamespace.Namespace.class
```

Here we have the *MyNamespace* base namespace with multiple namespaces that, in turn, contain all the needed operations, functions, and procedures to pro-

vide necessary services. Each namespace can have classes that have the same name; for example, *Assembly3* and *Assembly5* can both have a *Count* class. However, within a single namespace, there cannot be any duplicate class names. Namespaces can also be local or global; local namespaces can be seen only by the current application, and global namespaces can be seen on the entire machine.

The Common Type System

The *Common Type System* (CTS) gives the CLR a description of the types that are supported and used, and how they are presented in metadata. The type in CTS represents the type system, which is one of the more important parts of .NET for cross-language support. The type provides the rules and logical steps that a language compiler employs to define, reference, use, and store information. If you are using any CLR-compliant compiler outside of the .NET Framework, it must use the CTS system to properly create the assembly. The type system that the CTS uses contains classes, interfaces, and value types.

A class is now contained within a type. In fact, the term *type* is sometimes used (although sometimes erroneously) with the same meaning as *object* to reflect .NET. The term still has the same functionality as in any other object-oriented programming (OOP) language; it can define variables, hold the state of objects, perform methods and events, and create, set, and retrieve properties. Every time an instance of a .NET class is created, it is treated as an object. Table 1.4 lists the characteristics of a class, and Table 1.5 lists the characteristics of the members.

Table 1.4 Class Characteristics

Class	Characteristics
Sealed	Class derivations are prohibited.
Implements	Interface contracts are fulfilled by this class.
Abstract	This class can't be instantiated on its own; in order to use it, you must derive a class from it—just like abstract classes in C/C++.
Inherits	This means that the class being defined will inherit the characteristics (i.e., properties, fields, methods) of the class that is written next to it. You can use the same characteristics or override them.
Exported	This class can be viewed outside the assembly.
Not-Exported	This class cannot be viewed outside the assembly.

Table 1.5 Member Characteristics

Members	Characteristics
Private	Defines accessibility as permitted only within the same class or a member of a nested class within the same class.
Family	Defines accessibility as permitted within the same class as the member and subtypes that inherit it.
Assembly	Defines accessibility as permitted only from within the assembly in which the member is implemented.
Family or *Assembly*	Defines accessibility as permitted only by a class that qualifies as a family or an assembly.
Public	Defines accessibility as permitted from any class.
Abstract	A nonimplemented member; as with C/C++, you have to derive a class from it in order to implement it.
Final	A method with the final statement cannot be overridden; this helps prevent any unintentional overrides that can damage functionality.
Overrides	Used by virtual methods; it replaces the predefined implementation from the derived class.
Static	A method that is declared static exists without needing to be instantiated and can be referenced through all class instances.
Overloads	An overloaded method has the same name as another method and the same code, but its parameters, order of parameters, or calling convention might be different. This is useful for adding last-minute functionality to a method that you might only need once.
Virtual	Used to create a virtual method in order to have the functionality provided by *Overrides*.
Synchronized	Limits usage of implementation to one thread at a time.

NOTE

The Virtual Execution System is tied in with the CTS concept. In fact, it's a special execution engine that was created just to ensure that the tenants of the CTS are implemented.

Developing & Deploying…

Abstract Classes?

If you've never used C/C++, abstract classes might be a foreign concept to you. An *abstract class* can be defined as a "skeleton" class that has no actual code within it—simply a declaration of what a class that can be derived needs to have within its structure to be considered a derivative of the skeleton. In other words, the "flesh on the bones" is added later.

Abstract classes are useful when you need to create some sort of base class that needs to be reused, but have no need for it later—sort of like a blueprint. For example, take the abstract class *fruit_eater*:

```
Abstract class fruit_eater
{
   Private Me_eat As Integer
   Me_eat = 1

   Public Property Eat() As Integer
    Get
       Return Me_eat
    End Get
   End Property

End Class

Public class monkey_boy
  Inherits fruit_eater

  Public Property me_do_eat() as String
     If Eat = 1 Then
         'code goes here to tell you that monkey_boy eats fruit!
     End If
```

Continued

```
    End Property


  End Class
```

Using the abstract class *fruit_eater*, we set a requirement that class *monkey_boy* must have to say that *monkey_boy* eats fruit. This can be further expounded to another class, *animal_kingdom*, which can use *fruit_eater* to organize between herbivores and carnivores within its kingdom of wild animals and *monkey_boys*.

Type Safety

Type safety is one of the guarantees that .NET gives us; it limits access to memory locations to which it has authorization. Therefore, if we have Object A trying to reference the memory location of Object B that is within the memory area of Assembly C, Object A will not be allowed access. Even if Object A tries to access a memory location that is accessible by its assembly and does not have permission, it will be denied. An optional verification process can be run on the MSIL to verify that the code is type safe; it's optional because it can be skipped based on permissions given to the code.

Type-safe code tells the runtime that it can go ahead and isolate the code, since it's not going to need anything outside its boundaries. Even if the trust levels are different within type-safe code, it can execute on the same process. Code that is not type-safe might cause crashes during runtime or even shut down your entire system, so be careful with it. Remember, we're working with a beta runtime, and it can be touchy!

Relying on Automatic Resource Management

Here we get to the nuts and bolts of .NET. All the changes we've seen so far in this chapter are either semantically oriented or enhancements, but the way in which .NET handles memory management is a completely different story. For a long time, we've used the deterministic finalization system, in which we declare that the code ran on the class initialization and termination and had control over where a class was terminated. Deterministic finalization had its drawbacks; if the

programmer forgot to declare the class empty (null, in some cases), or simply forgot to run the termination event, we'd have a memory leak or worse when control over the project terminated.

This outdated memory management system is referred to as *reference counting*. A count is kept within each object, usually in its header, of how many references there are for the object. Each application (or client, as it is referred to in COM circles) that is referencing an object states when it is referencing the object and when it is releasing the object. As new objects are instantiated, the count (or number of objects in the count) is incremented and decremented when the object is either overwritten or recycled.

The burden of doing the actual cleanup of the object, however, was not on the application. All the application did was merely issue the destroy command to the object; the object then had to free itself from the reference count. When an object was not properly deallocated (destroyed), we had an instance of a memory leak. Reference counting also had a limited growth size; objects became "bloated" (made bigger artificially) in order to store the reference count, and, of course, cyclic objects generated the previously mentioned nonzero reference count.

.NET replaces all this with *automatic resource management*. The runtime is now "smart" enough to know when and how to handle memory allocation, deallocation, and usage. A major drawback is that we can't control when an object or a class is terminated; therefore, we have no knowledge of when the termination takes place. This is a very valid point and, quite honestly, the only noticeable drawback because it won't release the memory and so we encounter a dead reference. However, most of the time this won't matter, because Garbage Collection will eventually get to it. Now let's see how .NET handles memory and how Garbage Collection is tied up in all of this.

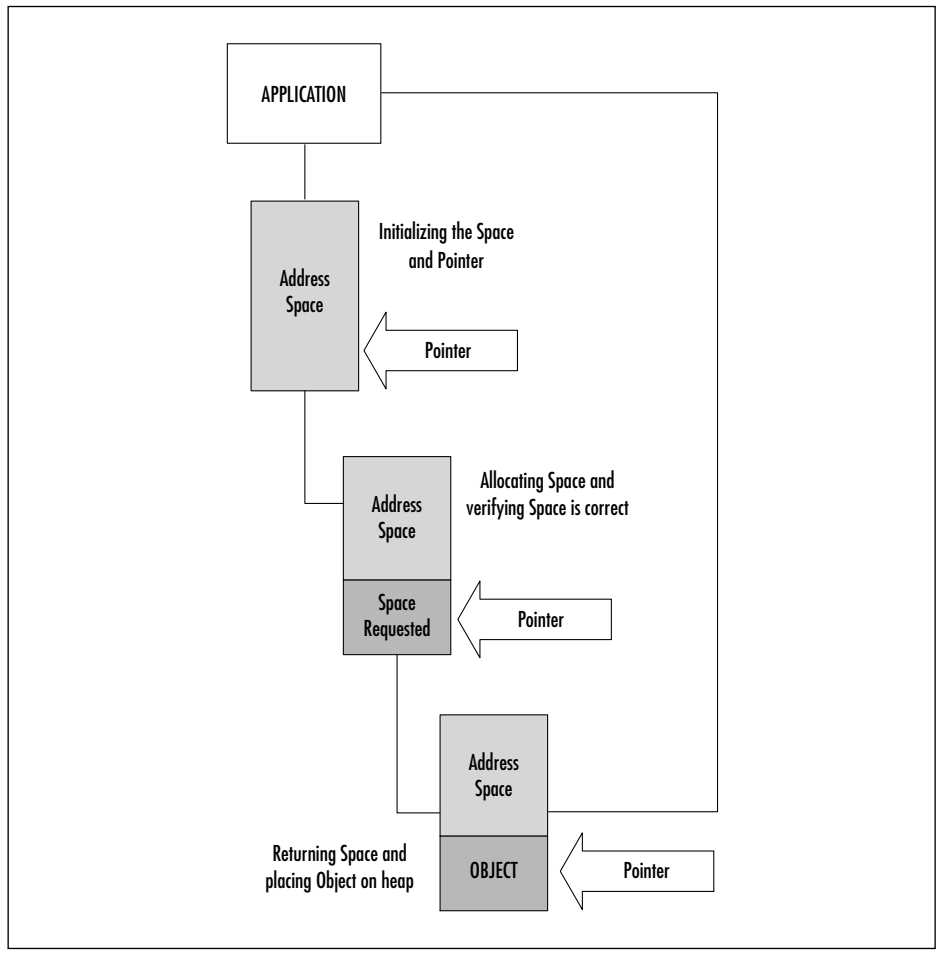
The Managed Heap

When a program is run in .NET, the runtime creates the region of address space it knows it needs, but does not store anything on it. This region is the *heap* (also referred to as the *free store* or *freestore*). .NET controls the heap and determines when it's time to free an object. Figure 1.6 illustrates the following pointer interaction process:

1. A pointer is created for the allocated space (heap) that keeps track of the next available free area on the allocated space that the runtime can use for storage.

2. As the application creates new objects, the runtime checks to see if the space currently being pointed to can handle the new object; if it can't, it dynamically creates the space.
3. The object is then placed on the heap, its constructor is called, and the new operator returns the address block of our newly created object.

Figure 1.6 Pointer Interaction with a Managed Heap



NOTE

When an object/type is over 20,000 bytes, a special "large heap" is created to store it. This special heap does not go through compression when Garbage Collection is called. Compression occurs during the generation process, described in a later section in this chapter.

Garbage Collection and the Managed Heap

As mentioned, .NET handles the managed heap by using Garbage Collection. In its purest sense, GC is an algorithm designed to determine when the life cycle of an object has ended. In order to determine if an object is at or near its end, GC analyzes the root of the object. Roots (also known as *strong references*), much like actual roots found in nature, act as roadmaps to where vital resources, such as objects, are stored. Global or static pointers, local variables that are on a thread stack, and CPU registers containing pointers to the heap are all considered roots. All visible roots are stored in a list created and updated by the JIT and the CLR.

Once GC starts, it'll assume that all the roots available to the heap are null. This makes the GC begin a *verification* process in which it goes through each root recursively and starts to make a graph that contains all the references available and any linked references (e.g., Object A references Object B). This step is repeated once more to make sure that everything is in place by assuming that if it's a duplicate object, it's already on the list and thus a legitimate object, meaning that the graph it just built is correct. The final step of this verification process is that GC starts to trace the root of each object to determine if the root is coming from the program that is going to use the current address space. Any objects without roots are considered null or no longer in use, and are treated as garbage—which is an accurate assumption since no two applications share the same address space—and are promptly removed from the heap. You can also manually invoke GC. It's not necessary to do that since GC works automatically, but it's useful for those times when you find an object that needs to be destroyed immediately (such as an object that needs to be reset by destroying it and recreating it immediately). You can manually invoke GC as follows:

```
System.GC.Collect()
```

This code automatically kicks in GC and has it run through its chores. However, it eventually creates overhead if used repeatedly, so it's best to use it sparingly. Roots also provide the fix to memory leaks and stray resources. The runtime uses the roots to determine when an object or resource is no longer in use, enabling GC to clean them up.

Now that we know how GC works, let's look at just what the GC namespace offers (Table 1.6).

Table 1.6 The GC Namespace

Property/ Method Type	Method	Description
Properties—public static	*Max Generation*	Lists the generations that the system can support.
	Total Memory	This method displays the total byte space of alive objects, and can occasionally overlap objects that will be released soon. This method is used frequently for high-usage areas, especially the areas that contain expensive and/or limited resources, such as CE.
Methods—public static	*Collect*	An example of an overloaded method; it forces a collection of all available generations. Can be useful in building your own garbage collection system for your particular application by analyzing available generations. You can then use this information to force any objects into a disposal.
	Get Generation	Another overloaded method; it returns the specific generation that an object is in.
	KeepAlive	A method that assists in migrating VB 6.0 code to VB.NET. Using *KeepAlive*, you can tell GC that this object does not get recycled, even if there are no roots to it from the rest of the managed code, by sending GC a "fake" alive response.
	Request Finalize OnShutdown	This method is an implemented workaround to a bug in the beta1 Framework; the .EXE engine usually shut downs without calling a finalize routine. This method causes all finalization that needs to be done on shutdown.

Continued

Table 1.6 Continued

Methods—public static	*Suppress Finalize*	This method simply tells the system to not finalize an object. Very useful for helping GC "skip" prefinalized objects (objects that have been manually finalized), and thus keeps GC from wasting time on something that's not there.
	WaitFor Pending Finalizers	A really buggy implementation of a good idea. This method suspends the current running thread until all finalizers in the queue are run. However, since running a finalizer almost always kicks in a GC, this method causes a circular loop that will keep waiting for finalizers as new finalizers are created. This method would be much more useful if it could target generations instead.
Methods—public instance (all these methods are inherited from the *System .Object* namespace)	*Equals*	Checks to see if the object being evaluated is the same instance as the current object.
	GetHash Code	Returns the hash function for a specific type.
	GetType	Returns the type from an object.
	ToString	Returns a string to represent the object.
Methods—protected instance (all these methods are inherited from *System.Object* namespace)	*Finalize*	Allows cleanup before GC gets to it. However, the CLR can decide to ignore this command, as when the root is still active or it's considered a constantly used resource.
	Memberwise Clone	Creates a copy of the current object's members.

We can use the methods and properties inherent to the Garbage Collection namespace to formulate a workaround to GC having full control over the disposal of objects. (Remember, the runtime controls the memory allocation through Garbage Collection; that includes the destruction of objects.) The following is an example of this code:

```
Imports System

'class/module/assembly code here to do whatever you want
'please note that this is just an example and is non-functioning.
' there is a very good functional example of this similar process
' available in the .NET SDK Samples under the GC/VB folder of the
' SAMPLES directory.

'now that we have the objects / resources set, let's create a typical
' Dispose class.

Public Class DisposeMe
Inherits Object

   Public Sub Dispose(objName as String)
   'objName would be received by previously using the
   '  ToString Public Instance Method and storing the value in a
string.

   Finalize

   GC.SuppressFinalize(objName)
   End Sub

   Protected Overrides Sub Finalize()
    ' no clean-up code needed; this will cause Finalize to be ran
   End Sub
End Class


'note the use of SuppressFinalize to keep the GC from repeating itself.
```

Congratulations! We've just worked around one of the basic problems of GC. With this example, we can successfully control manual termination of objects and resources. It's best to reserve this type of workaround for intensive resources.

Debugging…

Don't Use a Raw Finalize Method!

GC allows a small emulation of the *Class_Terminate* event via the *finalize method*. However, the finalize method does *not* supercede the authority of the GC/CLR, and it might not be instantly implemented if the GC/CLR assume that the resource/object is still needed or in use. It could very well be a couple of calls too late before it's shut down, which is especially frustrating when you need to remove an object for program flow. Finalized objects:

- Are promoted to older generations, causing unnecessary heap usage.
- Have longer initialization times.
- Are out of your control as to when and where they are actually terminated.
- Cause any other objects that are associated with them to be finalized, adding more strain to the heap.
- Can prolong the lifetime of other objects that are referenced from the finalized object.

For these reasons, it is better to avoid using finalize alone. If you determine that you must use it, make sure that you avoid all actions that could interfere with the finalize code, such as creating an instance of the finalized object after you run the finalize method, thread synchronization operations, and any exceptions from the finalize method.

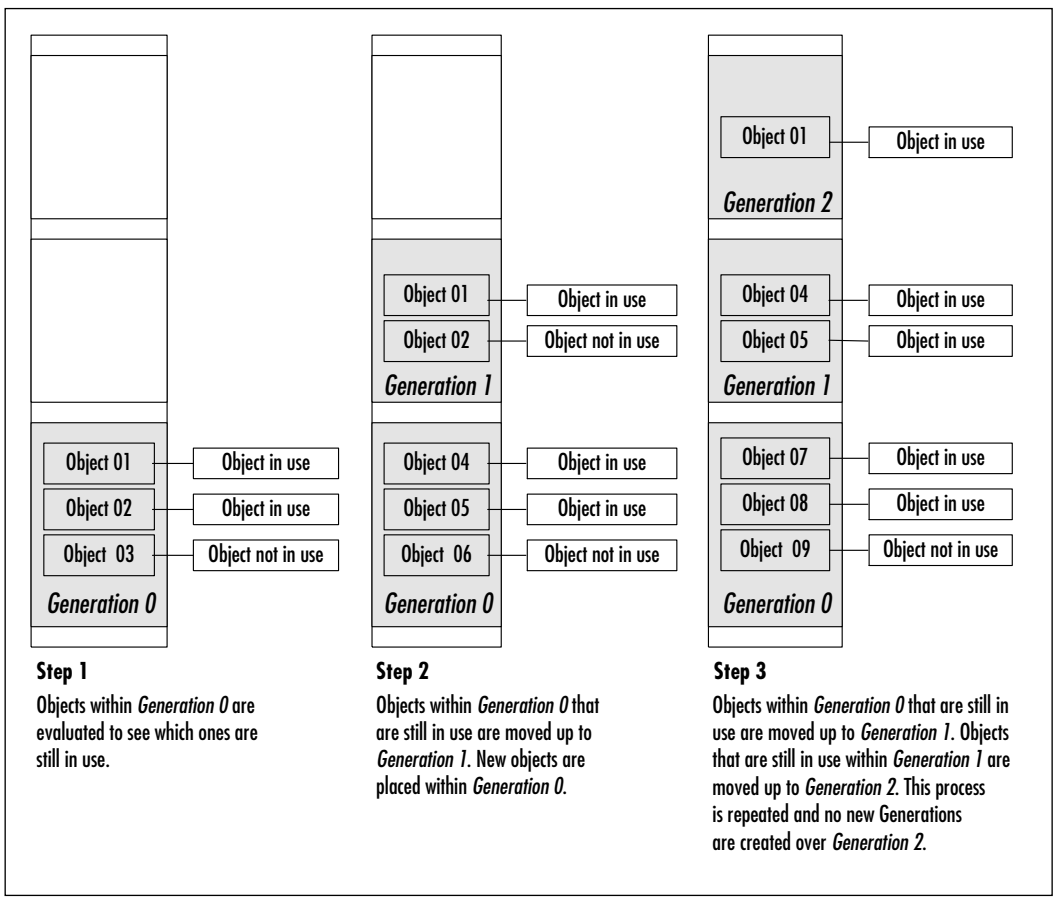
Resurrection is a side effect of finalization. Sometimes we'll be presented with a situation in which an object has been finalized but there is still a pointer to it, meaning that GC assumes it's alive when it's been already finalized. A typical scenario is to finalize an object in order to create a new instance of the same object; if the first object is still there in finalization, the pointer points to the old object, and the object, while in finalized stage, never gets cleaned out properly because it has a reference from the application. It's important that if you finalize something, you set a flag or a check routine to make sure that it's gone before you try to do anything else concerning that object type.

Assigning Generations

GC uses an ephemeral garbage collector, which describes the lifetime of an object in generations. Using this system, the garbage collector makes the following logical assumptions:

- Newer objects have shorter lifetimes.
- Older objects have longer lifetimes.
- Newer objects are created around the same time and have strong relationships.
- Compacting a portion of the heap is faster than compacting it completely.

Figure 1.7 Generations



Let's look at a new heap. Once the heap is created and the first set of objects is instanced, the objects are created and set as *Generation 0*. As a new set of objects is created, GC checks to see which objects from *Generation 0* still exist (Step 1 in Figure 1.7). Those that do exist are compacted, moved above *Generation 0*, and become *Generation 1* (Step 2). As the new *Generation 0* enters the same process, so does *Generation 1*. Any remaining members of *Generation 1* become *Generation 2*, and those that survived *Generation 0* become 1 (Step 3). Then, the new *Generation 0* is created. At this point, the process continues, but there can be no higher generation than 2; any survivors from any subsequent *Generation 1* members are placed in *Generation 2* with the previous *Generation 1* members that survived. This also means that a complete heap compacts portions at a time, thus increasing overall speed.

Objects within *Generation 0* are checked more frequently than the other two generations because of .NET's philosophy that new objects are more likely to be the first to be removed. In other words, the longer an object is alive, the more likely it is to stay alive.

Using Weak References

Another innovation that stems from the roots concept is *weak references*; a weak reference is a weak link to an object in memory that has been or is in the finalization process. It "acts" like a root and will be collected by GC the next time it runs. A *strong reference*, on the other hand, represents the primary object creation. Without a strong reference, you can't really create a weak one.

Weak references can provide a workaround when you are dealing with memory-intensive objects, and avoid the cost of constantly recreating and reinitializing objects. Imagine an object that traverses a database and stores a set of sorted fields. If the database is small enough, it can rest in memory without problem. However, if the database is large, we run the risk of tanking out our resources every time we have to create a new one. Using a weak reference, we can bypass having to create a new object and redoing the sort by keeping the items we need on standby. You can then recreate the strong reference by pointing to the weak reference.

Security Services

Security services are not to be confused with the concept of security as offered by .NET. Security services provide a type of check and balance within code, metadata, and MSIL in order to make sure that the CLR gets what it expects, that it's getting it through either the same developer or a trusted source, and that future references to items usually denied access to due to isolation can be granted access.

In .NET, the Virtual Execution System (VES) handles all the security checking. The still unreleased *Overview of CLR Runtime Security* spec sheet from Microsoft has all the details. This spec sheet will contain more nuts and bolts about how the VES controls security checking. We do know, however, that the VES provides the security services commonly used within the CLR.

Type safety is enforced through the VES by matching the same strong types in metadata with the corresponding MSIL (local variables and stack slots). You can look at it as a technical diagram; it draws a very strong line pointing from the metadata to the MSIL, and makes sure that everything matches up to the correct declaration and memory space.

The VES also covers versioning safety. Since the VES does the job of lining everything up, it also goes ahead and sees that all the information that's being checked also passes the version check. The VES also makes sure that the CLR will "see what it gets"; in other words, that the CLR will work within the assumptions it made about the code.

However, in order to make an assumption about the code, the CLR must be sure that the code is a proper executable. Again, the VES steps in by providing the only three methods that a code can use to become executable: *class loader*, *legacy-code-based platform invoke*, and, for migration purposes, an *unmanaged COM interop*. Using the legacy-code platform invoke and the unmanaged COM interop can cause some performance issues, so it's best to avoid them altogether when writing or migrating code and to stick to the class loader. The class loader connects implementations to the information about the implementation within a metadata. The VES uses the class loader to also determine who is trying to access a type, and thus takes the advantage to determine accessibility.

In addition, the VES has access, through the CTS, to the permissions that are stored within metadata to access methods. It checks each type against the permissions, and marks each type that has permission with a stub in the loader (the JIT and the linker also use VES to do the same) that tells the CLR to enforce the permissions to which the stub points. This is called *declarative security*.

NOTE

Although the CLR is impressive in terms of detection algorithms, it has a drawback in that it's still, in the end, simply a logical system. It can't tell when someone might trick it (although the CLR is very stringent, thus making it hard to trick). To prevent that, we can use *imperative security*; that is, we can set the rules in our code.

Framework Security

Code access security and *role-based security* are the two types of security provided by the .NET framework. They are mechanisms that are geared toward a "keep it simple" mentality regarding how to decide what a user can do. The keep it simple idea trickles down to a simple model with consistency, providing easy transitions from code-based to role-based security and back. The fundamentals that give the .NET security its robustness are *permission*, *principals*, and *security policy*.

Code access security, as you might have noticed, provides varying degrees of trust for an application. It can change these degrees according to the information that the assembly provides—such as developer, version, and the like—since this information is stored on the code. When the process of determining if a particular code can access, the runtime checks the current call stack of the code looking for the permission; if it can't find permission, it throws an exception.

Role-based security makes an authoritative decision based on the principal value from the current thread making the request. The role(s) listed within the principal value are then evaluated, and the action/ability requested is given or denied.

Financial software programmers and database coders might already be familiar with the concept of role-based security. Usually, in these situations, when a client requests access to a certain part of the system or resource, a check is run to determine from what role the client making the request comes. Let's say that a member of the group Alpha is trying to access a resource located with a member of the Omega group. Alpha starts the connection, and Omega picks off the first principal from the connection thread. The principal is then analyzed for roles, and Omega determines that the Alpha workgroup does not have permission for all the resources—just two of them. Omega allows the connection, but limits Alpha's request to the two resources. If Alpha tried to obtain a resource outside those two, the request would be denied.

Granting Permissions

Permission is the basic building block of security. Some view permission logically as a response given to a query in order to gain access; others look at it as a key fitting into a lock. Both views are equally correct. Permissions in .NET are used via requests, grants, and demands.

A code can *request* permissions to see if it can access a file. If it doesn't fall under those permissions, you could have a function to *grant* permission to the

code that's making the request. If a code with the permissions ready comes along, you might want to implement an added layer of permission called *demand*. In other words, while the code might have the basic permissions needed in order to satisfy the need, the code can also *demand* that (a) specific permission(s) be present. Table 1.7 lists the permissions for both code access security and role-based security.

Table 1.7 Code Access Security and Role-Based Security Permission Lists

Code Access Security Permissions	Description
DnsPermission	Provides access to a Domain Name System.
EnvironmentPermission	Provides access to the ability of read/write/query environment variables. Write access also includes the ability to create, remove, and write.
FileDialogPermission	Provides access to files acquired via a file dialog box.
FileIOPermission	Provides access to perform low-level (through stream) read, write, append, or create directories.
IsolatedStoragePermission	Provides access to an area that is attributed to a specific user within a part of the code identity.
ReflectionPermission	Used in conjunction with *System.Reflection* to have permission to find out information about a type at runtime.
RegistryPermission	Provides access to the Registry and the read, write, create, delete Registry functions; applies to keys and values. If you truly want to make people who use your .NET code happy, use the .NET and don't use the Registry anymore. This permission is really more of a migration step.
SecurityPermission	Provides the ability to do actions that are normally not allowed, such as calling into unmanaged code and skipping the verification process. Use this with caution; it can lead to holes in your system that can be used to access other parts of it.

Continued

Table 1.7 Continued

Code Access Security Permissions	Description
SocketPermission	Doesn't really grant any ability; either accepts or creates any attempted connections at a given transport address. Using this permission in conjunction with *SecurityPermission* for executables can cause some bad things to happen.
UIPermission	Provides the ability to use the functionality provided by the user interface.
WebPermission	Just like *SocketPermission*, it either accepts or creates any attempted connections from/to a Web address.

Role-Based Security Permissions	Description
PrincipalPermission	Demands that the identity of an active principal match. (See the *Principal* section for more information.)

Gaining Representation through a Principal

Have you ever wanted a go-between (and I don't mean a lawyer) to plead your case to the program to get access? A principal provides just that function. Depending on the situation, a principal provides the permission level needed on your behalf to enter. The CLR lets the principal in, but it's not letting *you* in; the CLR only allows you to do what the principal is supposed to.

A generic principal is your run-of-the-mill representation that you can use to find out what someone who is not unauthenticated can "see." Although this is not practical in an everyday program, it is very useful for testing and debugging situations, and is extremely helpful when trying to determine situations in which a permission shows up that you didn't plan for.

Custom principals are created on-the-fly by an application to suit a current need or requirement. They extend the basic usability of a generic principal, but are dependent on having the proper authentication modules and types given to them by the application. This dependency gives the custom principal an element of security, since it can't work without being given what it needs to work.

NOTE

A special class of principal—the Windows Principal—represents strictly Windows users. It uses this impersonation to get roles that are available for that particular user.

Security Policy

The rules that the CLR follows are referred to collectively as the *security policy*. The local administrator determines these configurable rules. Once an assembly is attempting to load, the security policy is checked to see what permissions the CLR can grant the assembly. It determines various possibilities and then, if it passes, provides the needed permissions or simply does not allow the program to run.

Three levels specify security policy: The local machine policy, the application domain policy, and the user policy. The runtime uses all three of these policies to filter out the final security policy that will be placed on the assembly and thus determines its permissions. Both the user and the application domain policy specify the set of permissions that are allowed, and then this set of permissions is compared to the machine policy. The permissions that are not filtered out become the security policy.

Application Domains

An application in .NET runs in a domain that's managed by a host. This host can be a shell host (launches .EXEs from a shell), a browser host (runs code from the site), a server host (ASP.NET; runs code that handles requests on a server), and a custom-defined host. When one of these creates the application domain, for example, the shell host—which would be Windows—sets the policy that the code must deal with under that domain. The policy generated cannot be added to, but can be made more flexible by the host.

After an application domain policy is set, the new policy applies only to assemblies that are loaded after the creation of the new policy. Any previous policyholders will have their previous policy covered and won't have to use the new one unless reloaded. Once the main assembly is loaded and the first reference to another assembly is made, the loader kicks in, places the assembly into the appropriate application domain, and then returns the information (referred to as

evidence) that proves it can be trusted (will return versioning information to verify) to the runtime. Table 1.8 lists the evidence that is/can be returned.

Table 1.8 Evidence

Application Directory	Where the Application Resides
Custom	Evidence created by the user or system defined; great for making 100-percent that sure it's the correct evidence.
Hash	Returns the hash encrypted in MD5 or SHA1.
Publisher	The AuthentiCode signature provided by the code.
Site	Location of origin.
Strong Name	Assembly's strong name.
URL	URL of origin.
Zone	Zone of origin; for example, *Internet Zone*. Matches the zones listed in your Properties box for IE under the Security tab.

Summary

The .NET platform is a great leap forward in the evolution of computing from PCs connected to servers through networks such as the Internet, to one where all manner of smart devices, computers, and services work together to provide a richer user experience. The .NET platform is Microsoft's vision of how the developers of this new breed of software will approach the challenges this change will provide.

If some of the .NET concepts sound familiar, there's a good reason: the .NET platform is the next generation of what was called Windows DNA. However, while Windows DNA did offer some of the building blocks for creating robust, scalable, distributed systems, it generally had little substance in and of itself, whereas .NET actually has an integrated, comprehensive design and well-conceived, usable tools.

The components at the heart of the .NET platform are the Common Language Runtime (CLR), the Base Class Library (BCL), and the Common Language Specification (CLS). The .NET BCL exposes the features of the CLR in much the same way that the Windows API allows you to use the features of the Windows operating system; however, it also provides many higher-level features that facilitate code reuse. The CLS gives language vendors and compiler developers the base requirements for creating code that targets the .NET CLR, making it much easier to implement portions of your application using the language that's best suited for it. The .NET platform allows languages to be integrated with one another by specifying the use of the Microsoft Intermediate Language (MSIL, or just IL) as the output for all programming languages targeting the platform. This intermediate language is CPU-independent, and much higher level than most machine languages.

Automatic resource management is one of the most discussed features of the .NET platform, and for good reason: countless man hours have been spent chasing problems introduced by poor memory management. Thanks to the managed heap memory allocator and automatic garbage collection, the developer is now relieved of this tedious task and can concentrate on the problem to be solved, rather than on housekeeping. When an allocated object is no longer needed by the program, it will be automatically be cleaned up and the memory will be placed back in the managed heap as available for use.

Once written and built, a managed .NET application can execute on any platform that supports the .NET CLR. Since the .NET CTS defines the size of the base data types that are available to .NET applications, and applications run

within the CLR environment, the application developer is insulated from the specifics of any hardware or operating system that supports the .NET platform. While currently this means only Microsoft Windows family of operating systems, work is underway to make the .NET core components available on FreeBSD and Linux.

The .NET architecture now separates application components so that an app always loads the components with which it was built and tested. If the application runs after installation, then the application should always run. This is done with assemblies, which are .NET-packaged components. Assemblies contain version information that the .NET CLR uses to ensure that an application will load the components from which it was built. Installing a new version of an assembly does not overwrite the previous version, thanks to the Assembly cache, a specialized container (directory) that stores a system's installed .NET components.

Given the massive amount of legacy code in use, it was necessary to allow .NET applications to interact with unmanaged code. As you can probably guess, unmanaged code is code that isn't managed by the .NET CLR. However, this code is still run by the CLR; it just doesn't get the advantages that it offers, such as the CTS and Automatic Memory Management. There are a couple of times when you will probably end up using unmanaged code, making API or other DLL calls, interfacing with COM components, or allowing COM components to utilize .NET components. However, realize that by calling unmanaged code, you might be giving up portability!

Developing software using .NET technology is a big change; there are a lot of pieces to the puzzle, and more than a few new ideas. Hopefully, we have given you a solid introduction to the basics, and you now have a foundation upon which to build your skills using the information found in the rest of the book. If you want more detail on a particular feature of the platform, the MSDN Web site contains a vast amount of reference material that covers the features of the .NET platform at a much more technical level than we attempted here.

Solutions Fast Track

What Is the .NET Framework?

- ☑ .NET provides developers with new possibilities on creating applications.

Introduction to the Common Language Runtime

- ☑ The CLR is the heart of the .NET Framework. It provides much of the functionality that .NET uses.
- ☑ CLR will provide the function of translating the application from its internal code to code within the native environment.
- ☑ Managed code will be able to get the most out of the new .NET features from the CLR.

Using .NET-Compliant Programming Languages

- ☑ Programming for .NET is not limited to the Microsoft standard languages. Any compiler that follows the CTS and other requirements for .NET can be created for any programming language.
- ☑ .NET's new interoperability allows us to use each language's strengths to counteract weak areas.
- ☑ Different programming languages will have the same method of communication within each other, thus ensuring true interoperability.

Creating Assemblies

- ☑ The new deployable unit for .NET is an assembly. It is more like a logical DLL file than a true executable file.
- ☑ All the information that the CLR needs to properly run an assembly is located within the assembly itself.
- ☑ Each assembly file consists of the internal code, the manifest area, and the metadata contained within the manifest area.

Understanding Metadata

- ☑ Metadata contains the map that .NET uses to layout objects in memory and how they are used.
- ☑ The manifest area within the assembly contains the metadata.

Using System Services

- ☑ More control is given to exception handling through the try/catch system.

- ☑ The automatic resource management system for .NET is "smart" enough to know when objects are in use and when they need to be removed. This takes the burden off the programmer, but the programmer can always opt to declare when an object should be removed.

Microsoft Intermediate Language

- ☑ MSIL is the bytecode that the just in time (JIT) compiler uses to create native code for the assembly file.
- ☑ MSIL is platform independent.
- ☑ The code within a .NET application is converted to MSIL.

Using the Namespace System to Organize Classes

- ☑ A namespace provides an organizational hierarchical system for classes.
- ☑ Each class that specifies to a specific function is stored within its respective namespace.
- ☑ The System namespace is the root namespace of all namespaces in .NET.

The Common Type System

- ☑ The Common Type System is the way in which types are supported within the runtime.
- ☑ The CTS also specifies how types can interact with each other, and how they are displayed as metadata.
- ☑ The CTS provides the rules that types must follow in order to work with .NET.

Relying on Auto Resource Management

- ☑ The managed heap system replaces the reference count system.
- ☑ The object cleanup is referred to as Garbage Collection. .NET controls when Garbage Collection runs and when an object is removed.
- ☑ The burden of object cleanup is placed more within .NET than on the developer.

Security Services

- ☑ Permissions are the rights needed to use a resource. There are many different types of permissions that can be used in any event and are primarily used within code access security.
- ☑ The principal acts as a go-between for you to get the permissions needed. There is only one type of principal. Principals are used within role-based 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** and click on the **"Ask the Author"** form.

Q: If any .NET language has access to everything in the Base Class Library, why is there so much talk about C#?

A: While in theory all .NET languages have equal access to the BCL, in reality it was left up to the language teams to determine what level of support they wanted to offer, at least beyond the minimum needed for basic compliance. In our opinion, C#, has received much attention, because many believe it was the only language specifically designed for .NET.

Q: Where can I find information on COM/CLR compatibility issues?

A: Visit http://msdn.microsoft.com/library/dotnet/cpapndx/_cor_appendix_e___known_compatibility_issues.htm.

Q: Do I have to use Visual Studio.net or a Microsoft-endorsed editor to create my VB.NET files?

A: No. With the implementation of VBC.EXE, you can use any editor you want to write the code, without suffering any bugs or problems.

Q: Is it better to learn and rewrite my existing VB / C++ applications in VB.NET / C#, or to make the necessary changes to my VB / C++ application to run on .NET?

A: That's a subject of debate. It all depends on the size of your code. Naturally, smaller programs will be easier to convert to .NET; even if you do convert to .NET, you might still miss out on the advantages .NET has over VB / C++. On the other hand, learning and rewriting a complete program in .NET can be time consuming. Keep these considerations in mind when deciding what you should convert and what you should rewrite.

Q: Is everything in the Win32 API exposed through the BCL?

A: Not through the BCL, but you can make API calls directly through most languages.

Q: Isn't the fact that .NET applications aren't native code going to increase PC requirements?

A: This depends on what type of application you're developing, but it's a pretty safe bet. There will be additional memory requirements introduced by the managed environment, but they will be negligible in practice. Every new development in software engineering has required more horsepower, and we're really not taxing today's processors with most software. Buying more memory if it is required should be a simple sale; developer man hours are generally much more expensive than more memory is.

Q: Where can I get more information about the .NET architecture feature X?

A: A search engine such as Google might help depending on what you're looking for, but start with MSDN online. Many white papers and various magazine articles about .NET can be found there.

Chapter 2

Visual Studio.NET IDE

Solutions in this chapter:

- Introducing Visual Studio.NET
- Components of VS.NET
- Features of VS.NET
- Customizing the IDE
- Creating a Project

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

In addition to the powerful .NET platform, Microsoft has introduced a new version of its Visual Studio Suite, called Visual Studio.NET (VS.NET). Even in its Beta stages, VS.NET provides the developer with powerful visual tools for developing all types of applications on the .NET platform.

VS.NET helps in the speedy creation and deployment of applications coded in any of the managed languages, including C#. This chapter gets you familiar with the new features of VS.NET and teaches you to customize it according to your needs. We cover the many new features of VS.NET, including the .NET Framework, Web Services, XML support, and the Integrated Development Environment (IDE).

We also cover the XML editor, which has tag completion for Extensible Stylesheet Language Transformations (XSLTs). We go over the IntelliSense feature and how it is used in the different windows. Finally, we cover how to customize your settings within the IDE.

VS.NET is a complete development environment. The components stay the same regardless of language, making it very easy to switch projects and languages and have the same features in the same place. Moreover, with the expanded IntelliSense with tag completion, routine code writing is faster.

NOTE

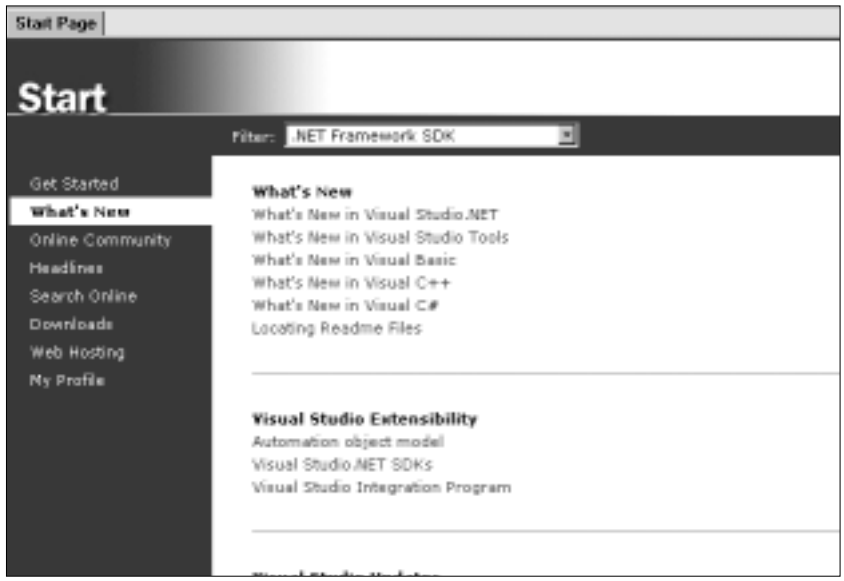
VS.NET, while a great way to work with XML.NET, VB.NET, or C#, has a hefty price tag of $1,000 for just the basic IDE, while the Enterprise version can scale up to $10,000 per license. Just remember that VS.NET is simply an IDE—any code writing or editing can still be performed with Notepad.

Introducing Visual Studio.NET

The Start pages deliver a great many resources for the development environment. The Start page is the default homepage for the browser inside of the IDE. You can tap all aspects of the IDE from these pages. We go over the three most useful Start pages, beginning with the "What's New" Start page and the "My Profile" page, and ending with the "Get Started" Start page. We show you what is new with VS.NET, set up your profile, and get started using the tool.

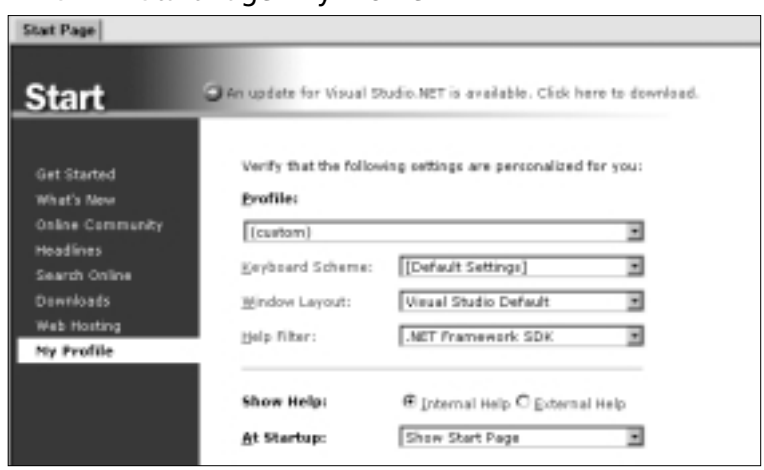
Let's open VS.NET and look at the first of the Start pages (Figure 2.1).

Figure 2.1 VS.NET Start Page: What's New



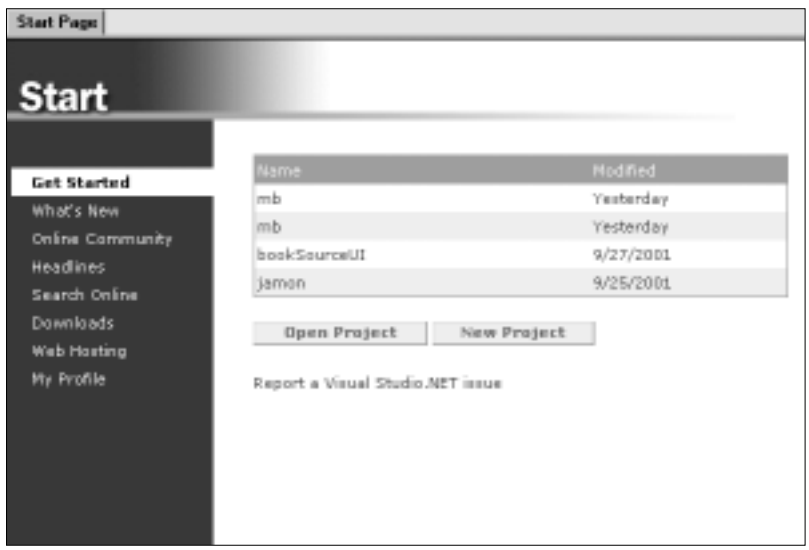
You can filter the "What's New" Start page to whatever topic you are interested in—we have chosen to filter by .NET Framework. All content in the "What's New" Start page will be rendered based on the filter, so you can save some time by not looking up new features for VB, for example. You can also select **Check availability of Visual Studio.NET service packs** from this Start page to see if you need the latest download for VS.NET. Let's look at the "My Profile" Start page next, shown in Figure 2.2.

Figure 2.2 VS.NET Start Page: My Profile



The "My Profile" section of the Start page lets you create your own (custom) profile or select from any of the options listed. If you happen to come from a VB background, using the VB profile would be beneficial so that you could be familiar with the tools from VS 6. Likewise, a C++ or Interdev user from VS 6 will benefit from the same environment. This will help you to learn the tool by showing a familiar layout. You can also select to have only external help, which will open the Help menu in a new window available outside of the IDE. You can filter the Help topics; in our case, we've selected **.NET Framework SDK** in the **What's New** section Start page. You can also select the window layout that you want to use. You then can select the **Get Started** Start page, shown in Figure 2.3.

Figure 2.3 VS.NET Start Page: Get Started



Here you can select projects you worked on previously, and you can also see where they are located on the machine by dragging the mouse over the name of the file. This is a nice feature that you can use when you have two projects named the same but at different locations.

The Start page is the default page for the Web browser window in VS.NET, so if you close it and want to get it back, simply click the **Home** icon on the Web toolbar and the page will load in the design window.

Components of VS.NET

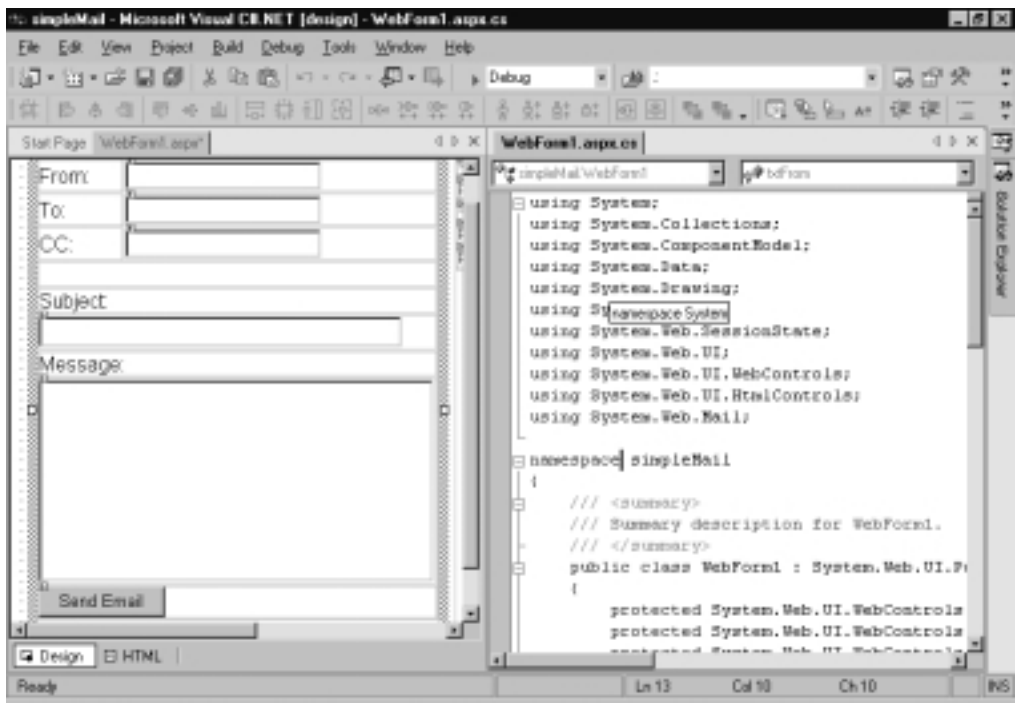
The Visual Studio.NET IDE is made up of many components that interact with one another. You can use each separately or at the same time. This feature lets the

user decide which set of components he wishes to use. All of the components together create an intuitive graphical user interface (GUI).

Design Window

The design window is used when you are graphically creating an application using the Toolbox to drag and drop objects onto the window. Much like the code window and browser, the design window cannot be docked or set to Auto Hide. You can split the design view or have tab groups added to it. Splitting the window helps when you need to compare code from two separate files (Figure 2.4).

Figure 2.4 Split Window View



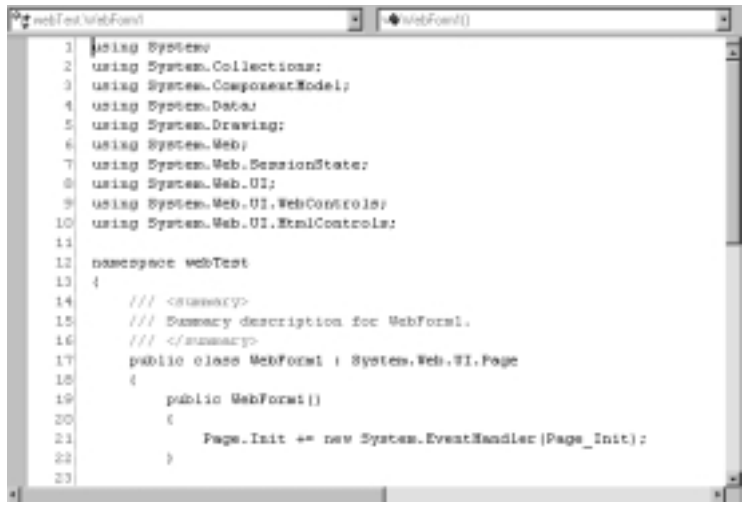
Here you can see windows for both design and code. This is a C# Web application, but the system is the same for any project.

Code Window

As we mentioned, the code window is much like the design window. There is no toolbox functionality within the code view, however—you cannot drag and drop objects from the toolbox and into the code view. If you create objects in the code view and then switch back to the design view, the objects that you added

will appear. Again, you cannot dock this window or allow it to float. You can, however, split it and add new tab groups to the display. Figure 2.5 shows the code window split and a tab vertical tab order added.

Figure 2.5 Code View



If you look at Figure 2.5 a little more closely, you can see a collapsible menu tree on the left-hand side. This is created every time you create a class or function, enabling you to collapse each section independently to save space for viewing other code present within the window. Note that you must have the default option Outlining Mode checked for this to be present. If you want to have line numbers show for your code, you will have to choose **Tools | Options**. In the **Options** dialog box, select **Text/Editors**. Select **C#**, and then choose the option to have line numbers added.

You can also define your own regions of code that might be collapsed. To do this, simply add the following code to your class or function you want to make into a region:

```
#region
        ///Comments
#endregion
```

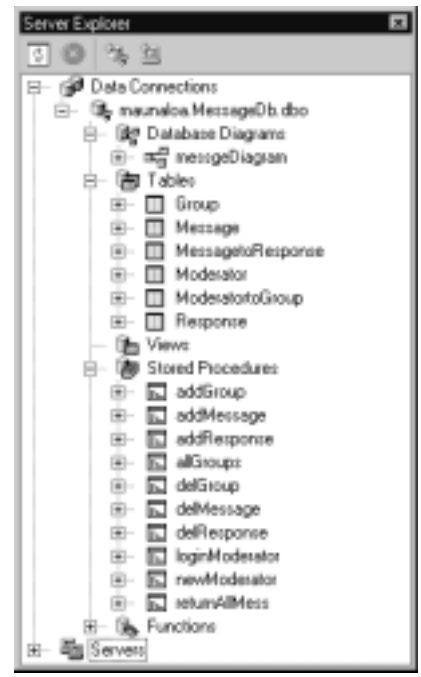
Server Explorer

The Server Explorer is by far one of the best features in VS.NET. From this window you can connect to a server on the network and have full access to that

server or servers. You can also link to any database servers on the network. Let's see how to do that. Click the **Connect to Database** icon in the title bar of the window (Figure 2.6). You will be prompted to give all information required for a Universal Data Link (UDL).

Figure 2.6 Add Database to Server Explorer

Figure 2.7 Expanded Database View



Fill out the UDL Wizard and test the connection. After this is done, you can access everything within that database that the user has rights to. Let's take a look at that in Figure 2.7.

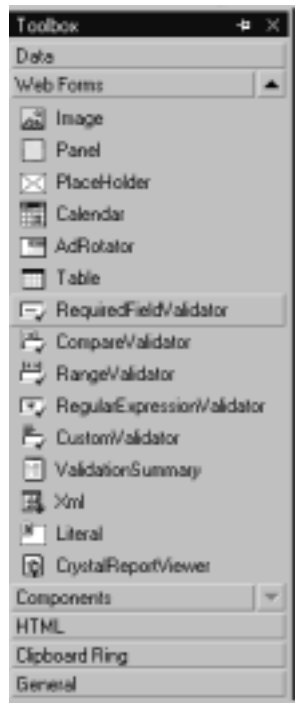
You can now click on any object within the Server Explorer and edit it within VS.NET. This is a timesaver from having to have both the Query Analyzer and VS open at the same time, and going back and forth between the two just to switch a data type of one stored procedure input parameter.

Toolbox

The Toolbox, shown in Figure 2.8, includes Data, Components, Web Forms, and Window Forms. As stated earlier in the chapter, you can use the Toolbox with the Design View window. You can drag and drop each tool onto the design

window. In addition, you can customize the Toolbox by adding your own code fragments and renaming them to something meaningful.

Figure 2.8 The Toolbox Window



To do this, simply right-click on the **Toolbox** and select **Add Tab**. Give it a name that is different from the existing tabs, and you are ready to add your own tools. To add a new tool, highlight a block of code that you want to make into a tool, and drag it onto the Toolbox label you just created.

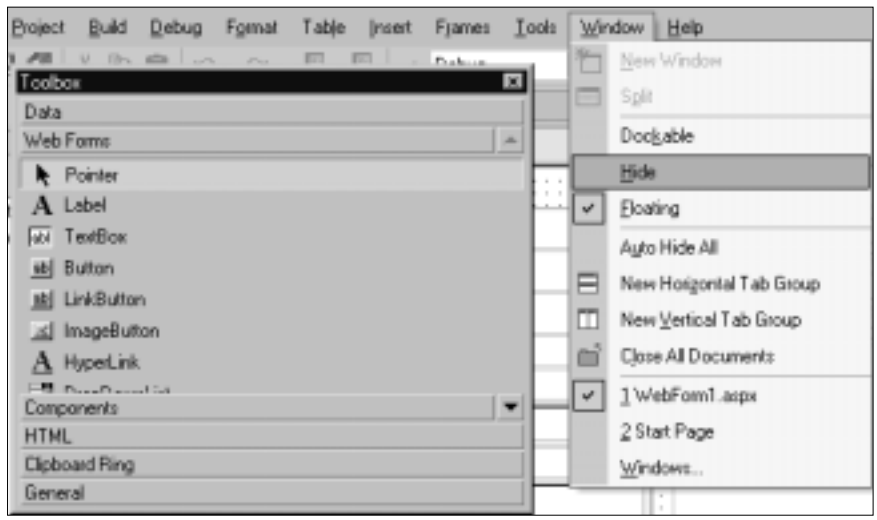
The Clipboard Ring stores all the items that you have copied. You can then double-click these and add them to the source code.

Docking Windows

One of the new features for VS.NET is that you can dock or expand or collapse all the windows within the view of the IDE. To add windows to your IDE, navigate to the standard toolbar and select **View**; here you can select all the windows that you want to have immediately available in your environment. One drawback to this is that you will not have much room left in which to work if you select a lot of windows to show, but the Auto Hide feature of each

window makes them slide off the screen and embed in the side when not needed. This enables you to have maximum code view, but still have all windows present. To see a window that has Auto Hide enabled, simply position your mouse over the **Window** icon on either side of the IDE. You can dock each window into place by clicking on the pin or by navigating to the standard toolbar and choosing the **Window** menu option. Once a window is docked, it is there permanently; you can, however, make the window float by selecting **Window | Floating** (Figure 2.9).

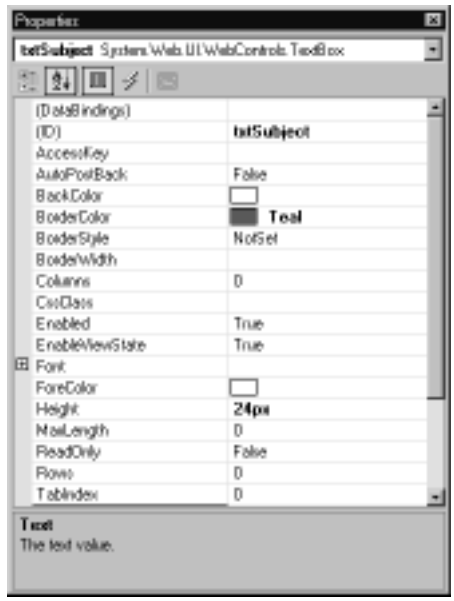
Figure 2.9 Floating Window



Properties Explorer

The Properties Explorer is much as it was in VS 6 and the Visual Basic IDE and Visual Interdev IDE. Select an object from the design window, and in the Properties Explorer, you will see available attributes for that object listed, as shown in Figure 2.10. The right-hand column lists the property names, and the left-hand column stores the attribute's value. The Properties window enables Rapid Application Development (RAD) by allowing you to quickly create a graphical representation of the application you are building without doing any coding whatsoever. Some options are available in the Properties Explorer. You can select from the drop-down list the actual object you want to view. You can also select the **Events** option and have the event available to that object displayed. You can organize the Properties Explorer either by categories or alphabetically.

Figure 2.10 Properties Explorer



Any changes made in this window will be propagated to the design view and code view windows, respectively.

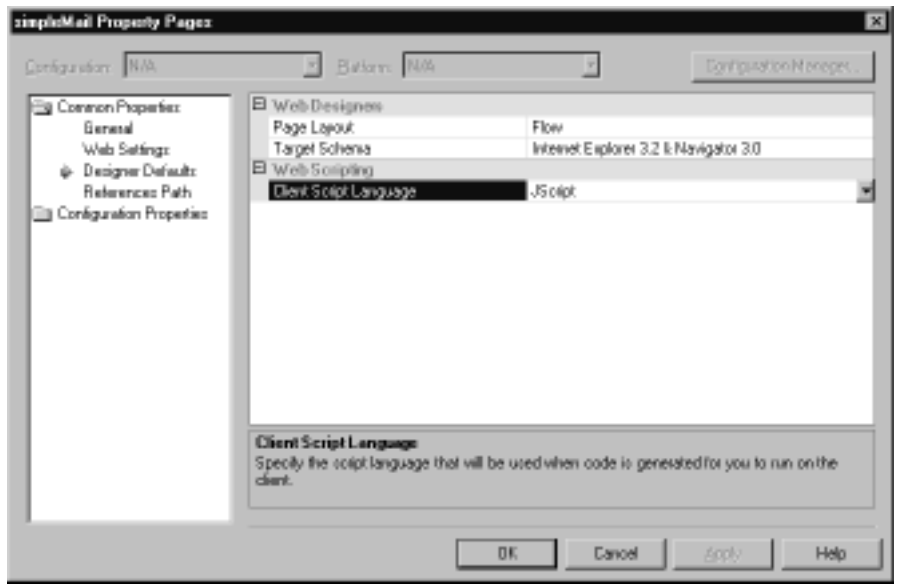
Solution Explorer

The Solution Explorer is the same as it was in VS 6. The Explorer is a look at all the files in your solution. In the title menu bar, you have four options: Refresh, Copy Web, Show All Files, and Properties. The Properties option lets you set all of your solutions' properties, including debug parameters options. The .NET IDE has two different types of containers available for holding items: *solutions* and *projects*. The main difference between the two is that you can have multiple projects within a solution, whereas the project container keeps only files and items within files. Let's look at this in more detail in Figure 2.11.

Here, you need to make two changes. Set the target schema to **Internet Explorer 3.2 & Navigator 3.0**. In addition, change the page layout from **Grid** to **Flow**—this is from a C# Web application. These two changes will make all the JavaScript comply with the selected browsers. This will enable you to code without having to check to make sure if your scripts will work in older browsers.

By making the change to *Flow layout*, you prevent your code from using absolute positioning within span tags so that it will be safe for Netscape users. These two changes are useful for any ASP.NET development you might do inside of the .NET IDE.

Figure 2.11 Solution Properties

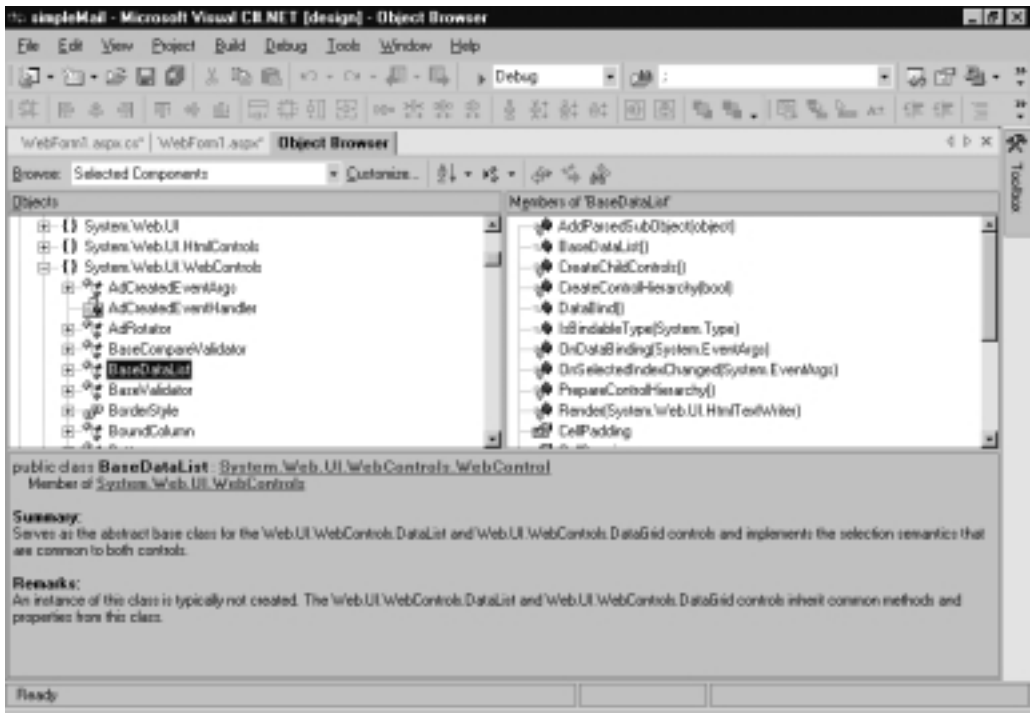


Class View

The Object Browser will give you a complete list of all classes' methods and properties in your solution. Everything is listed, and it is quite in depth. If you want, you can look up parents of classes that you are using and list out the methods and properties you might need. By double-clicking on an external class in your solution, the Object Browser will load and have all parent and child nodes of the class listed with each of their methods and properties included. This comes in handy when you need to find a suitable substitute class to handle some part of your application. As in Java, .NET has an incredible quantity of built-in classes that can accomplish just about everything you might need—the trouble is finding their location and how to access their methods and properties. Using the Object Browser enables you to achieve this in a timely fashion (Figure 2.12).

From this window, you can quickly drill through a class that is not your own and see what methods and properties it has; you also will get a summary of what it does and how it is instantiated.

Figure 2.12 Object Browser



Dynamic Help

Dynamic Help is a dockable window just like the previous windows we discussed. To get Dynamic Help to appear, simply choose **Help | Dynamic Help**. You can then make the window float or Auto Hide. One thing to note is that each part of Help (Index, Contents, Search, Index Results, and Search Results) is a separate window, so if you undock them and make them all float, you will have quite a few windows appearing on the screen. One thing you can do is load all the Help windows into themselves, and a bottom tab order will appear inside the main Help window; you can then access all parts of Help from the same window (Figure 2.13).

To customize the Dynamic Help window, choose **Tools | Options**. In the Options dialog box, select **Environment** and then select **Dynamic Help**. Here you can specify what topics you want to have available and in what order. You can also specify how many links are displayed per topic. In addition, you can create a custom Help file on your own for your project, by following the XML schema named vsdh.xsd. Create your XML file based off of that schema list and place the file where you want your Help topics to be displayed.

Figure 2.13 Docked Help Windows

Tabbing through the many different Help options and getting to the information you need is now easy. If you have the hard drive space, loading all the MSDN Help files from the disks that come with VS.NET would be beneficial. To do this, simply check the option on the installation sequence that will run from the computer and not the CD. This will prevent you from constantly having to load another disk every time you want to look up a particular topic. This gets quite annoying when you need one disk to open the tree view and another to access the topic within.

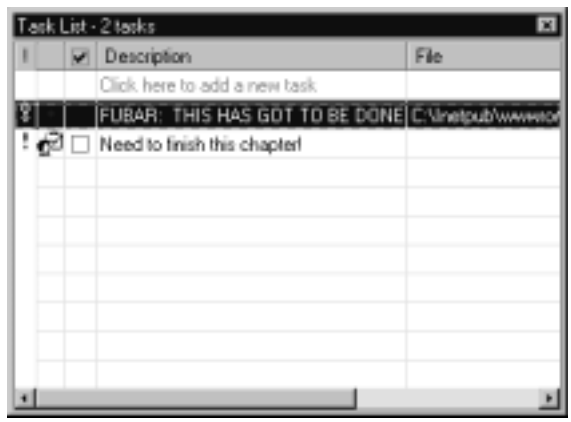
Task List Explorer

The Task List (Figure 2.14) enables you to add tasks that need to be done and organize them in a number of different ways and with priority. It is very simple to use. If you are using Source Safe, a group of developers can quickly see what needs to be done and what has been done by viewing the Task List for each file in the project.

Another feature of the Task List is that it will create tasks on-the-fly as you debug your application by marking down any errors. You can then go back and fix each task and have it removed. You can organize the task list on Build errors. In addition, you can create your own custom *token*, which is a unique key that

tells the Task List that a comment needs to be added to the list, to appear in your Task List from your code. You can map out your function or method or whatever you are coding with your own custom tokens and have them appear in the Task List.

Figure 2.14 Task List



To create your own custom token to add to the default tokens available (HACK, TODO, UNDONE), choose **Tools | Environment | Task List**. Give the token name and priority. To use the token, simply add something like the following in your code window (use the comment tag "//" and then the token name followed by the instruction for the task):

```
// FUBAR what I want in the task list to appear.
```

Features of VS.NET

VS.NET has a combination of new and old features built into the IDE. We discuss the additions to IntelliSense, the new features of XML support, and the many different ways you can now customize the IDE. Let's begin with IntelliSense.

IntelliSense

IntelliSense is a form of code completion that has been part of most Microsoft developer tools for many years. Code completion technology assists when you start to type a tag, attribute, or property by providing the resulting ending so that you will not have to write out the entire item. You will notice this right away.

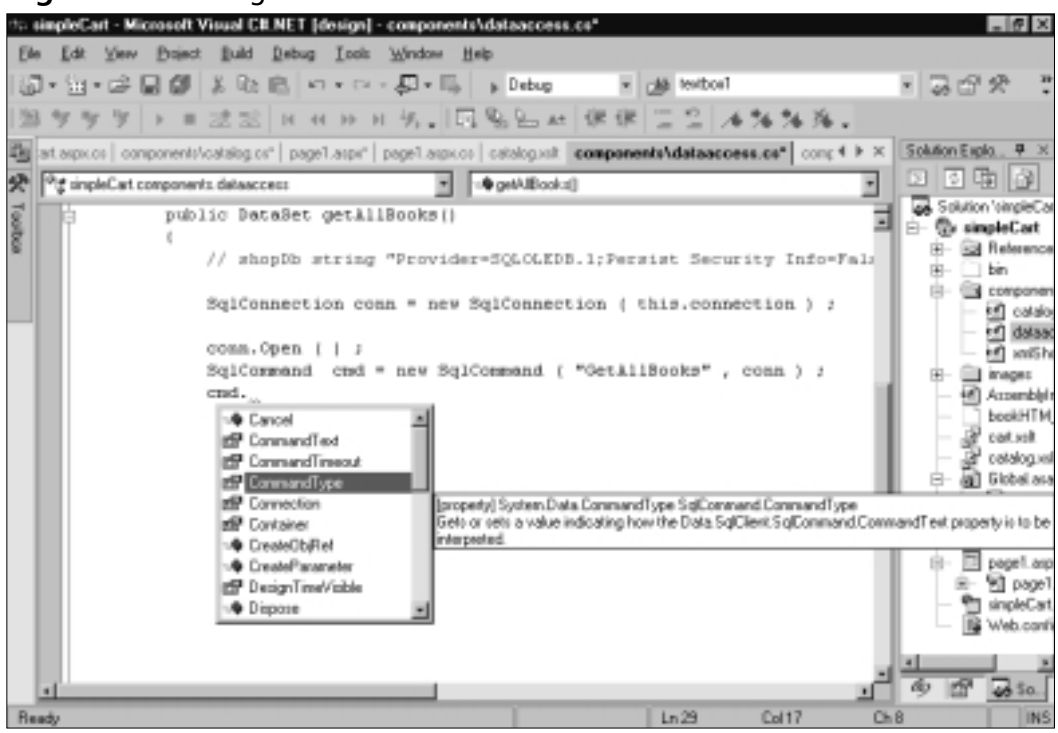
VS.NET has IntelliSense support for all the primary programming languages: VB.NET, C#, and C++. IntelliSense even exists for Cascading Style Sheets and HTML. Unfortunately, VS.NET doesn't include IntelliSense for XSLT in the Beta2 version—we might have to wait for the release version. Currently, ActiveState does make an XSLT plug-in for VS.NET that provides this functionality; you can obtain a free trial version at http://aspn.activestate.com/ASPN/Downloads/VisualXSLT.

While developing, you will notice that IntelliSense provides information about active classes only, meaning those that you have created in your project or those referenced in your page with the using Directive (for code-behind pages: pagename.aspx.cs). If you are trying to use an object or method, and no IntelliSense appears for it, you might have forgotten to include the reference.

For example, if you attempt to do data operations using the *SqlCommand* object, no IntelliSense will appear until you include the appropriate data class (Figure 2.15):

```
using System.Data.SqlClient;
```

Figure 2.15 Using IntelliSense



For C#, IntelliSense is available only in the code-behind page and not in the ASPX page itself. This might change in the release version. To disable IntelliSense, choose **Tools | Options | Text/Editor** and select the editor you are using, which should be C#. In the Statement Completion section, uncheck all the options, which will disable IntelliSense for the editor.

XML Editor

When working with XML, VS.NET has some interesting features. If you create a well-formed XML document of your own, you can easily generate a corresponding XSD schema that conforms to the 2001 W3C XML Schema. Once this is done, your XML document will have code completion based on this new schema. We'll take a detailed look at XML in general and schemas later, but for now, remember that XML is the "Extensible Markup Language" that can handle data, while schemas are used to validate their matching XML document.

To test creating a schema, let's open poll.xml and generate a schema for it:

1. Choose **File | Open**. Navigate to your CD-ROM drive and locate the file poll.xml.
2. Click **Open**. This should load the page into the IDE.
3. If the XML is one continuous line, simply click the **Format The Whole Document** icon (Figure 2.16).

Now, let's create a schema for this file. Right-click anywhere in the text editor and select **Create Schema**. You can see these resulting changes in Figure 2.17:

- A new file called poll.xsd was auto-generated by VS.NET.
- In the Properties window, the new schema is set as the file's target schema.
- An XML namespace attribute is added. This namespace behaves like the HTML namespace; it identifies the document as XML, lists its version, and its text encoding.
- IntelliSense based on the schema is now available for this document.

You can also select a different schema to base the XML file on by selecting a new schema from the targetSchema drop-down (Figure 2.18). This would then provide IntelliSense based on the schema selected.

Figure 2.16 Formatting an XML Document

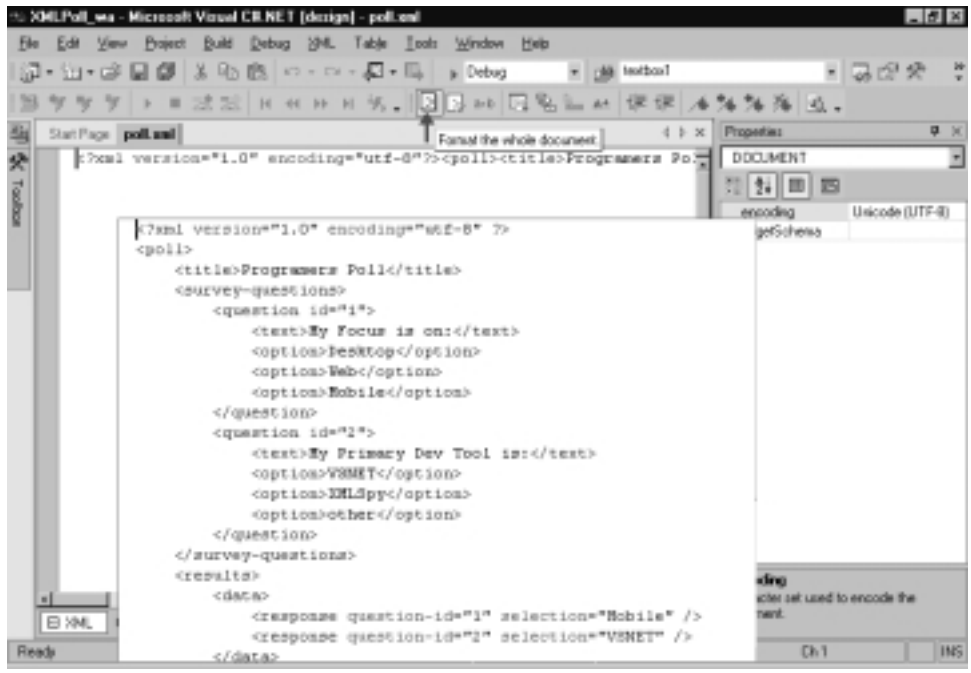


Figure 2.17 Generating a Schema for a Well-Formed XML Document

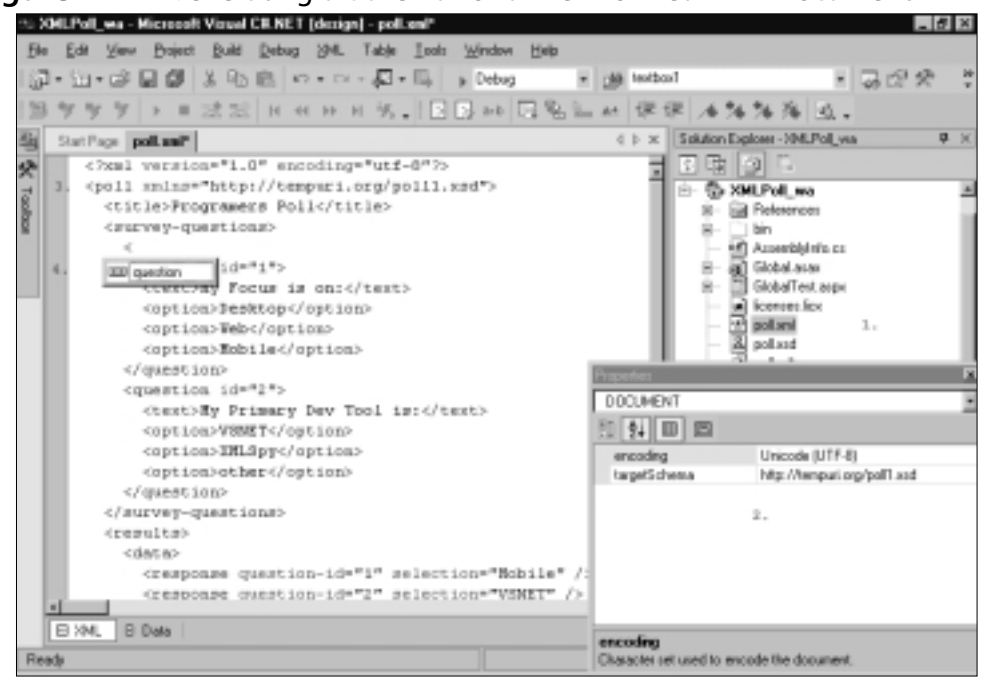
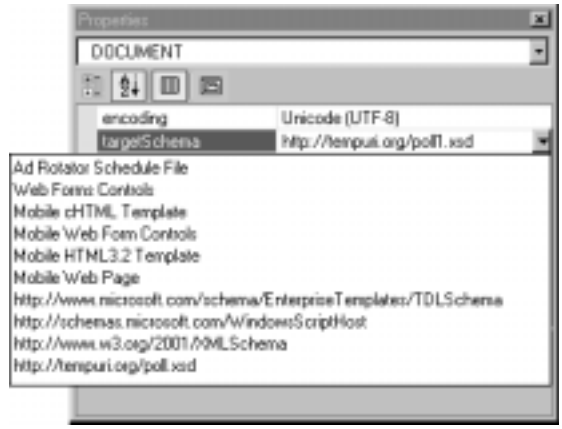
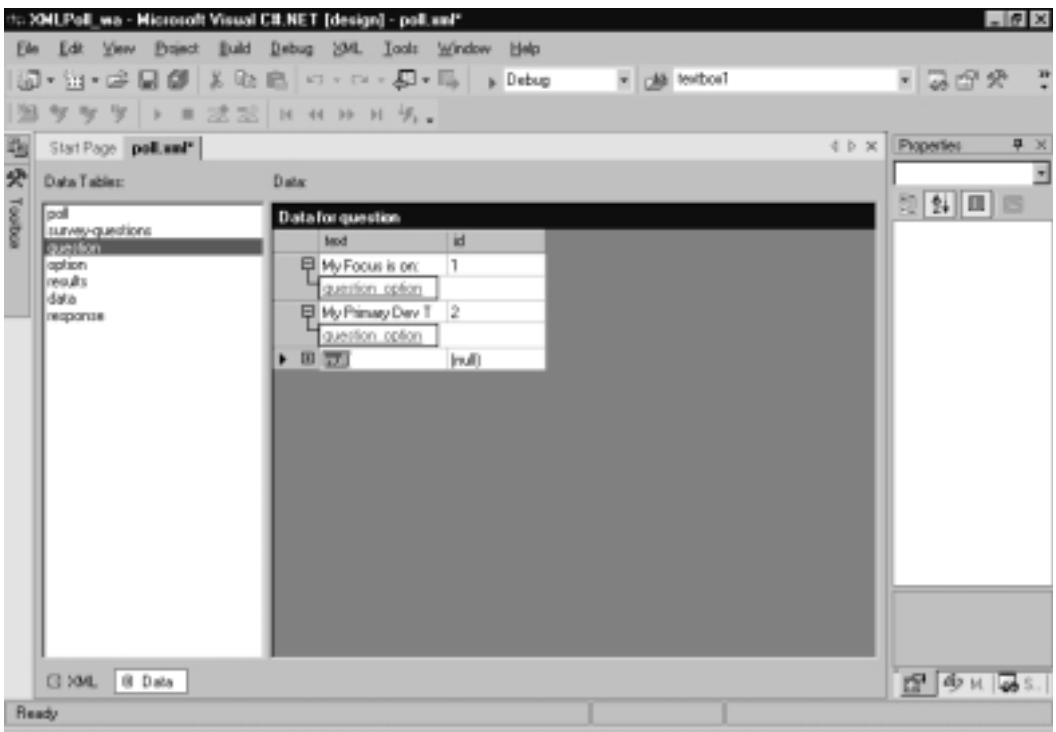


Figure 2.18 Selecting a Target Schema



You can also view XML documents in Data mode. This presents the document in a hierarchical structure. From this view, you can also add new nodes and data to the document (Figure 2.19).

Figure 2.19 Viewing an XML Document in Data Mode

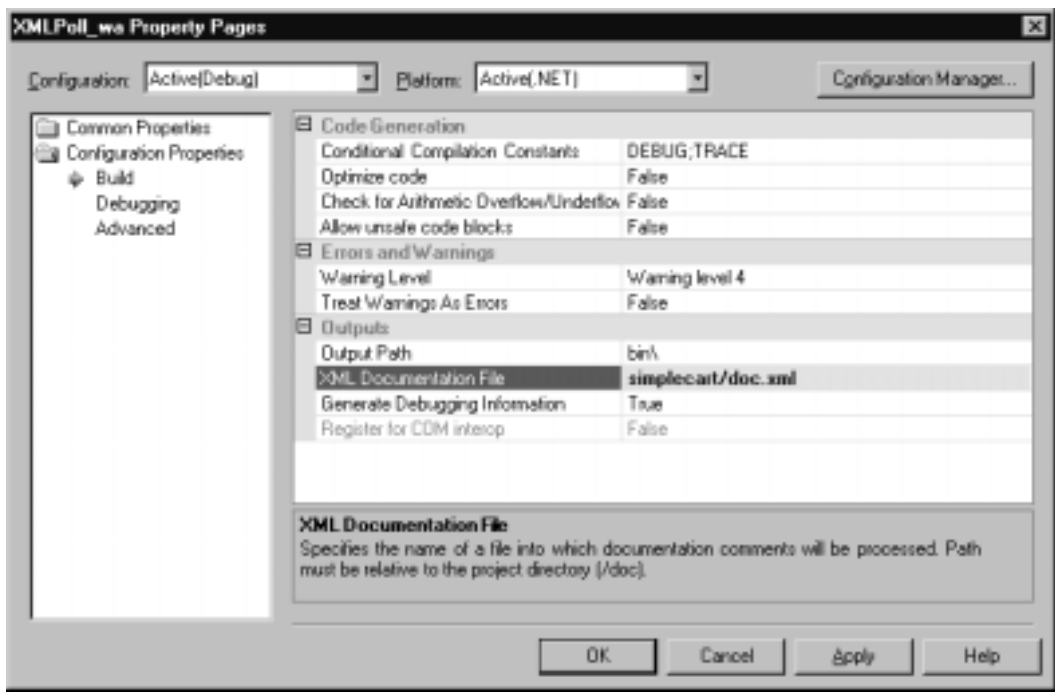


Documentation Generation (XML Embedded Commenting)

This feature enables you to comment your code with an embedded XML tagging structure. When XML documentation is enabled, an XML documentation file will be created during the build process. In the Solutions Explorer, right-click on the project name, and then select **Properties**. The Project Properties dialog appears. Click the **Configuration Properties** folder and select **Build**.

Find the item called **XML Documentation File** in the textbox next to this, provide a relative path to the file location you would like the Documentation written to, and click **Apply** (Figure 2.20).

Figure 2.20 Setting the XML Documentation File Source in the Project Properties Dialog



Now let's look at how to add XML comments to the code.

Adding XML Document Comments to C# Pages

To add XML documentation comments to your code, simply type three slashes above any class, method, or variable.

```
public DataSet catalogItemDetails( string book_isbn )
{
   return catalogRangeByCategory( -1, -1, book_isbn);
}
```

An XML representation of its inputs and outputs will be generated:

```
/// <summary>
///
/// </summary>
/// <param name="book_isbn"></param>
/// <returns></returns>
public DataSet catalogItemDetails( string book_isbn )
{
     return catalogRangeByCategory( -1, -1, book_isbn);
}
```

Simply add appropriate notes and build the project:

```
/// <summary>
/// Specialized interface to catalogRangeByCategory.
/// This Method returns all the data for only the given book
/// </summary>
/// <param name="book_isbn">string</param>
/// <returns>DataSet</returns>
public DataSet catalogItemDetails( string book_isbn )
{
return catalogRangeByCategory( -1, -1, book_isbn);
}
```

When you build the project, you will receive a list of warnings corresponding to every *Public* variable, property, method, and class that is not commented. Figure 2.21 shows what happens when you tell it to create comments; this is how it tells you what variable isn't commented. This will not prevent program execution, or the writing of the documentation file. Figure 2.22 contains the XML generated on build.

Figure 2.21 Warning for Uncommented Public Variables, Properties, Methods, and Classes

```
Output
Build
t\page1.aspx.cs(35,27): warning CS1591: Missing XML comment for publicly visible type or
t\page1.aspx.cs(36,23): warning CS1591: Missing XML comment for publicly visible type or
t\page1.aspx.cs(37,18): warning CS1591: Missing XML comment for publicly visible type or
t\page1.aspx.cs(38,20): warning CS1591: Missing XML comment for publicly visible type or
t\page1.aspx.cs(40,47): warning CS1591: Missing XML comment for publicly visible type or
t\page1.aspx.cs(41,51): warning CS1591: Missing XML comment for publicly visible type or
t\page1.aspx.cs(112,15): warning CS1591: Missing XML comment for publicly visible type or
```

Figure 2.22 Generated XML Documentation

```
D:\doc.xml - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address D:\doc.xml
    books within the given range</summary>
    <param name="startPos" />
    <param name="range" />
    <returns />
  </member>
- <member name="T:simpleCart.components.dataaccess">
    <summary>This Class is used for all interactions with the data source</summary>
  </member>
- <member name="M:simpleCart.components.dataaccess.getAllBooks">
    <summary>getAllBooks connects to the SQL database and executes the stored procedure
      "GetAllBooks"</summary>
    <returns>DataSet containing the DataTable "Books"</returns>
  </member>
- <member name="T:simpleCart.components.xmlShoppingCart">
    <summary>This component is a wrapper class for XML functions it provides add, remove,
      clear, and view methods</summary>
  </member>
- <member name="M:simpleCart.components.xmlShoppingCart.initCart
    (System.String,System.String)">
    <summary>Initializes the Cart. onpage load the cart can be initailized with an existing
      xmlCart string, this enables client caching of the cart.</summary>
    <param name="dataSource" />
  </member>
- <member name="T:simpleCart.Global">
    <summary>Summary description for Global.</summary>
  </member>
- <member name="T:simpleCart.page1">
Done
My Computer
```

Customizing the IDE

The VS.NET IDE is fully customizable. All windows can be set to *dockable*, *hide*, *auto hide*, and *floating*. You can display different toolbars for each different type of file, and you can create customizable toolbars. You can set font, tab, and text layout properties for each type of file. You can set the default Start page to open the last project, or even set it to a user-created page. If you mess up the layout, you can easily set it back to several predefined layouts.

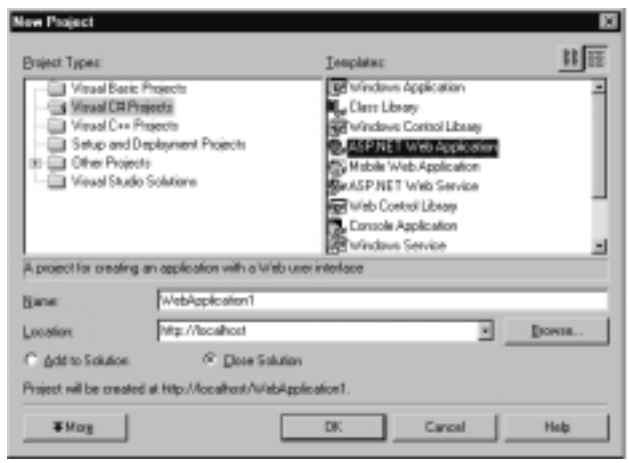
Creating a Project

Now that we have covered all the different aspects of the IDE, let's create a test project. We cover the different type of projects available, show how to add a Web reference to the project, and briefly go over some of the debugging tools available to the IDE. This should give a well-rounded tour of the complete IDE. Now let's go over the projects available.

Projects

Users new to .NET will see that three Web projects are added to the project listing for all languages: the ASP. NET, Application, Web Service, and Control Library. The other projects will be familiar to all VS 6 users (Figure 2.23).

Figure 2.23 Project Listing in the IDE



Creating a Project

For this example, we will build an ASP.NET Web application (Figure 2.23). You can keep the name as the default, or select a new name. The location should be **localhost** if you are developing on the same box as the IIS server; if not, you will have to place the location of the server in that text box, either through IP or the name of the server. The next option is to either close any open solutions and open this anew, or add it to the existing solution. We recommend that you choose to have it close all open solutions and open anew, so as not to task your machine with having multiple solutions in the same IDE. Click **OK**, and VS.NET will create the project for you.

Add Reference

One of the great benefits of working within the IDE of VS.NET is that you can add references to your project with ease. Try it out: In this project, select the project name in the Solutions Explorer. Right-click and select **Add Web Reference**. Now you will have to have a location to a WSDL file from which to locate and add in the Web Service to the project. This is covered later in the book.

You can also add a reference to a DLL to your project. This will be done in much the same way as the Web Reference. Instead of selecting Add Web Reference as we just did, select **Add Reference**, and then choose from all the available references on your machine.

Build the Project

To build a project, simply press **F5** or click the **Start** icon on the main window menu bar. The project will be compiled. You must also set a Start page before this takes place. To do that, right-click on the file you want as the Start page or window, and set it to **Start page**. This will launch this page first after the project has been compiled and run (Figure 2.24).

Figure 2.24 Compiling a Project

Debugging a Project

While building the project, any errors will bring up a dialog box, which will ask you to continue with the errors in place, or to stop debugging and correct any errors displayed. These errors will show in the Task window. You can double-click on any error in the Task window, and the IDE will take you to that location in the code. As you fix the bugs present in the task list, they will be removed. You can also set breakpoints and step over and step into options.

Summary

In this chapter, we've taken a tour of the VS.NET IDE. We've seen an overview of the interface, some of its component windows, and some of its built-in features. The design window and the code window are graphical tools used in creating an application. You can split the windows or have tab groups added to them; you can use the Toolbox (which includes Data, Components, Web Forms, and Window Forms) to drag and drop objects onto the design window. The Server Explorer window allows you to connect to a server on the network and have full access to that server, and to link to any database servers on the network.

One of the new features for VS.NET is that you can dock all the windows, or expand and collapse them within the view of the IDE. The Auto Hide feature of each window makes them slide off the screen and embed in the side when not needed; this enables you to have maximum code view but still have all windows present.

The Properties Explorer (similar to the one in VS 6 and the Visual Basic IDE and Visual Interdev IDE) allows you to select an object from the design window to see available attributes for that object listed. Any changes made in this window will be propagated to the design view and code view windows, respectively.

The Solution Explorer (the same as in VS 6) is a look at all the files in your solution via the four options: Refresh, Copy Web, Show All Files, and Properties. The .NET IDE has two different types of containers available for holding items: solutions and projects (you can have multiple projects within a solution, whereas the project container keeps only files and items within files). The Object Browser will give you a complete list of all classes' methods and properties in your solution.

Other windows include Dynamic Help and the Task List. Dynamic Help is a dockable window that you can fully customize to make it easy to tab to whatever information you are interested in. You can use the Task List for collaborative projects and in debugging; it lets you add and prioritize tasks.

IntelliSense, the code-completion technology Microsoft uses, is supported in VS.NET for VB.NET, C#, and C++, but not yet for XSLT. IntelliSense provides information about active classes. For C#, IntelliSense is available only in the code-behind page and not in the ASPX page itself.

Another important feature is XML Documentation. This feature enables you to comment your code with an embedded XML tagging structure. When XML documentation is enabled, an XML documentation file will be created during the build process.

We've looked at some issues such as the customizable, dockable, hide, auto hide, and float settings for many of the component windows, along with the profile setting on the Start page. VS.NET is a collection of integrated developer tools with which you should definitely be familiar.

Solutions Fast Track

Introducing Visual Studio.NET

- ☑ Visual Studio.NET (VS.NET) provides a consistent interface across the primary development languages.
- ☑ VS.NET provides easy-to-use tools for Windows and WebForms rapid prototyping across languages (including C# and Managed C++).

Components of VS.NET

- ☑ Enhanced window manipulation for user preferences within the Integrated Development Environment (IDE) gives the developer the ability to dock, auto hide, hide, or float all component windows.
- ☑ Task List has the ability to create custom tokens to map out and prioritize your code via the Task List.
- ☑ Server Explorer allows the developer to quickly connect and access any database server on the network, enabling direct access to all database objects, including stored procedures, functions, and user settings.

Features of VS.NET

- ☑ IntelliSense is one of the best tools at your disposal when learning a new language or technology. VS.NET has built IntelliSense into almost every aspect of the development process.

- ☑ Dynamically generated XML Documentation provides a fast and easy way to comment your code and generate a separate XML formatted documentation file. This tool makes code more self-documenting, and it should save developers time and ensure that some documentation is provided.
- ☑ Generating XML schemas from well-formed XML is now a breeze with .NET. You can also create new XML documents that conform to popular standards by selecting a *targetSchema* and using the IntelliSense feature to create valid XML documents.

Customizing the IDE

- ☑ The VS.NET IDE is fully customizable. All windows can be set to dockable, hide, auto hide, and floating. You can display different toolbars for each different type of file and create customizable toolbars. You can set font, tabbing, and text layout properties for each type of file.
- ☑ You can set the default Start page to open the last project, or even set it to a user-created page.
- ☑ The IDE also includes several common default settings in case you mess up while customizing your interface, settings such as the default VB 6 interface or Visual InterDev.

Creating a Project

- ☑ One of the great benefits of working within the IDE of VS.NET is that you can add references to your project with ease
- ☑ To build a project, simply press **F5** or click the **Start** icon on the main window menu bar.
- ☑ While building the project, any errors will bring up a dialog box, which will ask you to continue with the errors in place, or to stop debugging and correct any errors displayed.

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** and click on the **"Ask the Author"** form.

Q: How can I look up a parent class method or property of any system-level object?

A: Use the Class View window, accessed from the standard toolbar by clicking **View | Class View**.

Q: Does VS.NET support line numbering in its text editor?

A: Yes, from the standard toolbar, select **Tools | Options**. This will open the Options dialog; select the **Text Editor** folder, pick the language, and click on the check box for line numbering under the display section.

Q: Is there a way to set the tab size in the text editor?

A: Yes, from the standard tool bar, select **Tools | Options**. This will open the Options dialog; select the **Text Editor** folder, choose a language folder, select **Tabs**, and set them to your desired setting.

Chapter 3

Reviewing the Fundamentals of XML

Solutions in this chapter:

- An Overview of XML
- Well-Formed XML Documents
- Transforming XML through XSLT
- XPath

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

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, 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 the World Wide Web Consortium (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 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) use 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 one site to another 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 an XML document. The server can return the result via a subsequent XML document.

An Overview of XML

Extensible Markup Language (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 the conventional language of the Internet, the HyperText Markup Language (HTML). The following section touches on some of the basic concepts of XML.

The Goals of XML

Ten goals were defined by the creators of XML, which give definite direction as to how XML is to be used.

- XML shall be compatible with SGML.
- It shall be easy to write programs that process XML documents.
- The number of optional features in XML is to be kept to the absolute minimum; ideally, zero.
- XML documents should be human-legible and reasonably clear.
- The XML design should be prepared quickly.
- The design of XML shall be formal and concise.
- XML documents shall be easy to create.
- Terseness in XML markup is of minimal importance.
- XML shall be straightforwardly usable over the Internet.
- XML shall support a variety of applications.

In other words, XML is for sharing information easily via a nonproprietary format over the Internet. XML is made for everybody, to be used by everybody, for almost anything. In becoming the universal standard, XML has faced and met the challenge of convincing the development community that it is a good idea prior to another organization developing a different standard. The way in which XML achieved this was by being easy to understand, easy to use, and easy to implement.

What Does an XML Document Look Like?

The major objective is to organize information in such a way so that human beings can read and comprehend the data and its context; in addition, the document itself is technology and platform independent (nonproprietary, remember?). 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 preceding text file contains.

Now consider the following XML document (shown in Figure 3.1). The code is available in the *Catalog1.xml* file from www.syngress.com/solutions.

Figure 3.1 Catalog1.xml

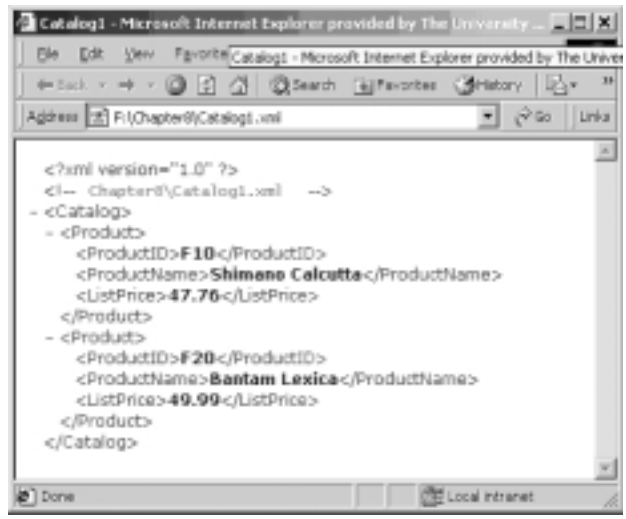
```
<?xml version="1.0"?>
<Catalog>
      <Product>
            <ProductID>F10</ProductID>
            <ProductName>Shimano Calcutta </ProductName>
            <ListPrice>47.76</ListPrice>
      </Product>
      <Product>
            <ProductID>F20</ProductID>
            <ProductName>Bantam Lexica</ProductName>
            <ListPrice>49.99</ListPrice>
      </Product>
</Catalog>
```

The document in Figure 3.1 is XML's way of representing data contained in a product catalog. It has many advantages: it is easily readable and comprehendible, self-documented, and 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!

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 from www.syngress.com/solutions in IE 5.0 or later. 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 by following these steps:

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, respectively. The XML Designer has many other tools to work with, which we will introduce later in this chapter.

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

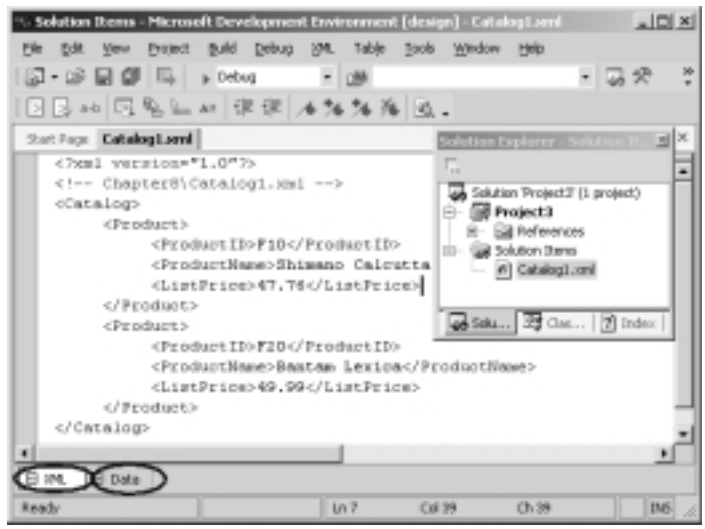
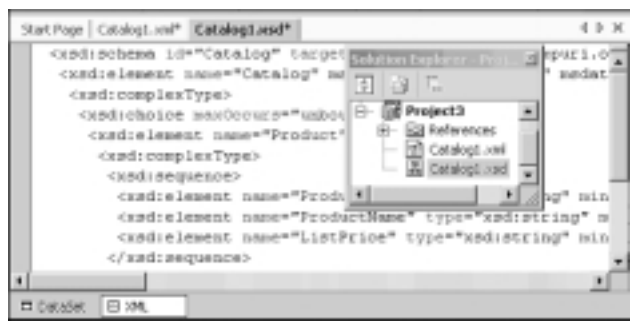


Figure 3.4 The Data View of an XML Document in VS.NET XML Designer



Components of an XML Document

An XML document contains a variety of constructs (also referred to as "elements"). Some of the frequently used ones include:

Declaration Each XML document can 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 recommendation for version 1.0.

Comment An XML document can contain HTML-style comments such as *<!--Catalog data -->*.

Schema or Document Type Definition (DTD) In certain situations, a schema or DTD might precede the XML document. A schema or DTD contains the rules about the elements of the document. For example, we can specify a rule like "A product element must have a *ProductName*, but a *ListPrice* element is optional." .NET uses schemas exclusively so we will not be discussing DTD in-depth.

Elements An XML document is mostly comprised of *elements*. An element has a start-tag and an end-tag. In between the start-tag and end-tag, we include the content of the element. An element might contain a piece of character data, or it might 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 such as *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 can 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 entire story in our first example. So far, we have said that an element can contain other elements, or data, or both. Besides these, an element can 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.

You can find a more complete list of XML's constructs at www.w3c.org/xml.

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, *Type* and *SupplierId*. As shown in Figure 3.4, 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 from www.syngress.com/solutions.

SYNGRESS
syngress.com

Figure 3.5 Catalog2.xml

```
<?xml version="1.0"?>

<Catalog>
   <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 comprised 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 might appear questionable. Nevertheless, they exist in the W3C recommendation, and in most situations become handy in designing otherwise complex XML-based systems.

Empty element

We have already mentioned a couple of times that an element can contain other elements, or data, or both. However, an element does not necessarily have to have any of these; if needed, it can be kept totally empty. For example, observe the following element:

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

The preceding element is a correct XML element. The name of the element is *Input*. It has three attributes: *type, id*, and *runat*. However, it does not contain any subelements, nor does it contain any explicit data. Hence, it is an *empty* element. We can specify an empty element in one of two ways:

- Just before the ">" symbol of the start-tag, add a slash (/), as shown in the preceding code.
- Terminate the element using a standard end-tag as follows:

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

Examples of empty elements include <br/>, <Pup Age=1 />, <Story> </Story>, and <Mail/>.

Structure of an XML Document

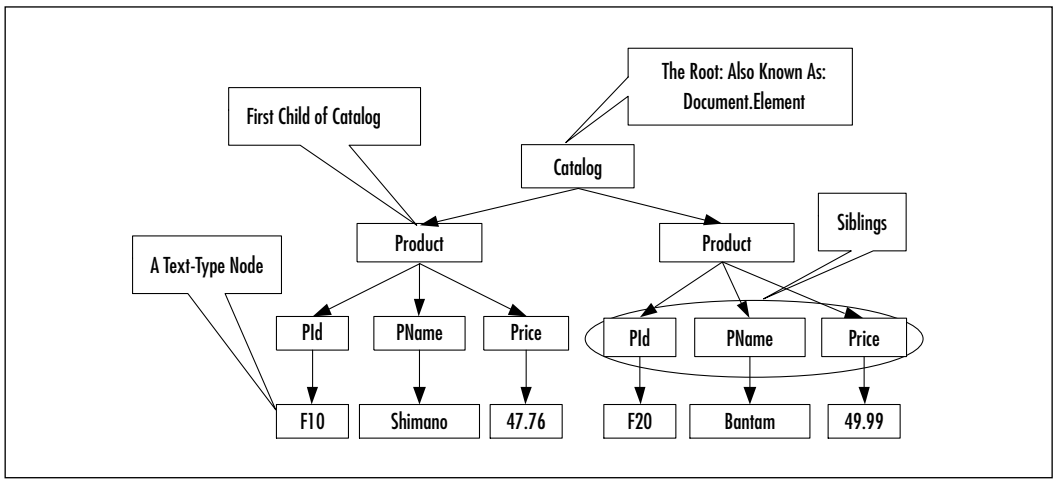
In an XML document, the data are stored in a hierarchical fashion. A hierarchy is also referred to as a tree in data structures. Conceptually, the data stored in the Catalog1.xml can be represented as a tree diagram as shown in Figure 3.6. Please note that certain element names and values have been abbreviated in the tree diagram, mostly to conserve real estate on the page.

In Figure 3.6, 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 *PId, PName*, and *Price* nodes.

In this particular tree diagram, the bottom-most nodes are *not* of *element-type*, but rather 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 *Pid, 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. Figure 3.6 illustrates these terms.

Figure 3.6 The Tree Diagram for *Catalog1.xml*



Well-Formed XML Documents

At first sight, an XML document might 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 allows us to spell many tags incorrectly (the browser will just ignore it), and it is a free world out there for people who are not case-sensitive. For example, we can use *<BODY>* and *</Body>* in the same HTML document without getting into trouble. When developing an XML document, however, certain rules must be followed. Some basic rules, among many others, include:

- 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 with an underscore.
- A particular attribute name can 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

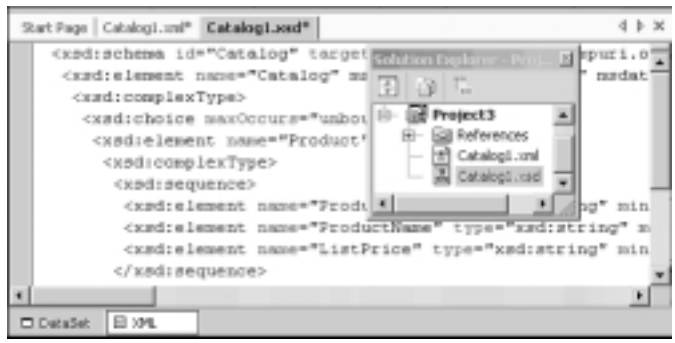
An XML document might be well-formed, but it might not necessarily be a valid XML document. A valid XML document conforms to the rules specified in its Document Type Definition (DTD) or schema. DTD and schema are actually two different ways to specify the rules about the contents of an XML document. The DTD has several shortcomings. First, a DTD document does not have to be coded in XML. That means that 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). 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 can develop a schema in many different ways. One way is to enter the definition manually using Notepad. We can also develop schema using visual tools, such as VS.NET or XML Authority. Many automated tools can 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 can generate a rough-cut schema of a sample XML document using VS.NET XML Designer. We can 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. To generate a rough-cut schema for our Catalog1.xml document (shown in Figure 3.1), follow these steps:

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.
2. 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 3.7. We will see the DataSet view tab and the XML view tab at the bottom of the screen. We will elaborate on the DataSet view later in the chapter.

Figure 3.7 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 might appear complex, there is no need to be 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 can 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 can 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 3.8. The code is also available from www.syngress.com/solutions.

NOTE

An XSD is itself a well-formed XML document.

Figure 3.8 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>
```

Continued

Figure 3.8 Continued

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

Developing & Deploying…

XML Validation in VS.NET

VS.NET provides a number of tools to work on XML documents. One allows us to check if a given XML document is *well-formed*. While on the XML view of an XML document, you can use **XML | Validate XML Data** in 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 also use the Schema Validation tool to check if your schema is well-formed. 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 preceding 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. Follow these steps to assign a schema to an XML document:

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** in the main menu.

By the way, many third-party software 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.

XML Schema Data Types

When an XML file acts as a database, and XSL and XPath act as SQL queries to render the XML file, we need a place where the contents in the XML file are declared somewhere with their data types. As in any database, whether SQL Server or Oracle, all columns are defined with data types, which is the relational-oriented concept. This led to the requirement of having data types in XML schema.

There are two types of data types, *primitive* and *derived*. Primitive data types are as is, and are not derived from any other data types (e.g., float). Derived data types are based on other data types. The integer data type is derived from the decimal data type.

The primitive data type defined for the purpose of XML schema need not be the same for other specifications or other databases, the same way in which the user-defined data types meant for XML schema are not meant for any other resources. Table 3.1 lists the various data types that XML schemas can take advantage of.

Table 3.1 XML Schema Data Types

Primitive Data Type	Derived Data Type	Fundamental Facets	Constraining Facets
String	normalizedString	equal	length
Boolean	Token	ordered	minLength
Decimal	Language	bounded	maxLength
Float	NMTOKEN	cardinality	pattern
Double	NMTOKENS	numeric	enumeration
Duration	Name		whiteSpace
dateTime	NCName		maxInclusive
Time	ID		maxExclusive
Date	IDREF		minExclusive
gYearMonth	IDREFS		minInclusive
gMonthDay	ENTITY		totalDigits
GDay	ENTITIES		fractionDigits
GMonth	Integer		
hexBinary	nonPositiveInteger		

Continued

Table 3.1 Continued

Primitive Data Type	Derived Data Type	Fundamental Facets	Constraining Facets
base64Binary	negativeInteger		
AnyURI	Long		
Qname	Int		
NOTATION	short		
GYear	Byte		
	nonNegativeInteger		
	unsignedLong		
	unsignedInt		
	unsignedShort		
	unsignedByte		
	positiveInteger		

Transforming XML through XSLT

Extensible Stylesheet Language Transformation (XSLT) is the transformation component of the XSL specification by the W3C (www.w3.org/Style/XSL). It is essentially a template-based declarative language that 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.

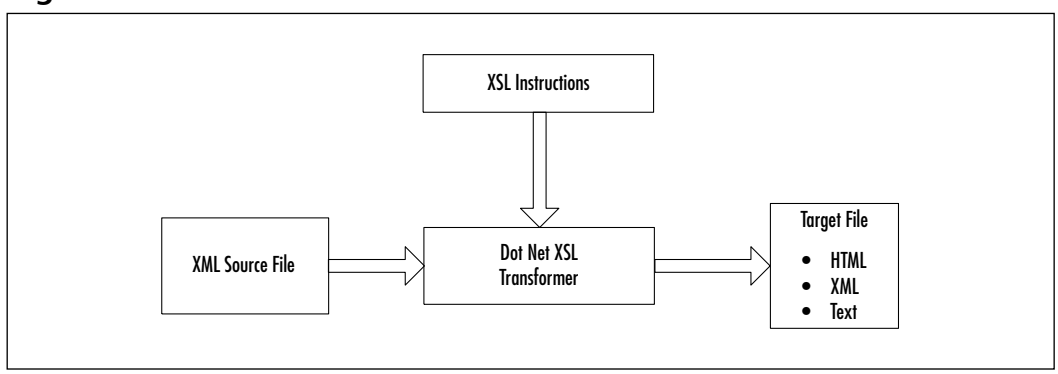
Internet Explorer (5.5 and later) has a built-in XSL transformer that automatically transforms an XML document to an HTML document. That is how, when we open an XML document in IE, it displays the data using a collapsible list view. However, Internet Explorer cannot be used to transform an XML document 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 ascending order of the unit price. Further, we might 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 can either develop relevant codes in a programming language such as C#, or we can 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 types of scenarios.

Since the majority of XML/XSLT transformations take place online, we will be using ASP.NET with VB.NET as our programming language to provide the following example. 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 3.9.

The following example will apply XSLT to transform an XML document to

Figure 3.9 XSL Transformation Process



an HTML document. We know that IE can automatically transform an XML document to a HTML document and display it on the screen in collapsible list view. However, in this particular example, we do not want to display all of our data that way; we want to display the filtered data in tabular form. Thus, we will transform the XML document to an HTML document of our choice (and not to IE's choice). The transformation process will select and filter some XML data to form an HTML table.

We will apply XSLT to extract the account information for Ohio customers from the Bank3.xml file shown in Figure 3.10, which is also available from www.syngress.com/solutions.

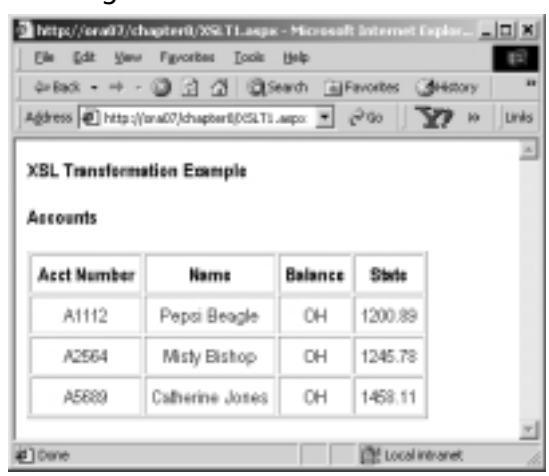
SYNGRESS
syngress.com

Figure 3.10 Bank3.xml file

```
<Bank>
   <Account AccountNo="A1112">
      <Name>Pepsi Beagle</Name>
      <Balance>1200.89</Balance>
      <State>OH</State>
   </Account>
   <Account AccountNo="A2564">
      <Name>Misty Bishop</Name>
      <Balance>1245.78</Balance>
      <State>OH</State>
   </Account>
   <Account AccountNo="A5689">
      <Name>Catherine Jones</Name>
      <Balance>1458.11</Balance>
      <State>OH</State>
   </Account>
</Bank>
```

The extracted data will be finally displayed in an HTML table. The output of the application is shown in Figure 3.11.

Figure 3.11 Transforming an XML Document into an HTML Document



If we need to use XSLT, we must first develop the XSLT style sheet (i.e., 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 3.12, and is available from 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 3.13, and is available from www.syngress.com/solutions.

SYNGRESS
syngress.com

Figure 3.12 XSLT1.xsl

```
<?xml version="1.0" ?>
<!-- Chapter 4\XSLT1.xsl -->
<xsl:stylesheet version="1.0"
       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">
```

Continued

Figure 3.12 Continued

```
      <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>
```

SYNGRESS
syngress.com

Figure 3.13 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 </b><br/>
<asp:Xml id="ourXSLTransform" runat="server"
      DocumentSource="Bank3.xml" TransformSource="XSLT1.xsl"/>
</form></body></html>
```

XSL Use of Patterns

Pattern matching occurs to define which XML elements belong to which XSL templates. To see an illustration of this function, look at the following examples of an XML document and an XSL style sheet. We used patterns in XSLT1.xsl to determine the location of the XML elements within Bank3.xml. Let's look at another, simpler example of patterns to better understand what they are. Figure 3.14 is an XML document containing some product information.

Figure 3.14 XML Product Information

```
<?xml version="1.0">
<Products>
      <Product>
            <ProductID>1001</ProductID>
            <ProductName>Baseball Cap</ProductName>
```

Continued

Figure 3.14 Continued

```
            <ProductPrice>$12.00</ProductPrice>
      </Product>
      <Product>
            <ProductID>1002</ProductID>
            <ProductName>Tennis Visor</ProductName>
            <ProductPrice>$10.00</ProductPrice>
      </Product>
</Products>
```

Now let's look at Figure 3.15 to see the XSL patterns used to transform our XML to Figure 3.16.

SYNGRESS
syngress.com

Figure 3.15 XSL Style Sheet for Product Information (products.xsl)

```
<?xml version="1.0">
<xsl:template xmlns:xsl="uri.xsl">
      <HTML>
            <HEAD>
                  <TITLE>Product list</TITLE>
            </HEAD>
            <BODY>
                  <TABLE cellpadding="3" cellspacing="0" border="1">
                  <xsl:repeat for="Products/Product>
                        <TR>
                              <TD>
                                    <xsl:get-value for="ProductName"/>
                              </TD>
                              <TD>
                                    <xsl:get-value for="ProductPrice">
                              </TD></TR>
                  </xsl:repeat>
                  </TABLE>
            </BODY>
      </HTML>
</xsl:template>
```

Figure 3.16 XML Product Info HTML Source Output

```
<HTML>
      <HEAD>
            <TITLE>Product list</TITLE>
      </HEAD>
      <BODY>
            <TABLE cellpadding="3" cellspacing="0" border="1">
                  <TR>
                        <TD>
                              Baseball Cap
                        </TD>
                        <TD>
                              $12.00
                        </TD></TR>
                  <TR>
                        <TD>
                              Tennis Visor
                        </TD>
                        <TD>
                               $10.00
                        </TD></TR>
            </TABLE>
      </BODY>
</HTML>
```

As you can see, you can use a combination of XML documents and XSL style sheets to transform your data into HTML. Why, you might ask? It seems like a lot more work than just generating HTML at runtime on the server. Well, it is more work, but the added benefits are worth it. Typically, your Web application will generate XML documents at runtime instead of HTML documents. The separation of data from display allows for parallel development of the presentation and business services of a Web application. This also reduces the friction between your Web developers and your component developers, as they tend to step on each other's toes a bit less. Also, you can use different style sheets to transform different HTML documents for different browsers, in an effort to utilize the additional functionality provided by those browsers.

Debugging…

Debugging XSL

The interaction of a style sheet with an XML document can be a complicated process, and, unfortunately, style sheet errors can often be cryptic. Microsoft has an HTML-based XSL debugger you can use to walk through the execution of your XSL. You can also view the source code to make your own improvements. You can find the XSL debugger at http://msdn.microsoft.com/downloads/samples/internet/xml/sxl_debugger/default.asp.

The following list contains examples of style sheet error messages you might run into when using Microsoft's XML Parser 3.0:

Description: Named template '<*template-name*>' does not exist in the style sheet.

You are trying to call or apply a style sheet by name that does not exist. Remember that XML is case sensitive. Make sure that the style sheet you are attempting to reference exists and is the correct case.

Description: End-tag '<*tag-name*>' does not match the start-tag '<*different-tag-name*>'.

Your XSL style sheet is not well-formed. Check your HTML to ensure that it is well-formed and that all your elements either are closed or are specified as empty tags.

Description: The character '<' cannot be used in an attribute value.

Typically, this error results from a missing " within an attribute list of an element.

XPath

XPath is another XML-related technology that has been standardized by the W3C. XPath is a language used to query an XML document for a list of nodes matching a given criteria. An XPath expression can specify both location and a pattern to match. You can also apply Boolean operators, string functions, and

arithmetic operators to XPath expressions to build extremely complex queries against an XML document. XPath also provides functions to do numeric evaluations such as summations and rounding. The full W3C XPath specification can be found at www.w3.org/TR/xpath. The following are some of the capabilities of the XPath language:

- Find all children of the current node
- Find all ancestor elements of the current context node with a specific tag
- Find the last child element of the current node with a specific tag.
- Find the nth child element of the current context node with a given attribute.
- Find the first child element with a tag of *<tag1>* or *<tag2>*.
- Get all child nodes that do not have an element with a given attribute.
- Get the sum of all child nodes with a numeric element.
- Get the count of all child nodes.

The preceding list just scratches the surface of the capabilities available using XPath. Again, the .NET framework provides support for XPath queries against XML DOM documents and read-only XPath documents. We will be working with XPath throughout the book by using its respective *System.XML* classes.

Summary

XML has emerged as the Web standard for representing and transmitting data over the Internet. The W3C has worked to establish standards for XML and related technologies, including XML DOM, XPath, XSL, and XML schemas. XML DOM is an API that is used to create, modify, and traverse XML documents. XPath is a language that is used to query XML documents. XSL translates XML documents from one format to another. XML schemas define the structure and data types of the nodes in an XML document. All of these technologies are industry standards backed by the W3C. Microsoft has taken all of these standards and packaged them into their .NET architecture. This book focuses heavily on the System.XML class, where we will find all of the necessary support for creating, reading, editing, and working with XML, schema, XPath, and limited XSL. This chapter was meant to be just a review of XML so that, as we look through the rest of the chapters, you will have a fresh memory of XML against which to reference. Now, we will get ready to look at XML as it is used with .NET.

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 standalone 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.

Well-Formed XML

- ☑ While XML does not need to be well-formed, it is a good habit to get into.

- ☑ There are two ways to provide validation for XML: through schema and DTD.
- ☑ .NET uses schema exclusively to provide validation for XML in .NET.
- ☑ Schemas allow for greater flexibility and precision compared to DTD.
- ☑ You can use VS.NET to generate a schema for your XML file.

Transforming an XML Document Using XSLT

- ☑ You can use XSLT (XML Style Sheet Language Transformation) 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.

XPath

- ☑ XPath is another W3 recommendation that acts as a query language for XML.
- ☑ XPath uses pattern-matching with expressions, just like XSLT, but with more support and functionality.
- ☑ XPath is not used to transform XML, but rather to facilitate the searching and querying of data.

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** and click on the **"Ask the Author"** form.

Q: I am a newbie to .NET; what impact does XML have on it?

A: XML is integrated heavily through .NET. It is found in everything from System.Data to System.Xml namespace, it is the backbone of the configuration system, and it is used in SOAP messaging and much more.

Q: How do I know when to use an element versus an attribute when defining the structure of my XML?

A: It is very hard to define catchall rules to determine when to use an element versus an attribute. Remember, though, that you can do very little validation with attributes other than making sure that they exist. For the most part, if there is any doubt, use an element to describe your content.

Q: Are there any XML editors out there?

A: Yes, quite a few, one of which is XML Notepad by Microsoft, which is not very good. The one we personally prefer to use is XML Spy. You might have a little learning curve with the user interface, but it is by far the best XML editor available when considering price. Sometimes, though, nothing beats Notepad when you need something down and dirty.

Q: Do I always have to define a schema for my XML document?

A: No, you don't always need a schema. Schemas are great for when you have to do validation—typically when exchanging XML documents over the Internet. Performing validation all the time might seem like a great idea, but it is a very expensive operation that can bog down a Web server. When shooting out XML to the Web, you typically don't need a schema, although it is a great way to document your XML.

Q: How can I use XSL to make my applications completely browser independent?

A: XSL is a tool you can use to transform XML to HTML. You can create several style sheets. Each can be especially suited for a particular browser, and depending on the browser of the client, you can transform the XML using the respective style sheet. This not only allows you to support Netscape and Internet Explorer, but also allows you to support almost any Internet-enabled device, from handhelds to cell phones.

Q: What W3C level of support is provided in the XML classes supplied with the .NET Framework?

A: The *XmlDataDocument* class supports W3C DOM Core Level 1 and Core Level 2 specifications. The *XmlSchema* class supports W3C XML Schemas for Structures and the XML Schemas for Data Types specifications. The *XslTransform* class supports the XSLT 1.0 specification. See the W3C Web site for details on the specifications at www.w3c.org.

Chapter 4

Using XML in the .NET Framework

Solutions in this chapter:

- Explaining the XML Document Object Model
- Introduction to the System.Xml Namespace
- Using the System.Xml Namespace

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

As you turn to the .NET Framework to start working with XML, you find a number of namespaces that have "Xml" in their names. They deliver a broad set of classes, supporting you in the use of XML. We will discuss all XML-related namespaces in the coming chapters, which will make it clear that they support in most cases all of your XML needs. However, the .NET Framework does not support all standards available. Luckily, it will let you built your own XML namespaces, using the existing ones.

XML and its related standards will keep on evolving, making it not always easy for Microsoft to keep the namespaces up with these developments. Let's hope that Microsoft makes use of the features they built into the .NET Framework by updating and releasing individual assemblies; in our case, the namespaces with improved XML support.

This chapter discusses the *System.Xml* namespace that contains the classes that implement the XML Core. For this reason, we will also take a closer look at the XML Document Object Model (DOM) that is at the heart of the XML definition.

Explaining the XML Document Object Model

The XML Recommendations from the W3C were not completed in one day. They were written very slowly, one at a time, allowing a good deal of time between each before the next was drafted. This allowed the W3C to iteratively create recommendations and determine features of the DOM levels. In doing so, each DOM level built upon the previous level, and it is possible to create an XML parser that implements some, but not all, of the levels (or even some features of each level, but not all features).

The DOM is a recommendation made by the W3C to enable a standard way of providing access to the data contained in an XML document. It is essentially a list of interfaces and class recommendations to allow anyone to internally implement an XML parser and expose a common interface to enable a developer access to his or her XML data. This interface is based on the idea that the XML is in a hierarchical or tree-like structure, where certain nodes might contain child nodes, which might also contain child nodes, and so on. The DOM provides the capabilities to navigate this tree of nodes in a relatively simple programmatic manner. Some of the DOM API capabilities include:

- Find the root node in a XML document.
- Find a list of elements with a given tag name.
- Get a list of children of a given node.
- Get the parent of a given node.
- Get the tag name of an element.
- Get the data associated with an element.
- Get a list of attributes of an element.
- Get the tag name of an attribute.
- Get the value of an attribute.
- Add, modify, or delete an element in the document.
- Add, modify, or delete an attribute in the document.
- Copy a node in a document (including subnodes).

The DOM is not a programming tool or COM object or anything you can directly program against. The W3C does not have a team of developers creating applications and tools to provide developers with the ability to read and write XML documents. The W3C simply works on creating recommendations for standards, so when a Java developer accesses XML, then switches to using Microsoft's XML parser (MSXML), then to a parser written in C, access to the XML data is the same, and the developer does not need to learn new tools and methodologies.

The Different XML DOM Levels

DOM Level 1 is effectively the building block for all DOM-based XML parsers. This level provides definitions for every part of the most basic XML document. If you consider the XML in Figure 4.1, you will see a number of different types of *elements*. An element is defined by DOM Level 1 to be an element, text, comment, processing instruction, CDATA section, or an entity reference. In addition to the elements in the XML, there is the processing instruction (*<? ?> tag*), a document element (the *<addressBook>* element), and a number of attributes. For more information on DOM Level 1, go to www.w3.org/TR/REC-DOM-Level-1/level-one-core.html.

Figure 4.1 Sample XML

```
<?xml version="1.0"?>
<addressBook>
   <category name="Friends">
      <entry name="Bill Gates" phoneNumber="555-1212" />
      <entry name="Steve Jobs" phoneNumber="555-1213" />
   </category>
</addressBook>
```

DOM Level 2 provides a number of small changes to DOM Level 1, and provides a number of new interfaces and functionality to programmatically access XML data. Level 2 adds support for Cascading Style Sheets (CSS) and events (user interface and tree manipulation). Level 2 Core also adds the notion of namespaces attached to different nodes. A *namespace* is simply another level of naming to add to your XML elements and attributes by specifying a namespace in front of the name. In addition to namespaces, DOM Level 2 adds definitions of DOM Ranges and DOM Traversal. DOM Range is a way to select content in an XML document based on two boundary points. DOM Traversal defines interfaces to navigate the hierarchical structure of XML documents. In order for an XML parser to support DOM Range and DOM Traversal, it must also support DOM Level 2 Core. For more information on DOM Level 2, go to www.w3.org/TR/DOM-Level-2-Core/core.html.

DOM Level 3 is the newest level in the Document Object Model specifications. Currently, Level 3 is not finished, but will extend Level 2 by finishing support for XML 1.0 with namespaces and adding additional keyboard events to the DOM API. In addition, it will add support for abstract schemas—this could be a very interesting feature, especially for those who have a lot of object-oriented experience; the idea of an abstract schema for your data that you can extend and override and still use the base abstract definition.

XML DOM Core Interfaces

Microsoft .NET currently supports DOM Level 1 and a subset of DOM Level 2. Microsoft has also provided additions to the DOM in their XML classes, in a manner to ease the work necessary to manipulate XML documents. Level 3 support has not been established yet, as the specification has not been finalized by the W3C, although you could rightfully expect Microsoft to examine and likely implement DOM Level 3 in a future version of their .NET framework.

DOM Structure Model

The W3C has broken down the DOM into a tree structure of nodes of varying types. This model has a number of interfaces and all of its attributes and methods clearly defined.

The *Node* interface is the base interface for all elements. All objects implementing this interface expose methods for dealing with children, yet not all objects implementing this interface may have children. Tables 4.1 and 4.2 detail a number of attributes and methods of the *Node* interface.

Table 4.1 The Node Interface Attributes

Attribute	Description
nodeName	The name of this node, depending on the type of node in question.
nodeValue	The value of this node. Note, however, that only certain types of nodes (attributes, text) will return a valid result.
nodeType	The type of node in question.
childNodes	The child nodes of this node.
firstChild	The first child node of this node.
lastChild	The last child node of this node.
Attributes	The attributes on this node.
ownerDocument	The Document object associated with this node.

Table 4.2 The Node Interface Methods

Method	Description
insertBefore(newChild, refChild)	Insert a new node before the refNode.
replaceChild(newChild, oldChild)	Replace the old child node with a new node.
removeChild(oldChild)	Remove the old child from the context node's list of children.
appendChild(newChild)	Append a new node to the end of the list of child nodes in the context node.
hasChildNodes()	Returns true or false depending on whether the context node has child nodes.
cloneNode()	Return a duplicate of the context node.

The *Document* interface represents the top-level root in an XML document, and implements the *Node* interface. It is the root of the XML tree, and provides the primary access to the XML contained within the document. The attributes and methods of the *Document* interface are listed in Tables 4.3 and 4.4.

Table 4.3 The *Document* Interface Attributes

Attribute	Description
doctype	The Document Type Declaration (DTD) with this document. If DTD is not present, this returns null.
documentElement	The root element in the document.

Table 4.4 The *Document* Interface Methods

Method	Description
createElement(tagName)	Creates and returns an element with the specified tag name.
createTextNode(data)	Creates and returns a *Text* object containing the specified string data.
createComment(data)	Creates and returns a *Comment* object containing the specified string data.
createCDATASection(data)	Creates and returns a CDATA object containing the specified string data.
createAttribute(name)	Creates and returns an *Attr* object with the specified name.

The Element interface is the most common type of node a developer will encounter when working with a DOM XML parser. Considering the XML shown previously in Figure 4.1, every *addressBook*, *category*, and *entry* element is of type *Node*. The attributes and methods of the *Element* interface are listed in Tables 4.5 and 4.6.

Table 4.5 The *Element* Interface Attributes

Attribute	Description
tagName	The name of the element.

Table 4.6 The *Element* Interface Methods

Method	Description
getAttribute(name)	Returns an attribute value by name.
setAttribute(name, value)	Adds or sets the value of a named attribute.
removeAttribute(name)	Removes an attribute by name.
getAttributeNode(name)	Returns an *Attr* node by name.
setAttributeNode(newAttr)	Adds or replaces an attribute.
removeAttributeNode(oldAttr)	Removes a given *Attribute* node.
getElementsByTagName(name)	Returns a *NodeList* of all descendent elements with the given node name.
removeAttribute(name)	Removes a given named attribute.
removeAttributeNode(oldAttr)	Removes a given *Attr* from the context node.
setAttribute(name, value)	Sets a named attribute to have a certain value.
setAttributeNode(newAttr)	Adds or updates the *newAttr* in the context node.

The *Attr* (*Attribute*) element is the second most common type of node a developer will encounter. It is incredibly simple and has no methods, only properties. See Table 4.7 for the attributes of the *Attribute* interface.

Table 4.7 The *Attribute* Interface Attributes

Attribute	Description
name	The name of this attribute.
ownerElement	The element that contains this attribute.
specified	This returns true if the attribute was defined and given a value in the original XML document, and false otherwise.
value	This returns the value of the attribute.

The last major interface in the DOM structure worth mentioning is the *NodeList*. The *NodeList* is a live list of nodes in a given XML document. A *NodeList* can be retrieved by the *Node.childNodes* property and the *Document.getElementsByTagName* method. Being a live list of nodes, any changes to the

nodes in a *NodeList* affects the original nodes in the containing XML document. This makes the *NodeList* a very powerful interface, allowing a developer to get a subset of nodes in his or her document, make changes to them, and not have to manually replace the nodes in the XML document.

There are a number of other different node types in the DOM Structure Model, including the *Text* node, *Comment* node, *ProcessingInstruction* node, and CDATASection node. Each of these additional node types represents other various types of nodes found in XML documents. They each have a few attributes and methods, and all inherit the *Node* interface. For a complete list of all the node types and their attributes and methods in the DOM Core Level 2, go to www.w3.org/TR/DOM-Level-2-Core/core.html.

DOM Traversal

DOM Level 2 includes a number of optional traversal interfaces, as well as defining the core DOM model. None of the interfaces in the DOM Traversal specification are required if an XML parser implements DOM Level 2. The three interfaces in the DOM Traversal specification are the *TreeWalker*, *NodeIterator*, and *NodeFilter*.

The *NodeIterator* and *TreeWalker* interfaces are defined to provide easy-to-use and robust traversal of a document's contents. The *NodeIterator* allows a developer to navigate forward and backward through a list of nodes, without the ability to move up and down the hierarchical relationships of the tree. The *TreeWalker* maintains the XML document's hierarchical structure and allows a developer to navigate the entire hierarchy without losing the parent-child relationship like the *NodeIterator*. In general, the *NodeIterator* is used to provide access to the data that specific nodes contain, whereas the *TreeWalker* is used when the structure of the document around the selected nodes is as important as the content of each node. Both the *TreeWalker* and *NodeIterator* are dynamic, in that changes to the nodes returned from either will affect the underlying XML document.

A *NodeIterator* or a *TreeWalker* can be associated with a *NodeFilter*, which examines each node and determines if the node should appear in the *logical view* of *Nodes* represented by each. By *logical view*, we mean that the physical view of the data can be different from the view a *NodeIterator* or *TreeWalker* shows to a developer. In either case, the physical view is maintained as well as the logical view, so additions, deletions, and updates to the logical view directly affect the real, physical view of the data.

NodeIterator

The *NodeIterator* view of its nodes is an ordered list, with the nodes appearing in document order. That is, the nodes in the list appear in the same order they appear in the underlying XML data. A *NodeIterator*'s position is always before the first node, between two nodes, or after the last node. When a *NodeIterator* is created, the position is set to before the first node, as shown in Figure 4.2. The asterisk (*) shows the position of the *NodeIterator*.

Figure 4.2 *NodeIterator* Position after Creation

```
* Node1 Node2 Node3 Node4 Node5
```

Each call to *nextNode()* returns the next node in the list, advances the position by one, and returns the node it just passed. For example, after two calls to *nextNode()*, the position is placed between *Node2* and *Node3*, with *Node2* being returned, as shown in Figure 4.3.

Figure 4.3 *NodeIterator* after Two Calls to *nextNode()*

```
Node1 [Node2] * Node3 Node4 Node5
```

As you can see, *Node2* is represented as being selected by being bracketed. If a call to *previousNode()* were to occur, the position would be placed between *Node1* and *Node2*, and *Node1* would be returned, as shown in Figure 4.4.

Figure 4.4 *NodeIterator* after Call to *previousNode()*

```
[Node1] * Node2 Node3 Node4 Node5
```

If a call to *previousNode()* were to occur again, the position would be placed before *Node1*, and since the *NodeIterator* is now at the beginning of the list, *null* would be returned. The same is true if *nextNode()* is called at the end of the list. The position would be placed at the end of the list, and *null* would be returned from the method call. Any subsequent calls to *nextNode()* or *previousNode()* when the position is at the end or beginning of the list, respectively, a *null* value would be returned and the position would not change.

The *NodeList* interface demands that the implementation be robust in that it must be able to gracefully handle changes such as additions or deletions to the underlying data and not fail. In the preceding example, if *Node3* were removed by

another method call and the call to *nextNode()* attempted to return *Node3*, it would instead gracefully handle this by updating its list of nodes (while still maintaining document order) and return the next available node in the list. For example, in Figure 4.3, *Node2* is the currently active node. If that node were to be removed, the position would stay the same, but that node would be removed from the logical view. Any subsequent calls to retrieve the current node would return the node directly before the removed node, or *Node1* (Figure 4.5).

Figure 4.5 After Removing the Context Node

```
[Node1] * Node3 Node4 Node5
```

TreeWalker

The *TreeWalker* provides much of the same functionality as the *NodeIterator*, but in a tree-oriented view of the nodes in a subtree, rather than a list-oriented view. The *TreeWalker* allows you to move forward and back, or to the parent node, one of its children, or to a sibling node. The *TreeWalker* navigation is very similar to the navigation used directly on *Node* objects. The major difference is that the *TreeWalker* can represent a logical view of the data, whereas direct *Node* navigation forces the developer to handle every node.

Like the *NodeIterator*, the *TreeWalker* interface demands that the implementation be able to gracefully handle changes in structure to the underlying XML document. Where the *NodeIterator* maintains its position in the list by staying close to the context node, the *TreeWalker*'s position is set based on the *currentNode* property of the *TreeWalker*. For example, if the context node of a *TreeWalker* in the underlying XML document is removed, the position of the *TreeWalker* would remain the same. Any calls to retrieve the current node would return the newly removed node, and would allow the developer to navigate through the deleted node and its children. Taking the example in Figure 4.5 and applying the *TreeWalker* to it, the context node would still be *Node2* as in Figure 4.6.

Figure 4.6 Removing a *Node* from a *TreeWalker*

```
Node1 Node3 Node4 Node5
[Node2]
```

Consider the XML document fragment in Figure 4.7. If the current context node is the *<category />* element and it is removed, the *TreeWalker* will still contain that node as the current node. The *TreeWalker* can navigate the children of

the context node freely without invalidating it. However, if the developer attempts to move out of this deleted subtree, the node that originally contained it in the XML document will effectively "recapture" it, maintaining the position in the document tree as if the *<category />* node had never been deleted and a call to move to the next sibling of that node was received.

Figure 4.7 XML Document Fragment

```
…
<category name="Friends">
   <entry name="Dave Thomas" phoneNumber="555-1212" />
   <entry name="Ronald McDonald" phoneNumber="555-1213" />
</category>
…
```

NodeFilter

The *NodeFilter* interface is used to further define a filter on the nodes returned by the logical view of either the *NodeIterator* or *TreeWalker*. It does this by its single method, *acceptNode()*. This method allows a *NodeIterator* or *TreeWalker* to ask the filter whether the current node should be present in the logical view. If a *NodeFilter* rejects the node in question, the process of selecting a node is repeated until a node is accepted by the filter. If no node is found, a *null* value is returned. Examine the mock C# code in Figure 4.8.

Figure 4.8 *NodeFilter* Mock C# Code

```
public Node nextNode()
{
   while(NodeFilter.acceptNode(this.nodes[this.position]) ==
      FILTER_SKIP && this.position < this.length)
   {
      this.position++;
   }
   if(this.position >= this.length)
      return null;
   else
      return this.nodes[this.position];
}
```

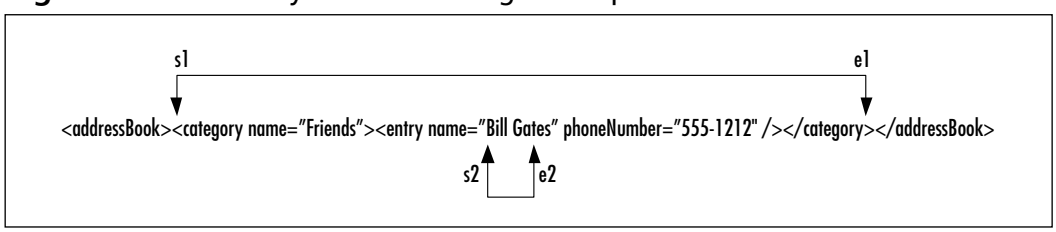
Obviously this is not real C# code using the *System.Xml* classes, but it does demonstrate the idea of the *NodeFilter*. The *nextNode()* method is requesting a *NodeFilter* to accept or not accept the node at the current position. It loops through all the nodes until it can find a node that the filter does not want to skip. At that point, it exits and returns the node at the current position. However, if the position goes to the end of the list, it returns *null* and exits the method call.

DOM Range

Like DOM Traversal, DOM Range is not mandatory for Level 2 XML parsers. A Level 2 parser can implement some or all of the DOM Range specification if the developers of the parser wish to do so. According to the W3C DOM Range specification, the *Range* interface can be seen as a convenience interface where all of the operations one can perform on the *Range* can be mapped to a series of node editing operations.

A range in an XML document can be thought of as two boundary points. A boundary point is essentially the node that contains it, and an offset. The node is called the *container* of the boundary point and its position. Every container and their ancestors are ancestor containers of the boundary point and its position. If the container is an *Attribute*, *Document*, *DocumentFragment*, *Element*, or *EntityReference*, the offset is between its child nodes, whereas if the container is a *CharacterData*, *Comment*, or *ProcessingInstructions*, the offset is between characters of the string contained by it. See Figure 4.9 for an example of boundary points and ranges.

Figure 4.9 Boundary Point and Range Example



The range denoted by the boundary points s1 and e1 both have a container of the *addressBook* element. The boundary point s1 has an offset of zero (0), because it is at the beginning of the container node. The corresponding end point e1 has an offset of one (1), since it points to the end of the first container in the *addressBook* element. Every node inside this element is considered an ancestor container of this range. The boundary point denoted by s2 has an offset of 1, since it starts at the "i" character, and the corresponding end boundary point

has an offset of 8. Both boundary points are located in the *name* attribute; therefore, these two boundary points both have the *name* attribute as their node.

A node is considered to be selected by a *Range* if it is between the two boundary points of the *Range*. That is, if a node's start position and end position fall between a range, then it has been selected by the *Range*. For example, range 1 in Figure 4.9 selects the *category* and *entry* nodes that are contained within it. A node is partially selected by a *Range* if a portion of it is contained within the range but the whole node is not contained within the range. For example, range 2 in Figure 4.9 selects "Bill Gat", but not the entire string "Bill Gates" .Hence, this node is only partially selected by the range.

The DOM Range specification defines a couple of interfaces to work with ranges in XML documents. One of the interfaces is aptly named "Range." The Range interface is the main interface provided to work with ranges. The methods provided on this interface allow a developer to move a range, select its contents, and replace the contents with something new. The other interface provided by the DOM Range specification is the DocumentRange interface. This interface contains a single method, *CreateRange*, which creates a range for use in a given XML document.

DOM XPath

XPath was not introduced to the Document Object Model until Level 3, which at the time of this writing is still under development by the W3C. Since the recommendation is not finished, we won't go into very much detail about XPath in the DOM, but we will go over some of the major items presented in the recommendation.

The main interface provided in this draft is the *XPathEvaluator*, which provides evaluation of XPath 1.0 expressions without support for extension functions or variables. As of this writing, the draft states that it is expected that the *XPathEvaluator* interface will be implemented on the same objects that implement the *Document* interface.

Another interface in the draft is *XPathExpression*, which represents a parsed XPath 1.0 expression. The *XPathExpression* interface provides a method to evaluate the expression based on a given context node. Tightly coupled with *XPathExpression* is the *XPathResult* interface. *XPathResult* represents the result from an evaluated *XPathExpression*, with methods to retrieve iterators on the result set of nodes. The iterator returned implements the *XPathSetIterator*, which only provides methods for iteration of a result set of nodes.

Introduction to the System.Xml Namespace

The *System.Xml* namespace is Microsoft's newest implementation of a DOM XML parser. The various classes that live in the *System.Xml* namespace represent their implementation of the various DOM interfaces. According to Microsoft, their *System.Xml* namespace classes implement DOM Level 1 Core and DOM Level 2 Core. In addition, the *System.Xml.Schema* namespace implements the Schema recommendation, and the *System.Xml.XPath* namespace implements DOM XPath. For the most part, all of the classes in the *System.Xml* namespace are named similarly to the DOM interfaces you read about earlier in this chapter, except they are prefixed with "Xml".

Overview of *System.Xml.Schema* Classes

The *System.Xml.Schema* classes are used to represent an XML schema. These classes support the XML Schema 1 (XML Schema for Structures) recommendation for schema mapping and validation. These classes also support the XML Schema 2 (XML Schema for Data Types) for data types in XML schemas. For more information on XML schemas, go to www.w3.org/TR/xmlschema-1/ or www.w3.org/TR/xmlschema-2/.

In short, XML schemas are used to define the data types and structures present in an XML document. XML schemas are used to define what each data type an XML element represents. DTDs are useful in that they provide information about what nodes and attributes can appear in an XML document and where they can appear, whereas an XML Schema Document (XSD) is used to provide an even further level of definition as to what the elements and attributes represent. Consider the XSD document fragment in Figure 4.10.

Figure 4.10 Sample XSD Document

```
<xs:complexType name="category">
<xs:sequence>
      <xs:element name="entry">
         <xs:complexType>
            <xs:attribute name="Name"
               type="xs:string" use="required"/>
            <xs:attribute name="PhoneNumber"
```

Continued

Figure 4.10 Continued

```
                type="xs:string" use="required"/>
            </xs:complexType>
        </xs:element>
    </xs:sequence>
    <xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
```

What Figure 4.10 shows is a very basic XML schema that a developer could write to guarantee that his or her XML data is in the correct format. The only problem with this text-based schema is that it's very complicated and difficult to learn and write. Luckily for those of us who do not necessarily want to spend a lot of our precious time writing these schemas, Microsoft has created a number of tools for us to use to automate this process. These tools include the visual XSD designers available when you add a *DataSet* or *XMLSchema* item to your project. These two tools are invaluable if you need to visually design your own schema. The other major tool Microsoft has provided in the .NET framework is the *System.Xml.Schema* classes.

The *System.Xml.Schema* classes are used to represent an XML schema in an object-oriented, class-based approach. The classes in this namespace can be used both to represent an existing XML schema, and to create a new schema programmatically. Everything that can be done by writing an XML schema by hand can be programmatically created and read using the classes in this namespace. Let's discuss a few of the more basic classes in this namespace.

Like the *System.Xml* classes, the *System.Xml.Schema* classes that are used to represent a schema are all descendants of the *XmlSchemaAnnoted* class (which is itself a subclass of the *XmlSchemaObject* class). The next level in the *XmlSchema* hierarchy is the *XmlSchema* class. The *XmlSchema* class contains the definition for the schema and corresponds to the schema element in the XSD document. All schema elements are added either directly to your Schema object or to a descendant of the Schema object. The *XmlSchema* class contains "Write an XSD document and compile the Schema Object Model (SOM) into schema information for validation."

The *XmlSchemaAttribute* class is used to describe an attribute in your XSD document. You can set the name and schema type of the attribute using public properties on the class. You can also set its default or fixed value depending on how your attribute is meant to work. The *XmlSchemaSimpleType* and *XmlSchemaComplexType* classes are used to represent simple and complex types,

respectively. The main distinction about simple and complex types is that simple types are used to represent a single piece of data, and complex types themselves contain multiple pieces of data. Look at Figure 4.11 to see the *System.Xml.Schema* classes in action (the code in this example can be found at www.syngress.com/solutions in the SchemaProject folder).

Figure 4.11 *System.Xml.Schema* Classes in Action

```
namespace SchemaProject
{
   class Class1
   {
      static void Main(string[] args)
      {
            XmlSchema mySchema = new XmlSchema();

            XmlSchemaSimpleType mySimpleType = new
              XmlSchemaSimpleType();
            mySimpleType.Name = "mySimpleType";

            XmlSchemaComplexType myComplexType = new
              XmlSchemaComplexType();
            myComplexType.Name = "complexType";
            mySchema.Items.Add(myComplexType);

            XmlSchemaAttribute myAttribute = new XmlSchemaAttribute();
            myAttribute.Name = "myAttribute";
            myComplexType.Attributes.Add(myAttribute);

            mySchema.Compile(new
              ValidationEventHandler(ValidationCall));
            mySchema.Write(System.Console.Out);

            System.Console.ReadLine();
      }

      public static void ValidationCall(object sender,
```

Continued

Figure 4.11 Continued

```
        ValidationEventArgs args)
      {
         Console.WriteLine(args.Message);
      }
   }
}
```

The example in Figure 4.11 is about as simple as it gets. It simply creates a new complex type, defines an attribute to be used in the complex type, and then compiles and shows the resulting XSD document. Table 4.8 lists the classes inside the *System.Xml.Schema* namespace.

Table 4.8 *System.XML.Schema* Classes

Class Name	Class Description
XmlSchema	This class stores the definition of a schema.
XmlSchemaAll	The W3C *all* element.
XmlSchemaAnnotated	Base class for *annotation* elements.
XmlSchemaAnnotation	The W3C *annotation* element.
XmlSchemaAny	The W3C *any* element.
XmlSchemaAnyAttribute	The W3C *anyAttribute* element.
XmlSchemaAppInfo	The W3C *appinfo* element.
XmlSchemaAttribute	The W3C *attribute* element.
XmlSchemaAttributeGroup	The W3C *attributeGroup* element with the *ref* attribute.
XmlSchemaChoice	The W3C *choice* element.
XmlSchemaCollection	Stores a cache of XSD and XDR schemas.
XmlSchemaCollectionEnumerator	Allows for simple iteration of the *XmlSchemaCollection*.
XmlSchemaComplexContent	The W3C *complexContent* element.
XmlSchemaComplexContent Extension	The W3C *extension* element.
XmlSchemaComplexContent Restriction	The W3C *restriction* element.

Continued

Table 4.8 Continued

Class Name	Class Description
XmlSchemaComplexType	The W3C *complexType* element.
XmlSchemaContent	Contains an abstract model for the schema content.
XmlSchemaContentModel	Contains an abstract model for the schema content model.
XmlSchemaDatatype	An abstract class for mapping XSD and .NET Framework types.
XmlSchemaDocumentation	The W3C *documentation* element.
XmlSchemaElement	The W3C *element* element.
XmlSchemaEnumerationFacet	Defines enumeration facets and represents the W3C *enumeration* facet.
XmlSchemaException	Returns exceptions from the schema.
XmlSchemaExternal	Returns information on the schema passed to it.
XmlSchemaFacet	Abstract class used for facets when simple types are derived by restriction.
XmlSchemaFractionDigitsFacet	The W3C *fractionDigits* facet.
XmlSchemaGroup	The W3C *group* element.
XmlSchemaGroupBase	Abstract class used for XmlSchemaChoice, XmlSchemaAll, and XmlSchemaSequence.
XmlSchemaGroupRef	The W3C *group* element with the *ref* attribute.
XmlSchemaIdentityConstraint	Represents the *key*, *keyref*, and *unique* elements.
XmlSchemaImport	The W3C *import* element.
XmlSchemaInclude	The W3C *include* element.
XmlSchemaKey	The W3C *key* element.
XmlSchemaKeyref	The W3C *keyref* element.
XmlSchemaLengthFacet	The W3C *length* facet.
XmlSchemaMaxEclusiveFacet	The W3C *maxExclusive* facet.
XmlSchemaMaxInclusiveFacet	The W3C *maxInclusive* element.
XmlSchemaMinLengthFacet	The W3C *minLength* facet.
XmlSchemaNotation	The W3C *notation* element.
XmlSchemaNumericFacet	The W3C *numeric* facets.

Continued

Table 4.8 Continued

Class Name	Class Description
XmlSchemaObject	Creates an empty schema.
XmlSchemaObjectCollection	Handles collections of *XmlSchemaObjects* when returned from methods.
XmlSchemaObjectEnumerator	An enumerator to walk through the collection generation by *XmlSchemaObjectCollection*.
XmlSchemaObjectTable	Allows for read-only helpers for the *XmlSchemaObject* and provides the collections for elements accessible from the *XmlSchema* class.
XmlSchemaParticle	Base class for all particle types.
XmlSchemaPatternFacet	Class for defining pattern facets.
XmlSchemaRedefine	The W3C *redefine* element.
XmlSchemaSequence	The W3C *sequence* element.
XmlSchemaSimpleContent	The W3C *simpleContent* element.
XmlSchemaSimpleContent Extension	A Microsoft extension; adds attributes to the *simpleType* content. The W3C *extension* element for *simpleType* content.
XmlSchemaSimpleContent Restriction	A Microsoft extension; affects an element within a subset of inherited simple types by restricting its range of values.
XmlSchemaSimpleType	The W3C *simpleType* element.
XmlSchemaSimpleTypeContent	*Abstract* class for all classes of *simpleType* content.
XmlSchemaSimpleTypeList	The W3C *list* element; a list of the *simpleType* elements.
XmlSchemaSimpleTypeRestriction	The W3C *restriction* element, but only for *simpleTypes*.
XmlSchemaSimpleTypeUnion	The W3C *union* element.
XmlSchemaTotalDigitsFacet	The W3C *totalDigits* element.
XmlSchemaType	Contains the base class for all *simpleType* and *complexType* classes.
XmlSchemaUnique	The W3C *unique* element.
XmlSchemaWhiteSpaceFacet	The W3C *whiteSpace* facet.
XmlSchemaXPath	The W3C *selector* element from the XPath specification.

Mapping XML DOM on the *System.Xml* Namespace

The next important step in understanding how to use XML in the .NET Framework is to understand how the DOM standards provided by the W3C have been followed and extended by Microsoft's implementation of their *System.Xml* classes.

Since their first XML parser, Microsoft has continually created extensions to the DOM recommendations. These extensions have been implemented with the idea that using these extensions would facilitate working with XML in certain areas. In truth, many of these extensions are really just "shortcuts" to what the XML DOM already provides us with. However, using the Microsoft XML parser is different from using a parser that sticks to the W3C DOM recommendations, so do not expect to be able to just hop over to another parser, thinking it will work the same as it has via .NET. The only noticeable changes from the W3C DOM in Microsoft are cosmetic. For example, their implementation of the DOM interface *node*, the *nodeName*, and *nodeValue* properties have been renamed to *Name* and *Value*, respectively. While examining the classes in the *System.Xml* namespace, you might find other places where method and attribute names vary from the W3C DOM interfaces, but rest assured, Microsoft did build in all the functionality that the DOM recommendations define. See Table 4.9 for a quick reference to the relationships between the DOM interface and Microsoft's implementation.

Table 4.9 DOM Interfaces Compared to the *System.Xml* Classes

DOM Interface	System.Xml Class
Node	*XmlNode*
Attribute	*XmlAttribute*
CDATASection	*XmlCDataSection*
CharacterData	*XmlCharacterData*
Comment	*XmlComment*
Document	*XmlDocument*
DocumentFragment	*XmlDocumentFragment*
DocumentType	*XmlDocumentType*
Element	*XmlElement*
Entity	*XmlEntity*
EntityReference	*XmlEntityReference*
NamedNodeMap	*XmlNamedNodeMap*
NodeList	*XmlNodeList*

Continued

Table 4.9 Continued

DOM Interface	System.Xml Class
Notation	*XmlNotation*
ProcessingInstruction	*XmlProcessingInstruction*
Text	*XmlText*

As you can see, every interface provided in the DOM Level 1 Core and DOM Level 2 Core has a corresponding class in the *System.Xml* namespace. If you examine the classes presented in Table 4.8, you should also notice that every method and property defined in the W3C recommendations has a corresponding method or property in the related class in the *System.Xml* namespace.

The extensions that Microsoft provides for XML are listed in Table 4.10. Remember, these extensions are not present in the DOM recommendation.

Table 4.10 *System.Xml* Classes Not Present in DOM

System.Xml Class	Description
XmlAttributeCollection	This class inherits from *XmlNamedNodeMap* and is a slight change from the DOM recommendation. An instance of this class is returned from the *Node.attributes* property. This class provides additional functionality from the *NamedNodeMap* interface Microsoft felt was necessary.
XmlConvert	This is mostly a utility class used to transform data between CLR and XSD types.
XmlDataDocument	This class inherits from *XmlDocument*, and is a specialized class when using a *DataSet* and you wish to perform XML operations on it.
XmlDeclaration	Represents the XML declaration node *<?xml version="1.0"...?>*.
XmlImplementation	Defines the context for a set of XML documents. This class provides a single method to create instances of *XmlDocuments*.
XmlQualifiedName	This class represents a fully qualified XML name in the form of *namespace:localname*.
XmlTextReader	Provides forward-only cursor-like read access to an XML document.
XmlTextWriter	Provides forward-only cursor-like write access to an XML document.

These classes are there mostly to extend and provide additional access to XML documents that the W3C has either not thought of or has decided not to include in their recommendations.

Explaining a Selection of *System.Xml* Classes

Now that you've seen how the DOM corresponds to the *System.Xml* classes, let's look at the most common of these classes. The first class we are going to look at is *XmlNode*. As you learned from examining the .NET DOM, every node type inherits directly from the *XmlNode* class. The *XmlNode* class is abstract, which means that you cannot directly create an *XmlNode* class; instead, you must create one of the classes that subclass *XmlNode*. See Table 4.11 for a subset of *XmlNode*'s properties and methods.

Table 4.11 *XmlNode* Properties and Methods

Type	Name	Description
Property	*Attributes*	Gets an *XmlAttributeCollection* containing the attributes of this node.
Property	*ChildNodes*	Gets the children of this node.
Property	*FirstChild*	Gets the first child of this node.
Property	*HasChildNodes*	Returns a Boolean value indicating whether the node has any child nodes.
Property	*InnerText*	Gets or sets the value of this node and all children.
Property	*InnerXml*	Gets or sets the markup representing the children of this node.
Property	*LastChild*	Gets the last child of this node.
Property	*Name*	Gets the name of this node.
Property	*NextSibling*	Gets the node immediately following this node.
Property	*NodeType*	Gets the type of the current node.
Property	*OuterXml*	Gets the XML representing this node and all of its children.
Property	*ParentNode*	Gets the parent of this node.
Property	*PreviousSibling*	Gets the node immediately before this node.
Property	*Value*	Gets or sets the value of this node.

Continued

Table 4.11 Continued

Type	Name	Description
Method	*AppendChild()*	Adds the specified node to the end of the list of children of this node.
Method	*InsertAfter()*	Inserts the specified node immediately after the context node.
Method	*InsertBefore()*	Inserts the specified node immediately before the context node.
Method	*PrependChild()*	Adds the specified node to the beginning of the list of children of this node.
Method	*RemoveAll()*	Removes all children from this node.
Method	*RemoveChild()*	Removes the specified child from this node.
Method	*ReplaceChild()*	Replaces the old child node with a new child node.
Method	*SelectNodes()*	Selects a list of nodes that match an XPath expression.
Method	*SelectSingleNode()*	Selects the first node that matches an XPath expression.

Remember that *XmlNode* is the base class for all node types in a .NET XML document, so every method in *XmlNode* is available in any node that inherits from it. The next node type to look at is the second most important when working with the *System.Xml* namespace, the *XmlDocument* class. See Table 4.12 for the *XmlDocument*'s properties and methods.

Table 4.12 *XmlDocument* Methods and Properties

Type	Name	Description
Property	*DocumentElement*	Gets the root *XmlElement* in the XML document.
Property	*DocumentType*	Gets the node containing the DOCTYPE declaration.
Property	*InnerXML*	Overridden version of the *XmlNode*'s implementation.
Property	*Name*	Overridden version of the *XmlNode*'s implementation.

Continued

Table 4.12 Continued

Type	Name	Description
Property	*NodeType*	Overridden version of the *XmlNode*'s implementation.
Property	*Preserve Whitespace*	Gets or sets the value indicating whether to preserve whitespace.
Method	*CloneNode*	Overridden version of the *XmlNode*'s implementation.
Method	*CreateAttribute*	Creates an *XmlAttribute* with the specified name.
Method	*CreateCData Section*	Creates an *XmlCDataSection* containing the specified data.
Method	*CreateComment*	Creates an *XmlComment* containing the specified text.
Method	*CreateDocument Fragment*	Creates an *XmlDocumentFragment*.
Method	*CreateDocument Type*	Creates an *XmlDocumentType* object.
Method	*CreateElement*	Creates an *XmlElement* object.
Method	*CreateNode*	Creates an *XmlNode* object.
Method	*CreateTextNode*	Creates an *XmlTextNode* object with the specified text.
Method	*GetElementsBy TagName*	Returns an *XmlNodeList* containing all the descendant elements with the specified name.
Method	*Load*	Loads the *XmlDocument*.
Method	*LoadXml*	Loads the *XmlDocument* from the specified string.
Method	*Save*	Saves the *XmlDocument*.

The *XmlDocument* class contains methods and properties to create, load, save, and edit an *XmlDocument*. Most of the interaction of the nodes contained in an *XmlDocument* are managed through the *XmlNode* class's properties and methods. See Figure 4.12 for an example of how to use the *XmlDocument* and *XmlNode* class to create a simple XML document (the code for this can be found in the *XmlDocumentProject* folder at www.syngress.com/solutions).

Figure 4.12 Simple *XmlDocument* Class

```
using System;
using System.Xml;

namespace XmlDocumentProject
{
   public class Class1
   {
      public Class1()
      {
      }
      public static void Main()
      {
         XmlDocument myDoc = new XmlDocument();
         XmlProcessingInstruction myProc =
            myDoc.CreateProcessingInstruction("xml",
               "version=\"1.0\"");
            myDoc.AppendChild(myProc);
         XmlElement myRoot = myDoc.CreateElement("rootNode");
         myDoc.AppendChild(myRoot);

         XmlElement myElement = myDoc.CreateElement("firstSubElement");
         myRoot.AppendChild(myElement);

         Console.WriteLine(myDoc.OuterXml);
         Console.Write("Press enter to finish...");
         Console.ReadLine();
      }
   }
}
```

The *XmlDocumentProject* is about as simple as it gets when working with XML data. In the first line of the *Main* method, an *XmlDocument* is created. Next, the program creates a processing instruction (the *<? ?>* tag) and adds it to the *XmlDocument*, although this is optional as the *XmlDocument* will still function

without it. Next, a root element is created with the name *rootNode* and is added to the *XmlDocument*. Finally, another element is created and added to the root node, and then the XML is written to the console window. The output of *myDoc.OuterXml* looks like this:

```
<?xml version="1.0"?><rootNode><firstSubElement /></rootNode>
```

The next class that is important to mention is the *XmlDataDocument* class. *XmlDataDocument* inherits from *XmlDocument* and is used to represent relational data obtained from a *DataSet* in an XML format. Since the *DataSet* is represented internally by the .NET framework as XML data, the transformation to and from formats and working with the resulting data is much easier than if you attempted to do the same thing with ADO.NET. Essentially, the *XmlDataDocument* has all of the same methods and properties as the *XmlDocument*, but some methods are overridden in order to more properly work with the underlying data structure. In addition to the overrides, there are a small number of methods and properties that are new to the *XmlDataDocument* class. See Table 4.13 for a list of these methods and properties.

Table 4.13 *XmlDataDocument* Methods and Properties

Type	Name	Description
Property	*DataSet*	Get the underlying *DataSet* for the XML data.
Method	*GetElementFromRow*	Get the *XmlElement* associated with a given *DataRow*.
Method	*GetRowFromElement*	Get the *DataRow* associated with a given *XmlElement*.

As you can see, there isn't very much more to the *XmlDataDocument* than the *XmlDocument*. The additional methods and the single property are there solely to allow you to work with the data in the *DataSet*. See Figure 4.13 for an example of how to work with the *XmlDataDocument* (the code for this can be found in the XmlDataDocumentProject folder at www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 4.13 *XmlDataDocument* Example

```
using System;
using System.Xml;
using System.Data;
```

Continued

Figure 4.13 Continued

```
using System.Data.OleDb;

namespace XmlDataDocumentProject
{
   class Class1
   {
      static void Main(string[] args)
      {
         OleDbConnection myConn =
            new OleDbConnection("Provider=SQLOLEDB.1;" +
            "Data Source=localhost;Initial Catalog=Pubs;" +
            "User ID=sa;Pwd=;");
         OleDbCommand myCommand = new OleDbCommand(
            "SELECT * FROM Authors", myConn);
         OleDbDataAdapter myAdapter = new OleDbDataAdapter(myCommand);
         DataSet myData = new DataSet();
         myAdapter.Fill(myData);

         XmlDataDocument myDoc = new XmlDataDocument(myData);
         Console.WriteLine(myDoc.OuterXml);
         Console.Write("Press enter to finish...");
         Console.ReadLine();
      }
   }
}
```

The first few lines in the code establish a connection to the PUBS database on the local instance of SQL Server. Next, a command object is built with the SQL statement *SELECT * FROM Authors*, and a *DataAdapter* is created and a *DataSet* filled with the resulting data. Next, an *XmlDataDocument* is created and the *DataSet* is passed as a constructor parameter to the *XmlDataDocument*. Finally, the *OuterXml* of the *XmlDataDocument* is written to the console. The following XML is a sample of what is returned:

```
<NewDataSet>
   <Table>
      <au_id>172-32-1176</au_id>
      <au_lname>White</au_lname>
      <au_fname>Johnson</au_fname>
      <phone>408 496-7223</phone>
      <address>10932 Bigge Rd.</address>
      <city>Menlo Park</city>
      <state>CA</state>
      <zip>94025</zip>
      <contract>true</contract>
   </Table>
</NewDataSet>
```

Designing & Planning…

Deciding Which Class to Use to Represent Your XML Data

The *System.Xml* namespace provides a number of classes that represent XML documents, and deciding what to use and how to use it should be a design consideration. The *XmlDocument* class is available for access to a text-based set of XML data. If you have XML passed to your class as a parameter, or you are able to directly access an XML document on a file system, then the *XmlDocument* class is definitely the correct choice. If you have a streaming model to your XML data, then the *XmlTextReader* might be a good class to use to access the streaming data. If you are receiving XML from a database, then the *XmlDataDocument* would be a good choice, although you do have several methods from which to choose to retrieve this data (*ExecuteReader* returns an *XmlReader*, the *SQLXML* classes allow use of the *FOR XML* clause in SQL statements) when using this method.

As you can see, the *XmlDataDocument* can be a very valuable addition for those of us who want to turn our relational data into a readable XML format. When working with ADO in the pre-.NET era, turning a *RecordSet* into XML and then attempting to read the resulting XML was something only the bravest or most ignorant did. The only thing you could really do with the resulting XML is save it to the

file system or pass it to another object somewhere that would turn the data back into a *RecordSet* and work with it that way, completely ignoring the XML. Luckily for those of us who had worked with the XML support in ADO, Microsoft has provided us with a much better way to work with relational data in XML format.

The next important node type to talk about is the *XmlElement* class. The *XmlElement* class represents any element in your XML document (anything with < and > surrounding it). *XmlElement* inherits from *XmlNode*, so it has the functionality of the *XmlNode* built directly into it, and it provides a small number of methods and properties in addition to what *XmlNode* provides. See Table 4.14 for a list of these methods and properties.

Table 4.14 *XmlElement* Additional Methods and Properties

Type	Name	Description
Property	*HasAttributes*	Indicates whether this element has attributes.
Property	*IsEmpty*	Gets or sets the tag format of this element.
Method	*GetAttribute*	Gets the attribute value for the specified attribute.
Method	*GetAttributeNode*	Gets the *XmlAttribute* node for the specified attribute.
Method	*GetElementsByTagName*	Gets a list of all descendant elements that match the specified tag name.
Method	*HasAttribute*	Indicates whether this element has the specified attribute.
Method	*RemoveAllAttributes*	Removes all attributes from this element.
Method	*RemoveAttribute*	Removes the specified attribute.
Method	*RemoveAttributeAt*	Removes the attribute at the specified index.
Method	*RemoveAttributeNode*	Removes the specified *XmlAttribute*.
Method	*SetAttribute*	Sets the value of the specified attribute.
Method	*SetAttributeNode*	Adds a new *XmlAttribute* (or replaces an existing one).

The *XmlElement* class's methods and properties provide additional functionality on top of what *XmlNode* provides for dealing with attributes and child nodes. The next node type we are going to discuss is the *XmlAttribute* class, since

it pretty much goes hand in hand with the *XmlElement* class. The *XmlAttribute* class represents an attribute in an XML element. The *XmlAttribute* class inherits from *XmlNode*, and also provides a bit of additional functionality on top of what *XmlNode* provides. See Table 4.15 for a list of these properties.

Table 4.15 *XmlAttribute* Additional Properties

Type	Name	Description
Property	*OwnerElement*	Gets the *XmlElement* that contains this attribute.
Property	*Specified*	Gets or sets a value indicating if this attribute value was explicitly set.

Being a relatively simple type of node in an XML document, the *XmlAttribute* class doesn't have much specialized functionality. See Figure 4.14 for an example of how to use the *XmlElement* and *XmlAttribute* classes (the source code for this can be found in the XmlAttributeElementProject folder at www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 4.14 *XmlElement* and *XmlAttribute* Example

```
using System;
using System.Xml;

namespace XmlAttributeElementProject
{
   class Class1
   {
      static void Main(string[] args)
      {
         XmlDocument myDoc = new XmlDocument();
         XmlElement myRoot = myDoc.CreateElement("rootElement");
         myDoc.AppendChild(myRoot);

         XmlElement myElement = myDoc.CreateElement("subElement");
         myRoot.AppendChild(myElement);
```

Continued

Figure 4.14 Continued

```
            XmlElement mySecondElement =
               myDoc.CreateElement("subSubElement");

            XmlAttribute myAttribute = myDoc.CreateAttribute("attribute");
            myAttribute.Value = "this is my attribute value";

            myElement.SetAttributeNode(myAttribute);
            myElement.SetAttribute("attribute2",
               "this is my second attribute value");

            myElement.AppendChild(mySecondElement);

            Console.WriteLine(myDoc.OuterXml);
            Console.Write("Press enter to finish...");
            Console.ReadLine();
         }
      }
}
```

The first step in the code is to create an *XmlDocument* and add a root element to it. Next, two additional *XmlElements* are created, *subElement* and *subSubElement*. Next, a new attribute is created from the *XmlDocument* object and its value is set to "this is my attribute value". Next, the attribute that was just created is added using the *SetAttributeNode* method of the *XmlElement* class, and another attribute is added using the *SetAttribute* method. Finally, the *mySecondElement* object is added as a child to the *myElement* object, and the entire document's XML is written to the console:

```
<rootElement>
   <subElement attribute="this is my attribute value"
         attribute2="this is my second attribute value">
      <subSubElement />
   </subElement>
</rootElement>
```

All this discussion about how to create and load XML documents has been great, but we're sure you want to know how to take a document, find a group of elements, and deal with the data contained in them. There are a number of methods available in the classes we've discussed thus far to select a list of nodes based on an XPath query or simply by knowing a given tag name. The class these methods return is *XmlNodeList*. The *XmlNodeList* class is just a live list of nodes that are represented in list form instead of in the DOM hierarchy. *XmlNodeList* does very little more than any other list class you've used in the *System.Collections* namespace, but it is only returned by classes in the System.Xml namespace. There is no need to list its methods and properties, as most of them are inherited from either System.Object or System.Collections.IEnumerable, so instead, let's just delve directly into an example of how to use the XmlNodeList in Figure 4.15 (the code for this example can be found in the XmlNodeListProject folder at www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 4.15 *XmlNodeList* Example

```
using System;
using System.Xml;
using System.Data;
using System.Data.OleDb;

namespace XmlNodeListProject
{
class Class1
{
   static void Main(string[] args)
   {
      OleDbConnection myConn =
         new OleDbConnection("Provider=SQLOLEDB.1;" +
         "Data Source=localhost;Initial Catalog=Pubs;" +
         "User ID=sa;Pwd=;");
      OleDbCommand myCommand = new OleDbCommand(
         "SELECT TOP 3 * FROM Authors", myConn);
      OleDbDataAdapter myAdapter = new OleDbDataAdapter(myCommand);
      DataSet myData = new DataSet();
```

Continued

Figure 4.15 Continued

```
        myAdapter.Fill(myData);

        XmlDataDocument myDoc = new XmlDataDocument(myData);

        XmlNodeList myList = myDoc.GetElementsByTagName("Table");
        for(int x = 0; x < myList.Count; x++)
        {
        Console.Write("Element #" + x.ToString());
            Console.Write("\t");
            Console.Write(myList[x].Name + " element ");
            Console.Write("has " + myList[x].Attributes.Count +
                " attributes");
            Console.Write("\n\t");
            Console.Write(" and has " + myList[x].ChildNodes.Count +
                " child nodes");
            if(myList[x].ChildNodes.Count > 0)
            {
                Console.WriteLine();
                foreach(XmlNode myNode in myList[x].ChildNodes)
                {
                    Console.Write(myNode.Name + " ");
                }
            }
            Console.WriteLine();
        }
        Console.Write("Press enter to finish...");
        Console.ReadLine();
    }
}
}
```

Designing & Planning…

Using foreach on the XmlNodeList Class

One important thing to remember when dealing with an *XmlNodeList* is that it is a live list. When you get an *XmlNodeList* from an *XmlDocument* object, iterating through the child nodes using a foreach structure proves to be effortless and has no problems.

```
XmlDocument myDoc = new XmlDocument();
…
XmlNodeList myList = myDoc.SelectNodes("/node1/node2");
foreach(XmlNode myNode in myList)
{
   Console.WriteLine(myNode.Name);
}
```

However, when retrieving an *XmlNodeList* from an *XmlDataDocument*, it is still a live list of data, but the nodes in the list are not created until requested. That is, when an *XmlNodeList* is created from an *XmlDocument*, the nodes are not brought into the list until the moment they are requested (it functions like this in order to reduce any overhead of relating data in the *DataSet* to data in the list of nodes. For example:

```
XmlDataDocument myDoc = new XmlDataDocument(myDataSet);
XmlNodeList myList = myDoc.SelectNodes("/Table/Au_id");
foreach(XmlNode myNode in myList)
{
   Console.WriteLine(myNode.Name);
}
```

This code will always fail in Beta 2 and Release Candidate 1 (RC1), and raise an exception detailing how the members of the list have changed since the foreach iteration began. Microsoft knows about this problem, and has an easy workaround: use an explicit for loop instead of the foreach loop and your code will work fine:

```
XmlDataDocument myDoc = new XmlDataDocument(myDataSet);
XmlNodeList myList = myDoc.SelectNodes("/Table/Au_id");
for(int x = 0; x < myList.Count; x++)
{
   XmlNode myNode = myList[x];
   Console.WriteLine(myNode.Name);
}
```

The *XmlDataDocument* is used in this case simply to get a good amount of data to work with without needing to have an XML document on hand. Loading an XML document using the *XmlDocument* instead of the *XmlDataDocument* would be a very minor change, and would simply involve changing the way you load the XML. The rest would be exactly the same.

After we've retrieved the *XmlDataDocument*, we call the *GetElementsByTagName* method and choose to retrieve every *Table* element in the XML. Next, we loop through the *XmlNodeList*, writing out the pertinent information on every node that appears in the list. As you can see, there isn't that much to the *XmlNodeList*, since it is essentially nothing but a list class, like the list classes in the *System.Collections* namespace.

Using the System.Xml Namespace

This section of the chapter provides you with a thorough example of how to use the *System.Xml* namespace and the classes in this namespace. This exercise will show you how to create and load an XML document using the *XmlDocument* class, select multiple and single nodes in the document, edit node values, delete nodes, and finally, save the XML document to the file system.

The exercise you are about to read is a simple address book using XML as the data store. The address book will store a list of categories and a list of category entries associated with individual categories. It will have the capabilities to handle multiple address books using multiple XML files, create the base XML files automatically, and be able to add, edit, and remove nodes using the DOM-compliant *System.Xml* classes.

Building the XML Address Book

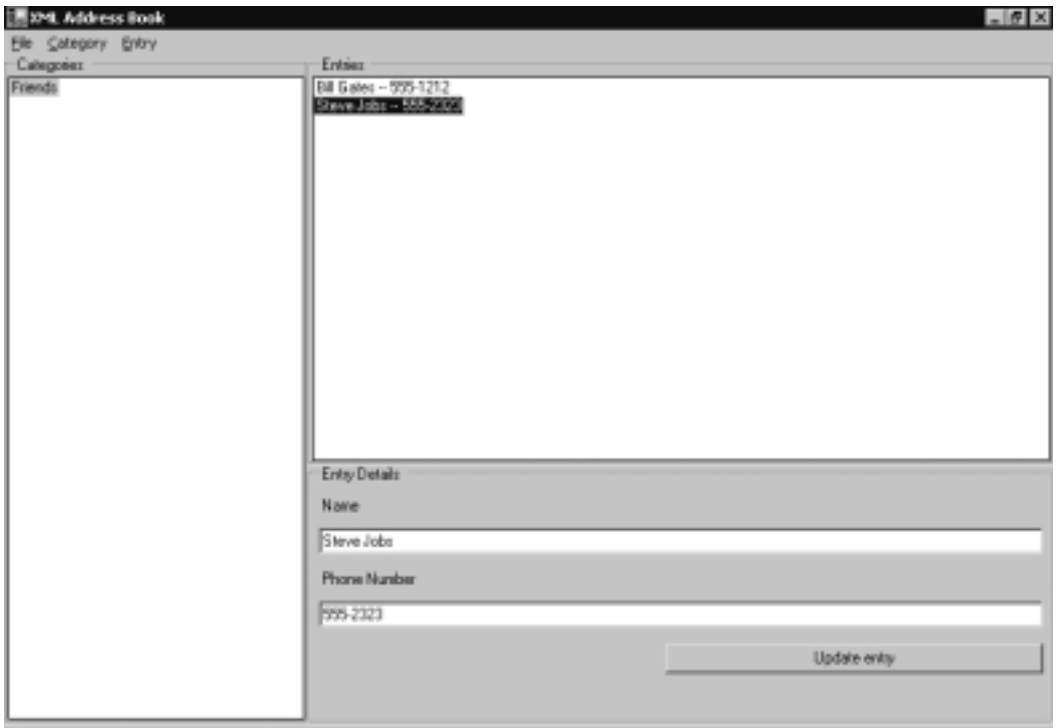
The XML Address Book is a C# Windows application that can be found at www.syngress.com/solutions in the XmlAddressBook folder. It uses three forms to accept user input and to display your address book grouped by categories. When you select a category, it displays the associated entries, and when you select an entry it displays the entry information and allows you to update the entry you selected. See Figure 4.16 for a screenshot of the user interface of the application.

Loading the XML Address Book

The first step in creating your address book is to define the format of your XML data. In this case, we've already seen the format of the XML in Figure 4.1 earlier

in this chapter. We will have a top-level *addressBook* element, multiple *category* elements that are children of *addressBook*, and multiple *entry* elements that are children of a *category* element.

Figure 4.16 XML Address Book GUI



Now that our data format has been defined, the next step is to create a new XML Address Book file. When the users clicks **File | New** (or presses **CTRL + N**), he or she will be prompted with a File Save dialog and will have the choice of where to save the file. Figure 4.17 shows the code for this.

Figure 4.17 Creating a New XML Address Book File

```
private void mnuFileNew_Click_1(object sender, System.EventArgs e)
{
   saveFileDialog1.CheckFileExists = false;
   saveFileDialog1.CheckPathExists = true;
   saveFileDialog1.Filter = "XML Address book files (*.xmlab)|*.xmlab";
```

Continued

Figure 4.17 Continued

```
    saveFileDialog1.OverwritePrompt = true;
    DialogResult result = saveFileDialog1.ShowDialog();

    if(result.ToString().ToLower() == "ok")
    {
        this.myDocument = new XmlDocument();
        XmlProcessingInstruction myProc =
            myDocument.CreateProcessingInstruction(
                "xml", "version='1.0'");
        XmlElement myElement = myDocument.CreateElement("addressBook");
        myDocument.AppendChild(myProc);
        myDocument.AppendChild(myElement);
        myDocument.Save(saveFileDialog1.FileName);
        this.loadAddressBook(saveFileDialog1.FileName);
    }
}
```

After the user has successfully selected and named a file to create, a new *XmlDocument* is created (this is a class-level variable). A processing instruction is added (the *<?xml version="1.0"?>* instruction), and the root *addressBook* element is added. Finally, the *myDocument* variable is saved and the *loadAddressBook()* method is called.

A very similar process happens when a user wants to open an existing address book file from the file system. See Figure 4.18.

SYNGRESS
syngress.com

Figure 4.18 Opening an Existing XML Address Book File

```
private void mnuFileOpen_Click(object sender, System.EventArgs e)
{
    openFileDialog1.CheckFileExists = true;
    openFileDialog1.CheckPathExists = true;
    openFileDialog1.Filter = "XML Address book files (*.xmlab)|*.xmlab";
    openFileDialog1.Title = "Open Address Book file";
```

Continued

Figure 4.18 Continued

```
    DialogResult res = openFileDialog1.ShowDialog();

    if(res.ToString().ToLower() == "ok")
    {
        this.lstCategories.Items.Clear();
        this.lstEntries.Items.Clear();
        this.loadAddressBook(openFileDialog1.FileName);
        saveFileDialog1.FileName = openFileDialog1.FileName;
    }
}
```

This method first prompts the user for a valid file to open. After a valid file is opened, the user interface list boxes are cleared and the *loadAddressBook()* method is called. The *loadAddressBook()* method is the main method for initializing the user interface. It adds the categories to the category list box and instantiates the class-level *XmlDocument* that stores the address book. Figure 4.19 shows the code for *loadAddressBook()*.

SYNGRESS
syngress.com

Figure 4.19 The *loadAddressBook()* Method

```
private void loadAddressBook(string path)
{
    this.initMenuItems(true);
    myDocument = new XmlDocument();
    myDocument.Load(path);
    XmlNodeList categories = this.myDocument.GetElementsByTagName("category");
    foreach(XmlNode category in categories)
    {
        this.addCategory(category.Attributes["name"].Value, false);
    }
}
```

First, the menu items are initialized by calling the *initMenuItems* method. This method simply enables certain menu items that wouldn't be valid if there were no address book currently loaded. Next, the *myDocument* variable is instantiated and

the XML file is loaded from the supplied path. Next, a list of *category* elements is retrieved from the documents using the *GetElementsByTagName* method in the *XmlDocument* class. Finally, each category is added to the UI using the *addCategory()* method.

Creating and Deleting Categories

The next step in creating the XML Address Book application is to create your categories. A separate form is created for this in your project called AddCategory.cs, and can be found at www.syngress.com/solutions in the XmlAddressBook folder. The sole purpose of this form is to collect a category name and verify that the name entered is not empty. After it does this, it calls the *addCategory()* method of the MainForm class, shown in Figure 4.20 and in XMLAddressBook folder at www.syngress.com/solutions.

Figure 4.20 The *addCategory* Method

```
public void addCategory(string name, bool isNew)
{
   this.lstCategories.Items.Add(name);
   if(isNew)
   {
      XmlElement newCat = myDocument.CreateElement("category");
      XmlAttribute newName = myDocument.CreateAttribute("name");
      newName.Value = name;

      newCat.Attributes.Append(newName);
      myDocument.DocumentElement.AppendChild(newCat);
   }
   this.clearEntryDetail();
}
```

The first step in this method is to add the requested category to the category list box. If the category is new (if it should be added to the XML document), then a new *XmlElement* is created and an *XmlAttribute* is added. You will end up with an XML element that looks like this:

```
<category name="The name parameter"></category>
```

Finally the new category element is added to the end of the child nodes of the *DocumentElement* of the *XmlDocument* (remember that the *addressBook* node is the root node, or document element). Finally, the entry detail text boxes are cleared to clean up in case the user clicked on an entry before adding the new category.

The next action to consider is how to delete an existing category from your address book. The user selects a category, and clicks the **Delete** menu item from the **Category** menu. This code is shown in Figure 4.21, and can also be found in the XMLAddressBook folder at www.syngress.com/solutions.

Figure 4.21 Deleting a Category

```
private void mnuCategoryDelete_Click(object sender, System.EventArgs e)
{
   if(this.lstCategories.SelectedItems.Count > 0)
   {
      string category = this.lstCategories.SelectedItems[0].Text;
      string xpath = "/addressBook/category[@name='" +
         category +
         "']";
      XmlNode myNode = myDocument.SelectSingleNode(xpath);
      myNode.ParentNode.RemoveChild(myNode);

      this.lstCategories.SelectedItems[0].Remove();
      this.lstEntries.Clear();
      this.clearEntryDetail();
   }
   else
   {
      MessageBox.Show("You must select a category");
   }
}
```

After verifying that a category has indeed been selected, an XPath expression is built in order to select the appropriate category from the XML document. This XPath expression ends up looking like */addressBook/category[@name='Selected Category']*. Next, that category is selected and it is removed from the *XmlDocument* by calling its parent node's *RemoveChild()* method, passing in itself as the child to

be removed. Finally, the category and entry list boxes are cleared, and the entry detail text boxes are cleared.

Creating, Editing, and Deleting Entries

Now that we have finished with creating and deleting categories, the next step is to be able to add, edit, and delete entries from each category. For adding entries, a form has been created to accept the entry name and telephone number and verify that each is not an empty string (this form is named AddEntry.cs and can be found in the XmlAddressBook folder at www.syngress.com/solutions). After a category has been selected, and an entry name and telephone number have been entered, the *addEntry()* method of the *MainForm* class is called. The code for this method can be found in Figure 4.22.

Figure 4.22 The *addEntry* Method

```
public void addEntry(string categoryName, string entryName,
   string phoneNumber, bool isNew)
{
      string entry = entryName + " -- " + phoneNumber;
      this.lstEntries.Items.Add(entry);
      if(isNew)
      {
         XmlNode oldCat = myDocument.SelectSingleNode(
            "/addressBook/category[@name='" + categoryName + "']");
         XmlElement newEntry = myDocument.CreateElement("entry");
         XmlAttribute newName = myDocument.CreateAttribute("name");
         newName.Value = entryName;
         XmlAttribute newPhone = myDocument.CreateAttribute("phoneNumber");
         newPhone.Value = phoneNumber;

         newEntry.Attributes.Append(newName);
         newEntry.Attributes.Append(newPhone);
         oldCat.AppendChild(newEntry);
      }
      this.clearEntryDetail();
}
```

The first step in this method is to create the display version of the entry for the entry list box, which is simply the name and telephone number separated by " – ", and add it to the entry list box. If the entry is new (that is, it should be added to the XML document), the entry is added to the XML document. First, we retrieve a reference to the selected category to which we are adding an entry. Next, an entry element is created and the name and phone number attributes are created, given a value, and added to the entry element. This element ends up looking like:

```
<entry name="Name parameter" phoneNumber="Phone Number parameter" />
```

Finally, the new entry element is added to the list of children of the category, and the entry detail text boxes are cleared to clean up the screen. The next thing to look at is what happens when a user clicks on an item in the category list. When the user clicks on a category, the entries in that category should appear in the entry list box. See Figure 4.23 and the XmlAddressBook folder at www.syngress.com/solutions for the code to do this.

SYNGRESS
syngress.com

Figure 4.23 Filling the Entries List Box

```
private void lstCategories_Click(object sender, System.EventArgs e)
{
if(this.lstCategories.SelectedItems.Count > 0)
{
   string category = this.lstCategories.SelectedItems[0].Text;
   string xpath = "/addressBook/category[@name='" +
      category +
      "']/entry";

   this.lstEntries.Items.Clear();
   XmlNodeList entries = myDocument.SelectNodes(xpath);
   foreach(XmlNode entry in entries)
   {
      string name = entry.Attributes["name"].Value;
      string phoneNumber = entry.Attributes["phoneNumber"].Value;
      this.addEntry(category, name, phoneNumber, false);
   }
   this.clearEntryDetail();
}
}
```

After it is verified that the user has selected a category, an XPath expression is built to select this category. The XPath here is the same as when a user deletes a category, except this expression selects all the *entry* nodes in the chosen category. Next, it loops through the *XmlNodeList* returned by executing the XPath expression, retrieving the name and telephone number for each entry and calling the *addEntry()* method for each.

When a user clicks on an entry, the bottom pane of information will be filled in with the details for that entry to allow the user to edit the entry and update the address book. The event that gets fired when the user clicks on an entry simply breaks apart the entry in the entry list box and fills in the two text boxes with the values from the entry. The user can make changes and then click the **Update Entry** button to update the XML document with the values in the text boxes. See Figure 4.24.

SYNGRESS
syngress.com

Figure 4.24 Editing an Entry

```
private void btnUpdate_Click(object sender, System.EventArgs e)
{
if(this.txtName.Text.Trim() != "" &&
   this.txtNumber.Text.Trim() != "")
{
   string category = this.lstCategories.SelectedItems[0].Text;
   string entry = this.lstEntries.SelectedItems[0].Text;
   string name = entry.Substring(0, entry.IndexOf(" -- "));

   string xpath = "/addressBook/category[@name='" +
      category + "']/entry[@name='" +
      name + "']";

   XmlNode entryNode = myDocument.SelectSingleNode(xpath);
   entryNode.Attributes["name"].Value = this.txtName.Text.Trim();
   entryNode.Attributes["phoneNumber"].Value = this.txtNumber.Text.Trim();

   this.lstEntries.SelectedItems[0].Text = this.txtName.Text.Trim() +
      " -- " + this.txtNumber.Text.Trim();
```

Continued

Figure 4.24 Continued

```
}
else
{
   MessageBox.Show("You must enter a name and a number.");
}
}
```

After the text boxes have been validated (no empty strings allowed), an XPath expression is built to find the entry the user is editing. The XPath for this looks like:

```
/addressBook/category[@name='Selected Category']/entry[@name='Selected
Entry']
```

Next, the entry element is selected, and its *name* and *phoneNumber* attributes are set to what the user entered into the Name and Phone Number text boxes. Finally, the entry in the entry list box is updated to what the user entered.

The last feature needed in the XML Address Book is the capability to delete entries. Deleting entries is incredibly similar to everything you've seen so far. The process is simple: Select a category, select an entry, and press the **Delete** button. The code to delete an entry is shown in Figure 4.25.

SYNGRESS
syngress.com

Figure 4.25 Deleting an Entry

```
private void mnuEntryRemove_Click(object sender, System.EventArgs e)
{
   if(this.lstEntries.SelectedItems.Count > 0)
   {
      string category = this.lstCategories.SelectedItems[0].Text;
      string entry = this.lstEntries.SelectedItems[0].Text;
      string name = entry.Substring(0, entry.IndexOf(" -- "));

      string xpath = "/addressBook/category[@name='" +
      category + "']/entry[@name='" +
      name + "']";

      XmlNode entryNode = myDocument.SelectSingleNode(xpath);
```

Continued

Figure 4.25 Continued

```
        entryNode.ParentNode.RemoveChild(entryNode);

        this.lstEntries.SelectedItems[0].Remove();
        this.clearEntryDetail();
    }
    else
    {
        MessageBox.Show("You must select an entry");
    }
}
```

After it has been verified that the user selected an entry, an XPath expression is built to select the specific entry element the user is deleting, which is exactly the same as the XPath expression built to find a node to edit. Next, the specific entry element is selected, and its parent node's *RemoveChild()* method is called, passing itself as the child to remove. Finally, the entry list box is updated and the entry detail text boxes are cleared so the user doesn't attempt to edit the newly deleted entry.

Summary

This chapter covered the very basics of the Document Object Model (DOM) for XML created by the World Wide Web Consortium (W3C). There are multiple levels defined by the W3C in the DOM, including DOM Level 1 Core, DOM Level 2 Core, DOM Range, DOM Traversal, and DOM XPath. DOM Level 1 Core and DOM Level 2 Core are, as their names suggest, the core of the DOM. They define the interfaces that must be adhered to in order to claim that an XML parser is compliant with DOM Level 1 and/or DOM Level 2. The interfaces provided by the Core DOM levels include the Document, Node, Element, and Attribute interfaces. These are the most basic and frequently used interfaces when working with XML data.

The DOM Range and DOM Traversal specifications are provided by the W3C to enable common access to editing and navigating nodes in an XML document. DOM Range is used to define a way for a developer to programmatically access data in an XML document without needing to navigate the DOM hierarchy. DOM Traversal was created to enable a developer to navigate the node hierarchy without necessarily needing to use the navigation built into the DOM interfaces. *NodeList* is provided to enable you to create a list of nodes and navigate through each node without manually building a list of nodes; instead, you simple provide an XPath expression and let the parser do the work for you. The *TreeWalker* is provided to allow a slightly more robust navigation of nodes in manner similar to what the DOM Core interfaces provide. The *TreeWalker* can navigate in any direction in a node tree, and also provides access to navigating deleted nodes even after they have been deleted from the node tree.

Microsoft implemented DOM Level 1 Core, DOM Level 2 Core, and the *NodeList* interface when building their *System.Xml* classes. Every interface is fully supported, including every property and method defined by the W3C. Microsoft has made additions to the interfaces that the W3C recommends in order to make development with XML easier for the developer. They added a number of classes, properties, and methods to slightly simplify access to data that might otherwise be more cumbersome in a strict DOM implementation.

We went through an application using the *System.Xml* classes that created an address book that used XML as its data store. Overall, the application was simple (as far as applications go), but it demonstrated how to use a number of the *System.Xml* classes, including *XmlDocument*, *XmlNode*, *XmlElement*, *XmlAttribute*, and *XmlNodeList* (the four most commonly used classes). The XML Address Book

built an XML document, added nodes, edited nodes, removed nodes, and provided the user interface to tie all the code to events that the user would generate while using the application.

Solutions Fast Track

Explaining the XML Document Object Model

- ☑ The W3C created the DOM to provide a common set of interfaces for all DOM-compliant parsers to use, so developers would not need to completely relearn new parsers for different languages and technologies.
- ☑ The four main node interfaces in the DOM are *Node*, *Document*, *Element*, and *Attribute*. Using these four interfaces, you have full programmatic ability to create, modify, and delete an XML document.
- ☑ DOM Range, DOM Traversal, and DOM XPath were created to assist in working with XML documents and the DOM Core interfaces.

Introduction to the *System.Xml* Namespace

- ☑ The *System.Xml* classes fully support all DOM Level 1 Core and DOM Level 2 Core interfaces.
- ☑ Some additional functionality has been added to the *System.Xml* classes to ease development efforts, including (but not limited to) new classes to access the XML data (*XmlDataDocument*), new methods on certain classes, and new properties on certain classes.

Using the *System.Xml* Namespace

- ☑ Determine the format for the XML Address Book in order to know ahead of time how your data will be structured when you need to modify it.
- ☑ Create the ability to add and delete categories, and how to select individual categories and display their information on the screen. Create the ability to select the list of entries that are in any given category.
- ☑ Create the ability to add, edit, and delete entries, and how to select individual entries and display their information on the screen.

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** and click on the **"Ask the Author"** form.

Q: Does it matter which XML classes I use in my project?

A: Yes. You should always use the classes that are most relevant to the problem at hand. If you need to access relational data in XML format, use the *XmlDataDocument* class. If you need a simple forward-only, cursor-like access to your XML data, use the *XmlTextReader* to get your XML data.

Q: I'm having trouble figuring out when to use XML in my project; where should I use it?

A: XML should be used when the best solution is to use XML. That is vague, but, overall, XML should be used when you need a textual representation of your data. This can include any number of things, including the data store for an address book or when you need to share information between applications.

Q: Who should define my XML data format?

A: That is completely up to the developer who is working on the application. There are a number of XML standards already created for various business types. More information can be found at www.oasis-open.org/.

Q: When working with the *System.Xml* classes, should I stick to the DOM-compliant methods and properties, or should I use everything available to me?

A: It is important to understand what methods and properties are DOM compliant if you ever plan to switch to another XML parser. However, using the additional functionality provided by the *System.Xml* classes shouldn't hinder you too much if you do switch if you already understand what you're using that isn't DOM compliant.

Chapter 5

Understanding .NET and XML Security

Solutions in this chapter:

- The Risks Associated with Using XML in the .NET Framework
- .NET Internal Security as a Viable Alternative
- Security Concepts
- Code Access Security
- Role-Based Security
- Security Policies
- Cryptography
- Security Tools
- Securing XML—Best Practices

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

When you discuss a topic within the realm of Information Technology (IT), the subject of security will be brought to the table, especially if the product carries the Microsoft logo. Luckily for Microsoft, on XML, the IT industry has a collective conscience. As odd as it might seem, security was never a requirement when XML was developed. This is no longer the case, but barely. As of September 2001, a standard for XML signatures has not reached the final W3C recommendation. On XML encryption, only a draft version exists. This also means that many manufacturers have yet to implement XML signatures and encryption.

Of course, we try to keep this security void under control by sending documents over secure channels, such as SSL and VPN. However, if somebody with ill intent can make use of this secure channel, he, as is mostly the case, can submit rogue XML documents into this channel. Without verifiable sender identification, you can never be sure of the legitimacy of an XML document. Moreover, even if you can sign your XML document, you need encryption to obscure the content from prying eyes.

In a way, this chapter is kind of an odd man out, since none of the XML-related .NET namespaces touches the subject of security. However, .NET has a single nasmespace within the XML namespace that deals with security. In the documentation, Microsoft warns that you should not use this namespace, since it is meant for the .NET Framework security system. No further explanation is given, and it even contradicts the founding principles of the .NET Framework. Although many of these classes carry much resemblance to soon-to be-standard XML signature. Our guess is that it not fully complies with the latest version of the XML signature standard and therefore will not interoperate with other implementations of this standard. A possible hint is the fact that the documentation of this namespace references a draft from February 2000.

Since security is such an important subject, we will spend this chapter on XML security within the .NET Framework. Then we will look at XML and its internal security capabilities.

The Risks Associated with Using XML in the .NET Framework

XML and XSL are very powerful tools, and when wisely wielded can create Web applications that are easy to maintain because of the separation of data and

presentation. With a little planning, you can reduce the amount of code necessary by compartmentalizing key aspects of functionality using XML and XSL and reusing them throughout the application. Along with changing the way in which your components communicate within your application, XML will change the way entities communicate over the Internet.

XML and XSL are open standards, which is one of the reasons why these standards have become so popular. Many times, XML schemas are published by organizations to standardized industry- or business-related information. This is done in the hopes of further automating business processes, increasing collaboration, and easily integrating with new business partners over the Internet. As XML becomes more popular, you will begin seeing more information being exchanged between businesses and organizations. As always, secure design and architecture are key to making sure that none of that information is compromised during the exchange. The next sections provide a basis for understanding and using the XML encryption and digital signature specifications.

Confidentiality Concerns

The best way to protect data is to not expose it, and let's face it; anything you send over the Internet is fair game. Although you might feel safer making a purchase over the Internet with a credit card than when your waiter picks up your credit card at the restaurant, a risk is still a risk.

As always, when dealing with the Internet, security is an issue, but remember that XML is about data, plain and simple, and XSL is about transforming XML. Security needs to be carefully implemented in all Web applications, but it should be implemented in a layer autonomous to XML and XSL. If information is not meant to be seen, it is much safer to transform the XML document to exclude the sensitive information prior to delivering the document to the recipient, rather than encrypt the information within the document.

XSL is a great way to "censor" your XML documents prior to delivery. Because XSL can be used to transform XML into anything, including a new XML document, it will allow you to have very granular control over what data gets sent to whom when it is used in conjunction with authentication.

If you find yourself adding a username and password element to your XML, stop. If you are encrypting values prior to entering them into an XML document, stop. Tools already exist that you can use for authentication, authorization, and encryption. These concepts are integral to Web applications, but at a higher level in the overall architecture.

For example, let's say that you have an e-commerce Web site that takes orders over the Web and then sends that order to a fulfillment company via XML to be packed and shipped. Because the credit card needs to be debited at the time of shipping, you feel it necessary to send the credit card number to the fulfillment company in the XML document that contains the rest of the order information. Feeling uncomfortable in exposing that information in clear text, you decide to encrypt the credit card number within the XML document. Although your intentions are good, the decision has consequences. The XML document no longer becomes self-describing. It has also become proprietary because you need the encryption algorithm in order to extract the credit card number. This decision reintroduces some of the problems XML was meant to eliminate. In many of these cases, other solutions exist. One might be to not send the credit card information to the fulfillment company along with the rest of the order. When the order has been shipped, have the fulfillment company send a shipping notification to your application and have your application debit the credit card.

Note that both your data and your code are at risk. XSL is a complete programming language, and at times may be more valuable than the information contained within the XML it transforms. When you perform client-side transformations, you expose your XSL in much the same way that HTML is exposed to the client. Granted, most of your programming logic will remain secure on the server, but XSL still comprises a great deal of your application. Securing it is as important as securing your XML.

.NET Internal Security as a Viable Alternative

As we discuss in the following sections, code access security and role-based security are the most important vehicles to carry the security through your applications and systems. However, let it be clear that we are not discussing VB or C# security, but .NET security; that is, the security defined by the .NET Framework and enforced by the CLR. Since the .NET Framework namespaces make full use of the security, every call to a protected resource or operation when using one of these namespaces automatically activates the code access security (CAS). Only if you start up the CLR with the security switched off, CAS will not be activated. The CLR is able to "sandbox" code that is executed, preventing code that is not trusted from accessing protected resources or even from executing at all. We

discuss this more thoroughly in the *Code Access Security* section later in this chapter. What is important to understand is that you can no longer ignore security as a part of your design and implementation phase. It is a priority to safeguard your systems from malicious code, and you also want to protect your code/application from being "misused" by less-trusted code. For example, let's say that you implement an assembly that holds procedures/functions that modifies Registry settings. Because these procedures/functions can be called by other unknown code, these can become tools for malicious code if you do not incorporate the .NET Framework security as part of your code.

To be able to use the .NET Security to your advantage, you need to understand the concepts behind the security.

Permissions

In the real world, permission refers to an authority giving you, or anyone else for that matter, the formal "OK" to perform a specified task that is normally restricted to a limited group of persons. The same goes for the meaning of permission in the .NET Security Framework: getting permission to access a protected resource or operation that is not available for unauthorized users and code. An example of a protected resource is the Registry, and a protected operation is a call to a COM+ component, which is regarded as unmanaged code and therefore less secure. The types of permissions that can be identified include:

- **Code access permissions** Protects the system from code that can be malicious or just unstable; see the *Code Access Security* section for more information.
- **Role-based security permissions** Limits the tasks a user can perform, based on the role(s) he plays or the identity he has; see the *Role-Based Security* section for more information.
- **Identity permissions** See the *Role-Based Security* section for more information.
- **Custom permissions** You can create your own permission in any of the other three types, or any combination thereof. This demands a thorough understanding of the .NET Framework security and the working of permissions. An ill-constructed permission can create security vulnerabilities.

You can use permissions through different methods:

- **Requests** Code can request specific permissions from the CLR, which will only authorize this request if the assembly in which the code resides has the proper trust level. This level is related to the security policy that is assigned to the assembly, which is determined on the base of evidence the assembly carries. Code can never request more permission than the security policy defines; such a request will always be denied by the CLR. However, the code can request less permission. What exactly security policy and evidence consist of is discussed over the course of this chapter.
- **Grants** The CLR can grant permissions based on the security policy and the trustworthiness of the code, and it requests code issues.
- **Demands** The code demands that the caller has already been granted certain permissions in order to execute the code. This is the security part for which you are actively responsible.

Principal

The term *principal* refers directly to the role-based security, being the security context of the executed code. A principal is created based on the identity and role(s) of the caller, whether it is a user or other code. In fact, every thread that is activated is assigned a principal that is by default equal to the principal of the caller. Although we just stated that the principal holds the identity of the caller, this is not entirely correct, because the principal has only a reference to the caller's identity, which already exists prior to the creation of the principal. Three types of principals can be identified:

- **Windows principal** Identifies a user and the groups it is a member of that exists within a Windows NT/2000 environment. A Windows principal has the capability to impersonate another Windows user, which resembles the impersonate you might know from the COM+ applications.
- **Generic principal** Identifies a user and its roles, not related to a Windows user. The application is responsible for creating this type of principal. Impersonation is not a characteristic of a general principal, but because the code can modify the principal, it can take on the identity of a different user or role.
- **Custom principal** You can construct these yourself to create a principal with additional characteristics that better suits your application. Custom principals should never be exposed, because doing so can create serious security vulnerabilities.

Authentication

In general, *authentication* is the verification of a user's identity; hence, the credentials he hands over. Because the identity of the caller in the .NET Framework is presented through the principal, the identity of the principal has to be established. Because your code can access the information that is available in the principal, it can perform additional authentication tests. In fact, because you can define your own principal, you can also be in control over the authentication process. The .NET Framework supports not only the two most-used authentication methods within the Windows 2000 domain—NTLM and Kerberos V5.0—but also supports other forms of authentication, such as Microsoft Passport. Authentication is used in role-based security to determine if the user has a role that can access the code.

Authorization

Authorization takes place after authentication, based on the established identity of the principal. Authorization in relation to roles has to be part of the code and can take place at every point in the code. You can use the user and role information in the principal to determine if a part of the code can be executed. The permissions the principal is given, based on its identity, determine if the code can access specific protected resources.

Security Policy

To be able to manage the security that is enforced by the CLR, an administrator can create new or modify existing security policies. Before an assembly is loaded, its credentials are checked. This evidence is part of the assembly. The assembly is assigned a security policy depending on the level of trust, which determines the permissions the assembly is granted. The setting of security policies is controlled by the system administrator and is crucial in fending off malicious code. The best approach in setting the security policies is to grant no permissions to an assembly for which the identity cannot be established. The stricter you define the security policies, the more securely your CLR will operate.

Type Safety

A piece of code is labeled *type safe* if it only accesses memory resources that do not belong to the memory assigned to it. Type safety verification takes place during the JIT compilation phase and prevents unsafe code from becoming

active. Although you can disable type safety verification, it can lead to unpredictable results. The best example is that code can make unrestricted calls to unmanaged code, and if that code has malicious intent, the results can be severe. Therefore, only fully trusted assemblies are allowed to bypass verification. Type safety can be regarded as a form of "sandboxing."

Code Access Security

The .NET Framework is based on the concept of distributed applications, in which an application does not necessarily have a single owner. To circumvent the problem of which parts of the application (being assemblies) to trust, code access security is introduced. This is a very powerful way to protect the system from code that can be malicious or just unstable. Remember that it is always active, even if you do not use it in your own code. CAS helps you in:

- Limiting access permissions of assemblies by applying security policies
- Protecting the code from obtaining more permissions than the security policy initially permits
- Managing and configuring permission sets within security policies to reflect the specific security needs
- Granting assemblies specific permissions that they request
- Enabling assemblies in demanding specific permissions from the caller
- Using the caller's identity and credentials to access protected resources and code

.NET Code Access Security Model

The .NET code access security model is built around a number of characteristics:

- Stack walking
- Code identity
- Code groups
- Declarative and imperative security
- Requesting permissions
- Demanding permissions

- Overriding security checks
- Custom permissions

By discussing these characteristics, you will get a better understanding of how CAS works, and how it can work for you during the design and implementation of applications.

Stack Walking

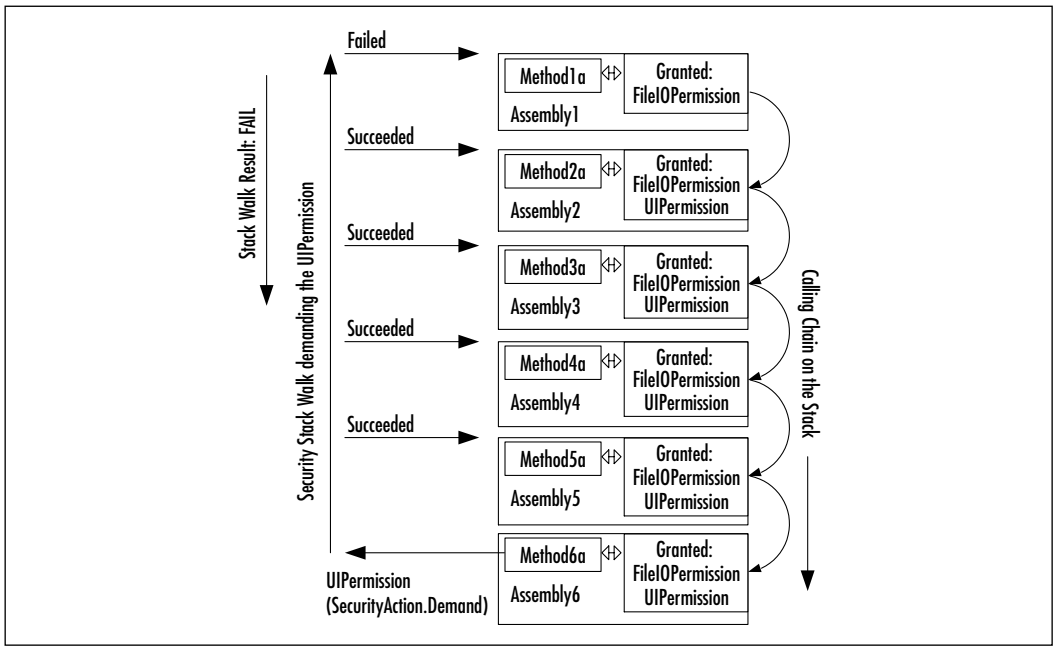
Perhaps *stack walking* is the most important mechanism within CAS to ensure that assemblies cannot gain access to protected resources and code during the course of the execution. As mentioned before, one of the initial steps in the assembly load process is that the level of trust of the assembly is determined, and corresponding permission sets are associated with the assembly. The total package of sets is the maximum number of permissions an assembly can obtain.

Because the code in an assembly can call a method in another assembly and so forth, a *calling chain* develops (Figure 5.1), with every assembly having its own permissions set. Suppose that an assembly demands that its caller have a specific permission (*UIPermission* in Figure 5.1) to be able to execute the method. Now the stack walking of the CLR kicks in. The CLR starts checking the stack where every assembly in the calling chain has its own data segment. Going back in the stack, every assembly is checked for the presence of this demanded permission, in our case *UIPermission*. If all assemblies have this permission, the code can be executed. If, however, somewhere in the stack an assembly does not have this permission (in our case this is in the top assembly *Assembly1*), the CLR throws an exception, and access to the method is refused.

Stack walking prevents calling code from getting access to protected resources and code for which it initially does not have authorization. You can conclude that at any point of the calling chain the effective permission set is equal to the intersection of the permission sets of the assemblies involved.

Even if you do not incorporate the permission demand in your code, stack walking will take place because all class libraries that come with the CLR make use of demand to ensure the secure working of the CLR. The only drawback of stack walking is that it can have a serious performance impact, especially if the calling chain is long. Suppose the stack contains eight assemblies, and the top assembly makes a call to a method that demands a specific permission and does so in a 200-fold loop. After executing the loop, 200 security stack walks are triggered. Since each stack walk performs eight security checks, the total number of security checks is 1600.

Figure 5.1 Performing Stack Walking to Prevent Unauthorized Access



Code Identity

The whole principle of the .NET Framework security rides on *code identity,* or to what level a piece of code can be trusted. The code identity is established based on the evidence that is presented to the CLR. Evidence can come from two sources:

- Evidence that is incorporated in the assembly, and put in there during the coding and subsequent compiling of the code, or which can later be added to the assembly.
- Evidence that is provided by the host where the assembly resides. The CLR controls the accepting of host evidence, through the security permission *ControlEvidence*, which should be granted only to trusted hosts.

Table 5.1 lists the default evidence that can be used to determine to what code group code belongs. Because you cannot control the identity of the assembly, you are never sure how reliable this evidence is, except for the signatures provided.

Table 5.1 The Available Default Types of Evidence

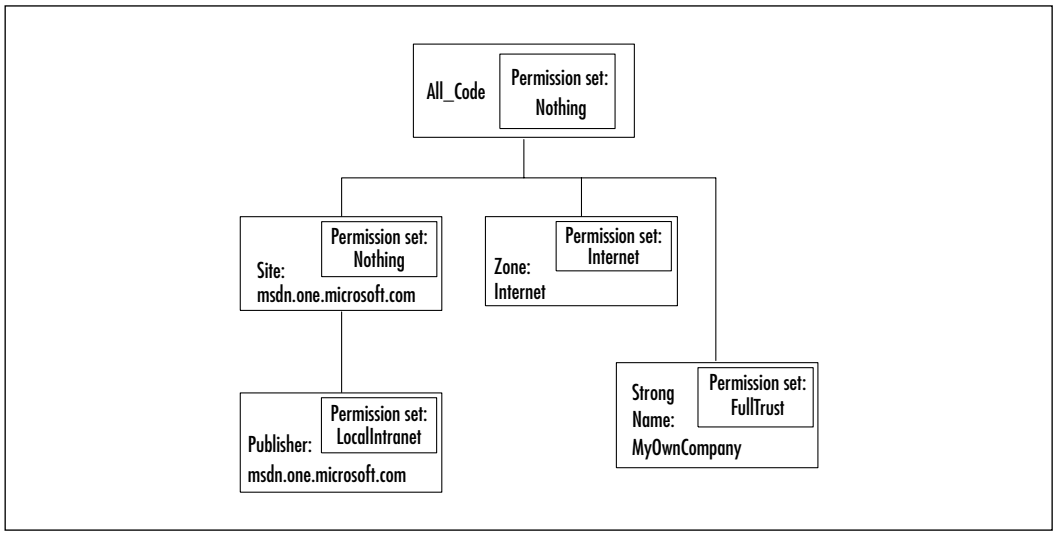
Evidence	Description
Directory	The directory where the application—hence, assembly—is installed.
Hash	The cryptographic hash that is used in the code of the code: MD5 or SHA1 (see the *Cryptography* section).
Publisher	The signature of the assembly's owner, in the form of a X.509 certificate, set through *Authenticode*.
Site	The name of the site from which the assembly originates; for example, www.company.com (prefixes and suffixes are disregarded).
Strong name	The strong name consists of the assembly name (given name), public key (of the publisher), version numbers, and culture.
URL	The full URL, also called code base, including prefix and suffix: https://www.company.com:4330/*.
Zone	The zone in which the assembly originates. Default zones are Internet, Local Intranet, My Computer, No Zone Evidence, Trusted Sites, and Untrusted (Restricted) Sites.

The more evidence you can gather about the assembly, the better you can determine to what extent you can grant it permissions. The strong name is of great importance. If you and all other serious application developers are persistent in providing assemblies with strong names, you can prevent your code from becoming the vehicle of someone's dubious intents. Sadly enough, malicious code can still have a convincing strong name, which is why the best evidence is the certificate and signature that should be present with the assembly. Once you have established the trustworthiness of an assembly, based on all the evidence before you, you can determine the appropriate permission sets. Here is where your realm of control starts, by constructing appropriate code groups.

Code Groups

A *code group* can be defined as a group of assemblies that share the same value for one, and only one, piece of evidence, called *membership condition*. Based on this evidence, a permission set is attached to the assembly. Because a code group is part of a code group hierarchy (Figure 5.2), an assembly can be part of more code groups. The effective permission set of the assembly is the union of the permissions sets of the code groups to which it belongs.

Figure 5.2 Graphical Representation of a Code Group Hierarchy



When an assembly is about to be loaded, the evidence is collected and the code group hierarchy is checked. When the assembly is matched with a code group, the CLR will check its child code groups. This implies that the construction of the hierarchy is very important and must be built starting with the general evidence items—for example, starting with zone and moving on to more specific ones such as publisher. A complicating factor is that there are three security levels (*Enterprise*, *Machine*, and *User*), each with its own code group hierarchy. All three are evaluated, resulting in three permission sets, which at the end are intersected, thereby determining the effective permission set.

It is the administrator's responsibility to construct code group hierarchies that can quickly be scanned and enforce a high level of security. To do so, you must take several factors into account:

- Limit the number of levels.
- Use membership conditions at the first level that are highly discriminatory, preventing large parts of the hierarchy from being checked.
- The hierarchy's root, *All Code*, should have no permissions assigned, so code that does not contain at least some evidence is not allowed to run.
- The more convincing the evidence—for example, the publishers certificate—the more permissions that can be granted.

- Make no exceptions or shortcuts by giving out more permissions than the evidence justifies. Assume that you have a specific application running in the intranet zone that needs to have full trust to operate. Because it is your own application, you implicitly trust it, without the factual evidence. If you do this, however, it can come back to haunt you.

Table 5.2 lists the available default membership conditions. You can construct your own, but that is beyond the scope of this chapter. Membership conditions are discussed in more detail later in the chapter.

Table 5.2 Default Membership Conditions for Code Groups

Membership Condition	Description
All Code	Applies to every assembly that is loaded.
Application directory	Applies to all assemblies that reside in the same directory tree as the running application; hence, the Application domain.
Hash	Applies to all the assemblies that use the same hash algorithm as specified, or have the specified hash value.
Publisher	Applies to all assemblies that carry the specified publishers certificate.
Site	Applies to all assemblies that originate from the same site.
Skip verification	Applies to all assemblies that request the Skip Verification permission. WARNING: This permission allows for the bypassing of type safety. Use it only at the lowest level after you have established that the code is fully trusted.
Strong name	Applies to all assemblies that have the specified strong name.
URL	Applies to all assemblies that originate from the specified URL, including prefix, suffix, path, and eventual wildcard.
Zone	Applies to all assemblies that reside in the specified zone.
(custom)	Applies to custom-made conditions that are normally directly related to specific applications.

Declarative and Imperative Security

There are two ways to add security to your code. This can be a demand that callers have a specific permission, or a request for a specific permission from the CLR.

The first method is *declarative security*, which can be set at assembly, class, and/or member level, so you can demand different permissions at different places in the assembly. Permission demand at member level will only be effectuated, as this part of the code is actually called. The VB.NET syntax of declarative code is *<[assembly:]Permission(SecurityAction.Member, State)>*; for example:

```
<assembly: FileIOPermission(SecurityAction.Demand, Unrestricted := True)>
<FileIOPermission(SecurityAction.Request, Unrestricted := True)>
```

The first security example is valid for the entire assembly; hence, every call in this assembly needs to have the FileIOPermission. The second example can be used for a class or a single method. Only a reference to a class or a call of the method will request the CLR for FileIOPermission.

As the syntax already suggests, by using *brackets (<>)*, this code is not treated as ordinary code. In fact, as you compile the code to an assembly, these lines are extracted and placed in the metadata part of the assembly. This metadata is checked at different points, such as during the load of the assembly or when a method in the assembly is called. Using declarative security, you can demand, request, or even override permissions before the code is even executed. This gives you a powerful security tool during the development of the code and assemblies. However, this means that you must be aware of the type of permissions you need to request and/or demand your code.

The second method is *imperative security,* which becomes a part of your code and can make permission demands and overrides. It is not possible to request permissions using imperative security, because that makes it unclear at what point a specific permission is needed and at what point it is no longer needed. That is why permission requests are related to identifiable units of code. You might want to use imperative security to check if the caller has a permission that is specific for a part of the code. For example, just before a conditional part of the code (this might even be triggered by the role-based security) wants to access a file or a Registry key, you want to check if the caller has this *FileIOPermission* or *RegistryPermission*. The VB.NET syntax of the imperative security in code looks like this:

```
Dim PermissionObject as New Permission()
PermissionObject.Demand()
```

Here is an example:

```
Dim CheckPermission as New FileIOPermission()
CheckPermission.Demand()
```

The permission object is valid only for the scope on which it is declared, and it will be automatically discarded at the time the code returns to a higher scope. During this scope, imperative security demands and overrides overrule the permissions demanded with a declarative security statement.

Having discussed declarative and imperative security, it is time to take a look at how you can use this to request, demand, and override permissions.

Requesting Permissions

Requesting permissions is the best way to create a secure application and prevent possible misuse of your code by malicious code. As mentioned before, based on the evidence, an assembly hands over to the CLR, and then a permission set is determined, using security policies. These security policies are constructed independently from the permissions an assembly needs. Of course, if you fully trust an assembly, you can grant it all the permissions it needs. An assembly can be granted more permissions than it actually needs. Requesting permissions is not asking for more permissions than you are granted, based on the security profile, but refraining from granting permissions the code does not need. By now you have probably started to wonder what the use of requesting permissions is if the security policy decides what permissions are available to the assembly. The term *available* implies two issues:

- If an assembly requests more permissions than it is granted, based on the security policy, it will not be loaded and/or the code will not be executed. Instead, the CLR will throw an exception.
- If an assembly requests less permissions, it protects itself from misuse of these additional permissions somewhere up or down the calling chain.

Requesting permissions is a characteristic of proper .NET applications, and demands from the developer a good understanding of the use of permissions related to the code he writes. Because you can only request permissions by using declarative security, you can first write and test the code and then add the permission requests later. This can make the development process easier, saving you the hassle of constantly having to consider permission requests for unfinished code.

There are three types of permission requests:

- **RequestMinimum** Defines the permissions the code absolutely needs to be able to run. If the *RequestMinimum* permission is not part of the granted permission set, the code is not allowed to run.

- **RequestOptional** Defines the permissions the code might not necessarily need to be able to run, but might need in certain circumstances. If the *RequestOptional* permission is not part of the granted permission set, the code is still allowed to run. However, you need the code to be able to handle the situation in which the permission is needed but not granted, thus handling exceptions.
- **RequestRefuse** Defines the permissions the code will never need and which should not be granted to the assembly. By refraining from certain permissions you prevent malicious code or unstable code from misusing these permissions.

After the code is completed and you compile assemblies, you should get into the practice of making a minimum, optional, or refuse request for *every* permission (as listed in Table 5.3), based on the permissions needed by the code. Eventually, you can make it more specific to relate it to classes or members. Besides the fact that you can create secure assemblies, it is also a good way of documenting the permissions related to your code.

Table 5.3 The Default Permission Classes Derived from the *CodeAccessPermission* Class

Permission Class	Permission Type	Description
DirectoryServices Permission	Resource	Controls access to the *System.DirectoryServices* classes.
DnsPermission	Resource	Controls access to the DNS servers on the network.
Environment Permission	Resource	Controls access to the user environment variables.
EventLogPermission	Resource	Controls access to the event log services.
FileDialogPermission	Resource	Controls access to files that are selected through an Open | File… dialog.
FileIOPermission	Resource	Controls access to files and directories.
IsolatedStorageFile Permission	Resource	Controls access to a private virtual file system related to the identity of the application or component.
MessageQueue Permission	Resource	Controls access to the MSMQ services.

Continued

Table 5.3 Continued

Permission Class	Permission Type	Description
OleDbPermission	Resource	Controls access to the OLE DB data provider and the data sources associated with it.
PerformanceCounter Permission	Resource	Controls access to the performance counters of Windows 2000 (or NT).
PrintingPermission	Resource	Controls access to printers.
ReflectionPermission	Resource	Controls access to metadata types.
RegistryPermission	Resource	Controls access to the Registry.
SecurityPermission	Resource	Controls access to *SecurityPermission*, such as Assert, Skip Verification, and Call Unmanaged Code.
ServiceController Permission	Resource	Controls access to services on the system.
SocketPermission	Resource	Controls access to sockets that are needed to set up or accept a network connection.
SqlClientPermission	Resource	Controls access to SQL server databases.
UIPermission	Resource	Controls access to UI functionality, such as Clipboard.
WebPermission	Resource	Controls access to an Internet-related resource.
PublisherIdentity Permission	Identity	Permission is granted if the evidence publisher is provided by the caller.
SiteIdentityPermission	Identity	Permission is granted if the evidence site is provided by the caller.
StrongNameIdentity Permission	Identity	Permission is granted if the evidence strong name is provided by the caller.
UrlIdentityPermission	Identity	Permission is granted if the evidence URL is provided by the caller.
ZoneIdentity Permission	Identity	Permission is granted if the evidence zone is provided by the caller.

Now let's look at some examples of the different types of requests:

```
<assembly: SecurityPermissionAttribute(SecurityAction.RequestMinimum, _
      Flags := SecurityPermissionFlag.ControlPrincipal)>
```

In order for this assembly to run, it needs at least the permission to be able to manipulate the principal object. This is a permission you would give only to an assembly you trust.

```
<assembly: SecurityPermissionAttribute(SecurityAction.RequestMinimum, _
      ControleEvidence : = True)>
```

In order for this assembly to run, it needs at least the permission to be able to provide additional evidence and modify the evidence as provided by the CLR. This is a powerful permission you would give only to fully trusted assemblies.

```
<FileIOPermissionAttribute(SecurityAction.RequestOptional, _
      Write := "C:\Test\*.cfg")> Public Class ClassAct
```

The *ClassAct* class requests the optional permission to be able to write to files in the C:\Test directory with the extension .cfg. If the security policy permits *FileIOPermission*, this restricted request is given. If the *FileIOPermission* is not granted, then any subsequent write to a CFG file in C:\Test will fail.

```
<assembly: FileIOPermission(SecurityAction.RequestRefuse, _
Unrestricted := True)>
```

The assembly refuses the *FileIOPermission*, even if the security policy grants this permission. If you used this request in combination with the previous example, and the security policy grants *FileIOPermission*, only *ClassAct* will get this restricted *FileIOPermission*, and the rest of the code in the assembly will not have any *FileIOPermission*.

```
<assembly: FileIOPermission(SecurityAction.RequestRefuse, _
      All := "C:\Winnt\System32\*.*")>
```

The assembly refuses only *FileIOPermission* to the access of files in the C:\Winnt\System32 directory. If the security policy grants this permission, the assembly can access all files, except for the one in the stated directory.

Instead of making requests for every code access permission, you can also request one of the following named permission sets: *Nothing*, *Execution*, *Internet*, *LocalIntranet*, *SkipVerification*, and *FullTrust*. You can do this by issuing the following request:

```
<assembly: PermissionSetAttribute(SecurityAction.RequestMinimum, _
       Name := NamedPermissionSet)>
```

Another way to request more code access permissions in one statement is to use XML-coded permission sets:

```
<assembly: PermissionSetAttribute(SecurityAction.RequestMinimum, _
     File := "Filename.xml")>
```

Demanding Permissions

By demanding permissions, you force the caller to have a specific permission it needs to execute the code. If the caller has this request, it is very likely that he obtained it by requesting it at the CLR. As we discussed before, a permission demand triggers a security stack walk. Even if you do not perform these demands yourself, the .NET Framework classes will. This means that you should never perform permission demands related to these classes, because they will take care of those themselves. If you do perform a demand, it will be redundant and only add to the execution overhead. This does not mean that you should ignore it; instead, when writing code, you must be aware of which call will trigger a stack walk, and make sure that the code does not encourage a surplus of stack walks. However, when you build your own classes that access protected resources, you need to place the proper permission demands, using the declarative or imperative security syntax.

Using the declarative syntax when making a permission demand is preferable to using the imperative syntax, because the latter might result in more stack walks. There are, of course, cases that are better suited for imperative permission demands. For example, if a Registry key has to be set under specific conditions, you will perform an imperative *RegistryPermission* demand just before the code is actually called. This also implies that the caller can lack this permission, which will result in an exception that the code needs to handle accordingly. Another reason why you want to use imperative demands is when information is not known at compile time. A simple example is *FileIOPermission* on a set of files whose names are only known during runtime because they are user related.

Two types of demands are handled differently than previously described. First, the *link demand* can be used only in a declarative way at the class or method level. The link demand is performed only during the JIT compilation phase, in which it is checked if the calling code has sufficient permission to link to your code. A security stack walk is not performed because linking

exists only in a direct relation between the caller and code being called. The use of link demands can be helpful to methods that are accessible through reflection. The link demand will not only perform a security check on code that obtains the *MethodInfo* object—hence, performing the reflection—but the same security check is performed on the code that will make the actual call to the method. The following two examples show a link demand at class and at method level:

```
<SecurityPermissionAttribute(SecurityAction.LinkDemand, _
                                Unrestricted := True)> _
                                Public Class ClassAct


Public Shared Function _
     <SecurityPermissiobAttribute(SecurityAction.LinkDemand)> _
     Act1() As Integer
          ' body of the function
End Function
```

The second type of demand is *inheritance demand*, which can be used at both the class and method level, through the declarative security. Placing an inheritance demand on a class can protect that class from being inherited by a class that does not have the specified permission. Although you can use a default permission, it makes sense to create a custom permission that must be assigned to the inheriting class to be able to inherit from the class with the inheritance demand. The same goes for the class that inherits from the inheriting class. For example, let's say that you have created the *ClassAct* class that is inheritable, but also has an inheritance demand set. You have defined your own inherit permission *InheritAct*. Another class called *ClassActing* wants to inherit from your class, but because it is protected with an inheritance demand, it must have the *InheritAct* permission in order to be able to inherit. Let's assume that this is the case. Now there is another class called *ClassReacting* that wants to inherit from the class *ClassActing*. In order for *ClassReacting* to inherit from *ClassActing*, it also needs to have the *InheritAct* permission assigned. The inheritance demand would look like this:

```
<InheritActAttribute(SecurityAction.InheritanceDemand)> Public Class
ClassAct
```

The inheritance demand at method level can be the following:

```
Public Overridable Function
<SecurityPermissionAttribute(SecurityAction.InheritanceDemand)>
     Act1() as Integer
       ' Body of the function
End Function
```

Overriding Security Checks

Because stack walking can introduce serious overhead and thus performance degradation, you need to keep stack walks under control. This is especially true if they do not necessarily contribute to security, such as when a part of the execution can only take place in fully trusted code. On the other hand, your code has permission to access specific protected resources, but you do not want code that you call to gain access to these resources—so you want to have a way to prevent this. In both cases, you want to take control of the permission security checks; hence, overriding security checks. You can do this by using the security actions *Assert*, *Deny*, and *PermitOnly* (meaning "deny everything but").

After the code sets an override, it can undo this override by calling the corresponding *Revert* method: *RevertAssert*, *RevertDeny*, and *RevertPermitOnly*, respectively. Get into the practice of first calling the *Revert* method before setting the override, because performing a revert on a nonexisting override has no effect.

WARNING

You can place more than one override of the same type—for example, *Deny*—within the same piece of code. However, this is not acceptable to the CLR. If during a stack walk, the CLR encounters more than one of the same asserts it throws an exception, because it does not know which of the overrides to trust. If you have more than one place in a piece of code where you set an override, be sure to revert the first one before setting the new one.

Assert Override

When you set an assert override on a specific permission, you force a stack walk on this permission to stop at your code and not continue to check the callers of your method.

WARNING

If you use an assert, you inadvertently create a security vulnerability, because you prevent the CLR from completing security checks. You must convince yourself that this vulnerability cannot be exploited.

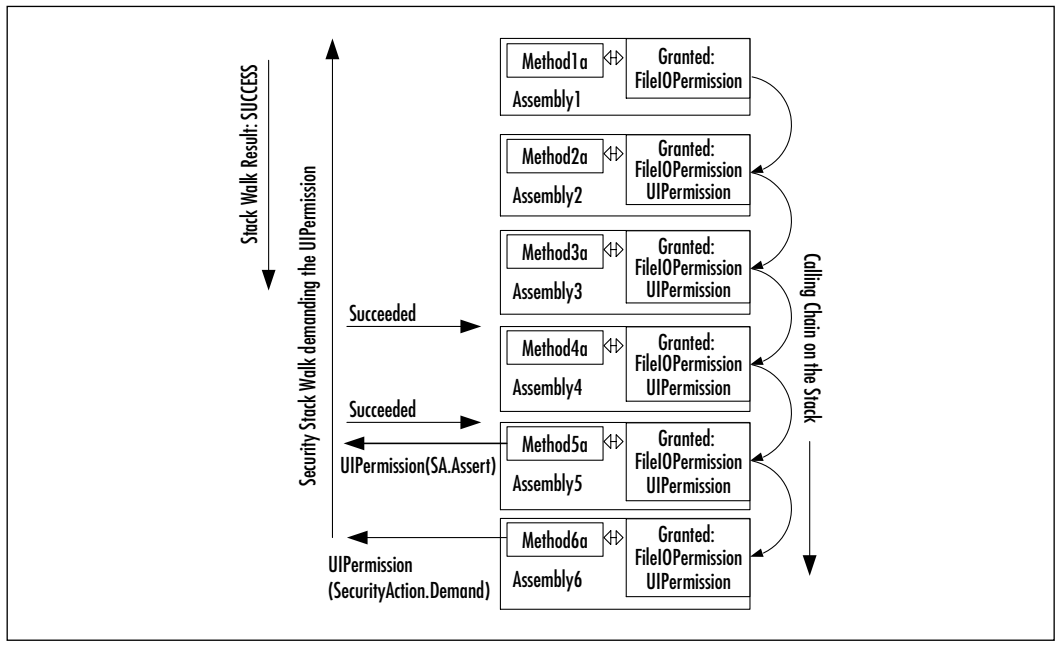
The use of *Assert* makes sense in the following situations:

- You have coded a part of an application that will never be exposed to the outside world. The user of the application has no way of knowing what happens within that part of the application. Your code does need access to protected resources, such as system files and/or Registry keys, but because the callers will never find out that you use these protected resources, it is reasonably safe to set an *Assert* to prevent a full security check from being performed. You do not care if the caller has that permission or not.
- Your code needs to make one or more calls to unmanaged code, but because the caller of the code obtains access through your Web site, you are safe in assuming that he will not have permissions to make calls to unmanaged code. On the other hand, the callers cannot influence the calls you make to unmanaged code. Therefore, it is reasonably safe to assert the permission to access unmanaged code.
- You know that somewhere in your code you have to perform a search, using a *Do..Loop* structure that at one point has to access a protected resource. You also know that the code that calls the protected resource cannot be called from outside the loop. Therefore, you decide to set an assertion just before the call to the protected resource, to prevent a surplus of stack walks. In case the particular piece of code that does the call to the protected resource can be called by other code, you have to move up the assertion to the code that can only be called from the loop.

Let's look at the stack walk that was initially used in Figure 5.1, but now let's throw in an assertion and see what happens (Figure 5.3). The assert is set in *Assembly4* on the *UIPermission*. In the situation with no assert, the stack walk did not succeed because *Assembly1* did not have this permission. Now the stack walk starts at *Assembly6* performing a permission demand on *UIPermission*, and goes on its way as it usually goes. Now the stack walk reaches *Assembly4* and recognizes

an assert on the permission it is checking. The stack walk stops there and returns with a positive result. Because the stack walk was short-circuited, the CLR has no way of knowing that *Assembly1* did not have this permission.

Figure 5.3 A Stack Walk Is Short-Circuited by an *Assert*



An *Assert* can be set using both the declarative and the imperative syntax. In the first example, the declarative syntax is used. An *Assert* is set on the *FileIOPermission.Write* permission for the CFG files in the C:\Test directory:

```
Public Function _
     <FileIOPermission(SecurityAction.Assert, Write :=
"C:\Test\*.cfg")> _
       Act1() As Integer
       ' body of the function
End Function
```

The second example uses the imperative syntax setting the same type of *Assert*:

```
Public Function Act1() As Integer
     Dim ActFilePerm As New
FileIOPermission(FileIOPermissionAccess.Write, _
                                                "C:\Test\*.cfg")
```

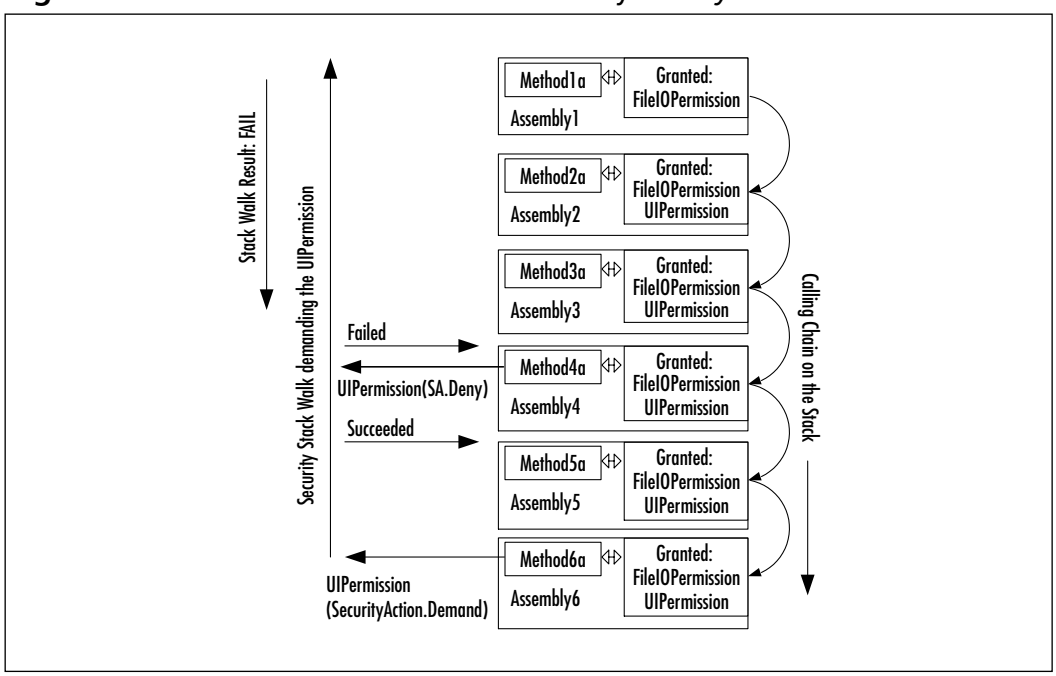
```
            ActFilePerm.Assert
             ' rest of body
End Function
```

Deny Override

The *Deny* does the opposite of *Assert* in that it lets a stack walk fail for the permission the *Deny* is set on. There are not many situations in which a *Deny* override makes sense, but here is one: Among the permissions your code has is *RegistryPermission*. Now it has to make a call to a method for which you have no information regarding trust. To prevent that code from taking advantage of the *RegistryPermission*, your code can set a *Deny*. Now you are sure that your code does not hand over a high-trust permission.

Because unnecessary *Deny* overrides can disrupt the normal working of security checks (because they will always fail on a *Deny*), you should revert the *Deny* after the call ends for which you set the *Deny*.

Figure 5.4 A Stack Walk Is Short-Circuited by a *Deny*



For the sake of the example, we use the same situation as in Figure 5.3, but instead of an *Assert*, there is a *Deny* (Figure 5.4). Again, the security stack walk is

triggered for the *UIPermission* permission in *Assembly6*. When the stack walk reaches *Assembly4*, it recognizes the *Deny* on *UIPermission* and it ends with a fail. In our example, the security check would ultimately have failed in *Assembly1*, but if *Assembly1* had been granted the *UIPermission*, the stack walk would have succeeded, if not for the *Deny*. Effectively this means that *Assembly4* revoked the *UIPermission* for *Assembly5* and *Assembly6*.

You can set a *Deny* by using both the declarative and the imperative syntax. In the first example, the declarative syntax is used. A *Deny* is set on the *FileIOPermission* permission for all the files in the C:\Winnt\System32 directory:

```
Public Function _
        <FileIOPermission(SecurityAction.Deny, All :=
"C:\Winnt\System32\*.*")> _
        Act1() As Integer
               ' body of the function
End Function
```

The second example uses the imperative syntax setting the same type of *Assert*:

```
Public Function Act1() As Integer
     Dim ActFilePerm As New
FileIOPermission(FileIOPermissionAccess.AllAccess, _
                                                "C:\Winnt\System32\*.*")
          ActFilePerm.Deny
          ' rest of the body
End Function
```

PermitOnly Override

The *PermitOnly* override is similar to the negation of the *Deny*, by denying every permission but the one specified. You use the *PermitOnly* for the same reason you use *Deny*, only this one is more rigorous. For example, if you permit only the *UIPermission* permission, every security stack walk will fail but the one that checks on the *UIPermission*. Take Figure 5.4 and substitute *Deny* with *PermitOnly*. If in *Assembly6* the security check for *UIPermission* is triggered, the stack walk will pass *Assembly4* with success, but will ultimately fail in *Assembly1*. If any other security check is initiated, it will fail in *Assembly*. The result is that *Assembly5* and *Assembly6* are denied any access to a protected resource that incorporates a *Demand* request, because every security check will fail. As you

can see, *PermitOnly* is a very effective way of killing any aspirations of called code in accessing protected resources. The *PermitOnly* is used in the same way as *Deny* and *Assert*.

Custom Permissions

The .NET Framework enables you to write your own code access permissions, even though the framework comes with a large number of code access permission classes. Because these classes are meant to protect the protected resources and code that are exposed by the framework, it might well be the case that the application you are developing has defined resources that are not protected by the framework permissions, or you want to use permissions that are more tuned toward the needs of your application.

You are completely free to replace existing framework permission classes, although this requires a large amount of expertise and experience. In case you are just adding new permission classes to the existing ones, you should be particularly careful not to overlap permissions. If more than one permission protects the same resource or operation, an administrator has to take this into account if he has to modify the rights to these resources.

NOTE

The subject of overlapping permissions brings up a topic not discussed earlier. Although the whole discussion of code access permission has been from the standpoint of the CLR, or .NET Framework, eventually the CLR has to access resources on behalf of the users/application. Even if the code has been granted a specific permission to access a protected resource, that does not automatically mean that it is allowed to access that system resource. Take the example of a method having the *FileIOPermission* permission to the directory C:\Winnt\System32. If the identity of the Windows principal has not been given access to this part of the file system, accessing a file in that directory will fail anyway. This implies that the administrator not only has to set up the permissions within the security policy, but he also has to configure the Windows 2000 platform to reflect these access permissions.

Building your own permissions does not only imply that certain development issues are raised, but even more so, the integrity of the entire security system must be

discussed. You have to take into account that you are adding to a rigid security system that relies heavily on trust and permissions. If mistakes occur in the design and/or implementation of a permission, you run the risk of creating security holes that can become the target of attacks or let an application grant access to protected resources that it is not authorized to access. Discussing the process of designing your own permissions goes beyond the scope of this chapter. However, the following steps give you an understanding of what is involved in creating a custom permission:

1. Design a *permission class*.
2. Implement the interfaces *IPermission* and *IUnrestrictedPermission*.
3. In case special data types have to be supported, you must implement the interface *ISerializable*.
4. You must implement XML *encoding* and *decoding*.
5. You must implement the support for *declarative security*.
6. Add *Demand* calls for the custom permission in your code.
7. Update the *security policy* so that the custom permission can be added to permission sets.

Role-Based Security

Role-based security is not new to the .NET Framework. If you already have experience with developing COM+ components, you surely have come across role-based security. The concept of role-based security for COM+ applications is the same as for the .NET Framework. The difference lies in the way in which it is implemented. If we talk about role-based security, the same example comes up, over and over again. This is not because we can't create our own example, but because it explains role-based security in a way everyone understands. So here it is: You build a financial application that can handle deposit transactions. The rule in most banks is that the teller is authorized to make transactions up to a certain amount, let's say $5,000. If the transaction goes beyond that amount, the teller's manager has to step in to perform the transaction. However, because the manager is only authorized to do transactions up to $10,000, the branch manager has to be called to process a deposit transaction that is over this amount.

Therefore, as you can see, role-based security has to do with limiting the tasks a user can perform, based on the role(s) he plays or the identity he has. Within the .NET Framework, this all comes down to the principal that holds the identity and

role(s) of the caller. As discussed earlier in this chapter, every thread is provided with a principal object. In order to have the .NET Framework handle the role-based security in the same manner as it does code access security, the permission class *PrincipalPermission* is defined. To avoid any confusion, *PrincipalPermission* is *not* a derived class of *CodeAccessPermission*. In fact, *PrincipalPermission* holds only three attributes: *User*, *Role*, and the Boolean *IsAuthenticated*.

Principals

Let's get back to where it all starts: the principal. From the moment an application domain is initialized, a default call context is created to which the principal will be bound. If a new thread it activated, the call context and the principal are copied from the parent thread to the new thread. Together with the principal object, the identity object is also copied. If the CLR cannot determine what the principal of a thread is, a default principal and identity object is created so that the thread can run at least with a security context with minimum rights. There are three type of principals: *WindowsPrincipal*, *GenericPrincipal*, and *CustomPrincipal*. The latter goes beyond the scope of this chapter and is not discussed any further.

WindowsPrincipal

Because the *WindowsPrincipal* that references the *WindowsIdentity* is directly related to a Windows user, this type of identity can be regarded as very strong because an independent source authenticated this user.

To be able to perform role-based validations, you have to create a *WindowsPrincipal* object. In the case of the *WindowsPrincipal*, this is reasonably straightforward, and there are actually two ways of implementing it. This depends on whether you have to perform just a single validation of the user and role(s), or you have to do this repeatedly. Let's start with the single validation solution:

1. Initialize an instance of the *WindowsIdentity* object using this code:

   ```
   Dim WinIdent as WindowsIdentity = WindowsIdentity.GetCurrent()
   ```

2. Create an instance of the *WindowsPrincipal* object and bind the *WindowsIdentity* to it:

   ```
   Dim WinPrinc as New WindowsPrincipal(WindIdent)
   ```

3. Now you can access the attributes of the *WindowsIdentity* and *WindowsPrincipal* object:

```
Dim PrincName As String = WinPrinc.Identity.Name
Dim IdentName As String = WinIdent.Name 'this is the same as
    the previous line
Dim IdentType As String = WinIdent.AuthenticationType
```

If you have to perform role-based validation repeatedly, binding the *WindowsPrincipal* to the thread is more efficient, so that the information is readily available. In the previous example, you did not bind the *WindowsPrincipal* to the thread because it was intended to be used only once. However, it is good practice to always bind the *WindowsPrincipal* to the thread because in case a new thread is created, the principal is also copied to the new thread:

1. Create a principal policy based on the *WindowsPrincipal* and bind it to the current thread. This initializes an instance of the *WindowsIdentity* object, creates an instance of the *WindowsPrincipal* object, binds the *WindowsIdentity* to it, and then binds the *WindowsPrincipal* to the current thread. This is all done in a single statement:

   ```
   AppDomain.CurrentDomain.SetPrincipalPolicy(PrincipalPolicy.
       WindowsPrincipal)
   ```

2. Get a copy of the *WindowsPrincipal* object that is bound to the thread:

   ```
   Dim WinPrinc As WindowsPrincipal = Ctype(Thread.CurrentPrincipal, _
                                            WindowsPrincipal)
   ```

It is possible to bind the *WindowsPrincipal* in the first method of creation to the thread. However, your code must be granted the *SecurityPermission* permission to do so. If that is the case, you bind the principal to the thread with the following:

```
Thread.CurrentPrincipal = WinPrinc
```

GenericPrincipal

In a situation in which you do not want to rely on the Windows authentication but want the application to take care of it, you can use the *GenericPrincipal*.

NOTE

Always use an authentication method before letting a user access your application. Authentication, in any shape or form, is the only way to establish an identity. Without it, you are not able to implement role-based security.

Let's assume that your application requested a username and password from the user, checked it against the application's own authentication database, and established the user's identity. You then have to create the *GenericPrincipal* to be able to perform role-based verifications in your application:

1. Create a *GenericIdentity* object for the *User1* you just authenticated:

   ```
   Dim GenIdent As New GenericIdentity("User1")
   ```

2. Create the *GenericPrincipal* object, bind the *GenericIdentity* object to it, and add roles to the *GenericPrincipal*:

   ```
   Dim UserRoles as String() = {"Role1", "Role2", "Role5"}
   Dim GenPrinc As New GenericPrincipal(GenIdent, UserRoles)
   ```

3. Bind the *GenericPrincipal* to the thread. Again, you need *SecurityPermission*:

   ```
   Thread.CurrentPrincipal = GenPrinc
   ```

Manipulating Identity

You can manipulate the identity that is held by a principal object in two ways. The first is replacing the principal; the second is by impersonating.

Replacing the principal object on the thread is a typical action you perform in applications that have their own authentication methods. To be able to replace a principal, your code must have been granted the *SecurityPermission*, or more specifically, the *SecurityPermission* attribute *ControlPrincipal*. This will allow your own code to be able to pass on the *PrincipalObject* to other code. This attribute grants you the permission to manipulate the principal, so you are allowed by the CLR to pass on the principal. Replacing the principal object can be done by performing these steps:

1. Create a new identity and principal object, and initialize it with the proper values.
2. Bind the new principal to the thread:

   ```
   Thread.CurrentPrincipal = NewPrincipalObject
   ```

Impersonating is also a way of manipulating the principal, with the intent to take on the identity of another user to perform some actions on his behalf. You can identify two variations:

- The code has to impersonate the *WindowsPrincipal* that is attached to the thread. This might seem a little odd, but you have to remember that your code is part of an application domain that runs in a process. A user—whether a system account, a service account, or even an interactive user—starts this process on the Windows platform. Although the principal can be used to perform role-based verification within the code, accessing protected resources is still done with the identity of the process user, unless you actively use the user account of principal through impersonation.
- The code has to impersonate a user that is not attached to the current thread. The first thing you have to do is obtain the Windows token of the user you want to impersonate. This has to be done with the unmanaged code *LogonUser*. The obtained token has to be passed to a new *WindowIdentity* object. Now you have to call the *Impersonate* method of *WindowsIdentity*. The old identity—hence, token—has to be saved in a new instance of *WindowsImpersonationContext*.

At the end of the impersonation, you have to change back to the original user account by calling the *Undo* method of the *WindowsImpersonationContext*.

Remember, the principal object is not changed; rather, the *WindowsIdentity* token, representing the Windows account, is switched with the current token. At the end of the impersonation, the tokens are switched back again, as shown in the following steps:

1. Call the *LogonUser* method, located in the unmanaged code library advapi32.dll. You pass the username, domain, password, logon type, and logon provider to this method that will return you a handle to a token. For the sake of the example, we will call it *hImpToken*.
2. Create a new *WindowsIdentity* object and pass it the token handle:

   ```
   Dim ImpersIdent As New WindowsIdentity(hImpToken)
   ```

3. Create a *WindowsImpersonationContext* object and call the *Impersonate* method of *ImpersIndent*:

   ```
   Dim WinImpersCtxt As WindowsImpersonationContext =
   ImpersIdent.Impersonate()
   ```

4. At the end of the call, the original Windows token has to be put back in the *Identity* object:

   ```
   WinImpersCtxt.Undo()
   ```

You could have done Steps 2 and 3 in one statement that looks like this:

```
Dim WinImpersCtct As WindowsImpersonationContext = _
          WindowsIdentity.Impersonate(hImptoken)
```

Remember that you cannot impersonate when you use a *GenericPrincipal* because it does not reference a Windows identity. For generic principals, you will need to replace the principal with one that has a new identity.

Role-Based Security Checks

Having discussed the creation and manipulation of *PrincipalObject*, it is time to take a look at how they can assist you in performing role-based security checks. Here is where *PrincipalPermission*, already mentioned in the beginning of the section Role-Based Security, comes into play. Using *PrincipalPermission*, you can make checks on the active principal object, be it the *WindowsPrincipal* or the *GenericPrincipal*. The active principal object can be one you created to perform a one-time check, or it can be the principal you bound to the thread. Like the code access permissions, the *PrincipalPermission* can be used in both the declarative and the imperative way.

To use *PrincipalPermission* in a declarative manner, you need to use the *PrincipalPermissionAttribute* object in the following way:

```
Public Shared Function
<PrincipalPermissiobAttribute(SecurityAction.Demand, _
                          Name := "User1", Role := "Role1")> Act2()
As Integer
           ' body of the function
End Function
<assembly: PrincipalPermissionAttribute(SecurityAction.Demand, _
  Role := 'Administrator')>
```

To use the imperative manner, you can perform the *PrincipalPermission* check as shown:

```
Dim PrincPerm As New PrincipalPermission("User1", "Role1")
PrincPerm.Demand()
```

It is also possible to use the imperative to set the *PrincipalPermission* object in two other ways:

```
Dim PrincState As PermissionState = Unrestricted
Dim PrincPerm As New PrincipalPermission(PrincState)
```

The permission state (*PrincState*) can be *None* or *Unrestricted*, where *None* means the principal is not authenticated. Therefore, the username is *Nothing*, the role is *Nothing*, and *Authenticated* is false. *Unrestricted* matches all other principals.

```
Dim PrincAuthenticated As Boolean = True
Dim PrincPerm As New PrincipalPermission("User1", "Role1",
  PrincAuthenticated)
```

The *IsAuthenticated* field (*Princauthenticated*) can be true or false.

In a situation in which you want *PrincipalPermission.Demand()* to allow more than one user/role combination, you can perform a union of two *PrincipalPermission* objects. However, this is only possible if the objects are of the same type. Thus, if one *PrincipalPermission* object has set a user/role, and the other object uses *PermissionState*, the CLR throws an exception. The union looks like this:

```
Dim PrincPerm1 As New PrincipalPermission("User1", "Role1")
Dim PrincPerm2 As New PrincipalPermission("User2", "Role2")
PrincPerm1.Union(PrincPerm2).Demand()
```

The *Demand* will succeed only if the principal object has the user *User1* in the role *Role1* or *User2* in the role *Role2*. Any other combination fails.

As mentioned before, you can also directly access the principal and identity object, thereby enabling you to perform your own security checks without the use of *PrincipalPermission*. Besides the fact that you can examine a little more information, it also prevents you from handling exceptions that can occur using *PrincipalPermission*. .You can query the WindowsPrincipal in the same way the *PrincipalPermission* does this:

- The name of the user by checking the value of *WindowsPrincipal.Identity.Name*:

  ```
  If (WinPrinc.Identity.Name = "User1") or _
      WinPrinc.Identity.Name.Equals("DOMAIN1\User1") Then
  End If
  ```

- An available role by calling the *IsInRole* method:

  ```
  If (WinPrinc.IsInRole("Role1")) Then
  End If
  ```

- Determining if the principal is authenticated, by checking the value of *WindowsPrincipal.Identity.IsAuthenticated*:

```
If (WinPrinc.Identity.IsAuthenticated) Then
End If
```

Additionally for *PrincipalPermission*, you can check the following *WindowsIdentity* properties:

- ***AuthenticationType*** Determines the type of authentication used. Most common values are NTLM and Kerberos.
- ***IsAnonymous*** Determines if the user is identified as an anonymous account by the system.
- ***IsGuest*** Determines if the user is identified as a guest account by the system.
- ***IsSystem*** Determines if the user is identified as the system account of the system.
- ***Token*** Returns the Windows account token of the user.

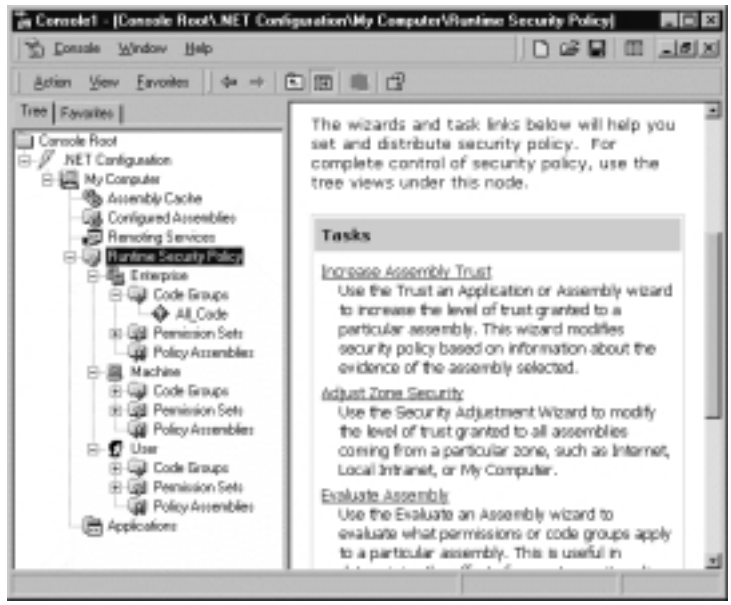
Security Policies

This section takes a closer look at the way in which security policies are constructed and the way you can manage them. To create and modify a security policy, the .NET Framework provides you two tools: a command-line interface (CLI) tool, called caspol.exe (see the section *Security Tools*) and a Microsoft Management Console snap-in, mcscorcfg.msc (Figure 5.5). The latter will be used for demonstration purposes because it is more visual and intuitive.

As you can see in Figure 5.5, the security policy model is comprised of the following:

- The runtime security policy levels are:
  - **Enterprise** Valid for all managed code that is used within the entire organization (enterprise); therefore, this will have "by nature" a restrictive policy because it references a large group of code.
  - **Machine** Valid for all managed code on that specific computer. Because this already limits the amount of code, you can be more specific with handing out permissions.

Figure 5.5 The .NET Configuration Snap-In



- **User** Valid for all the managed code that runs under that Windows user. This will normally be the account that starts the process in which the CLR and managed code runs. Because the identity of the user is very specific, the granted permissions can also be more specific, thus less restrictive.

- A code groups hierarchy that exists for each of the three policy levels. We will look at how you can add code groups to the default structure, which already exists for user and machine.
- (Named) Permission Sets. By default, the .NET Framework comes with seven named permission sets:
  - ***FullTrust*** Unlimited access to all protected resources and operations.
  - ***EveryThing*** Granted all .NET Framework permissions, except the security permission *SkipVerification*.
  - ***LocalIntranet*** The default rights given to an application on the local intranet.
  - ***Internet*** The default rights given to an application on the Internet.
  - ***Execution*** Has only the security permission *EnableAssemblyExecution*.

  - ***SkipVerification*** Has only the security permission *SkipVerification*.
  - ***Nothing*** Denied all access to all protected resources and operations.
- Evidence, which is the attribute that the code hands over to the CLR and on which it determines the effective permission set. Evidence is used in the construction of code groups.
- Policy assemblies that list the trusted assemblies that hold security objects used during policy evaluation. You should add your assemblies to the list that implements the custom permissions. If you omit this, the assemblies will not be fully trusted and cannot be used during the evaluation of the security policy.

Understand that the evaluation process of the security policy will result in the effective permission set for a specific assembly. For all of the three policy levels, the code groups are evaluated against the evidence presented by the assembly. All the code groups that meet the evidence deliver a permission set. The union of these sets determines the effective permission set for that particular security policy level. After this evaluation is done at all three security levels, the three individual permission sets are intersected, resulting in the effective permission set for an assembly. This means that the code groups within the three security levels cannot be constructed independently, because this can result in a situation in which an assembly is given a limited permission set that is too limited to run. When you take a look at the permission set for the *All_Code* of the enterprise security policy, you will see that it is *Full Trust*. Doing the same for the *All_Code* of the user security policy, you will see *Nothing*. Because the code group tree of the enterprise is empty, it cannot make evidence decisions; therefore, it cannot contribute to the determination of the effective permission set of the assembly. By setting it to *Full Trust*, it is up to the machine and user security policy to determine the effective permission set.

Because the user code group already has a limited code group tree, the root does not need to participate in the determination of the permission set. By setting it to *Nothing*, it is up to the rest of the code groups to decide what the effective permission group for the user security policy is.

You can determine the permission set of a code group by performing these steps:

1. Run Microsoft Management Console (MMC) by choosing **Start | Run** and typing **mmc**.
2. Open the .NET Management snap-in, via **Console | Add/Remove Snap-in**.

3. Expand the **Console Root | .NET Configuration | My Computer**.
4. Expand **Runtime Security Policy | Enterprise |Code Groups**.
5. Select the code group **All_Code**.
6. Right-click **All_Code** and select **Properties**.
7. Select the **Permission Set** tab.
8. The **Permission Set** field lists the current value.

Creating a New Permission Set

Suppose you decide that none of the seven built-in permissions sets satisfy your need for granting permissions. Therefore, you want to make a named permission set that does suit you. You have a few options:

- Create a permission from scratch.
- Create a new permission set based on an existing one.
- Create a new permission from an XML-coded permission set.

To get a better understanding of the working of the security policy and to get some hands-on experience with the tool, we discuss the different security policy issues in the following exercises.

We use the second option and base our new permission set on the permission set *LocalIntranet* for the user security policy level:

1. Expand the **User** runtime security policy, and expand **Permission Sets** (Figure 5.6).
2. Right-click the permission set **LocalIntranet** and select **Duplicate**; a permission set called **Copy of LocalIntranet** is added to the list.
3. Select the permission set **Copy of LocalIntranet** and rename it to **PrivatePermissions**. Then, right-click it and select **Properties**. Change the **Permission Set Name** to **PrivatePermissions** and, while you're at it, change the corresponding **Permission Set Description**.
4. Change the permissions of the permission set: Right-click the **PrivatePermissions** permission set, and select **Change Permissions**.
5. The **Create Permission Set** dialog box appears (Figure 5.7). You see two permissions lists: on the left, the Available Permissions that are not assigned, and on the right, the list with Assigned Permissions.

Figure 5.6 The Users Permission Sets and Code Groups

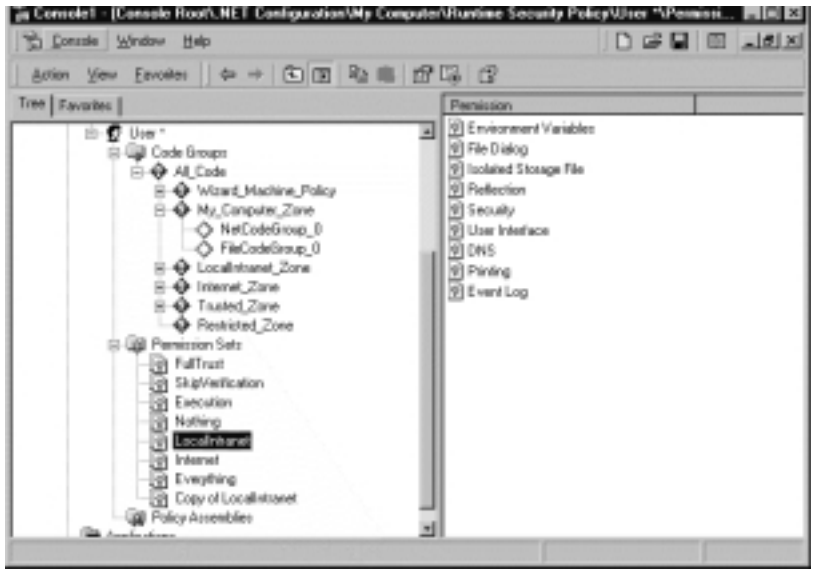
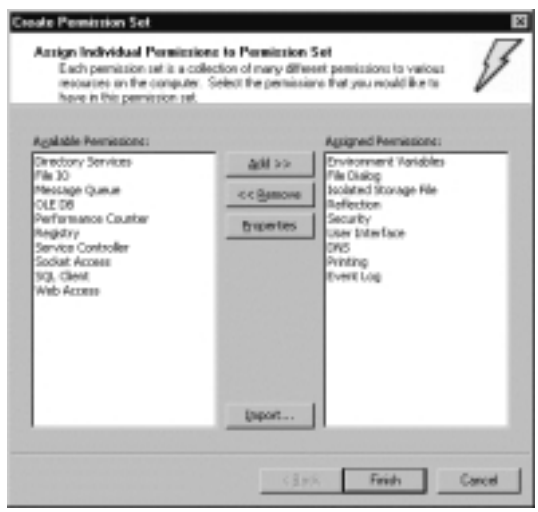


Figure 5.7 Modify the Permission Set Using the Create Permission Set Dialog Box



Between the two Permissions lists are four buttons. The **Add** and **Remove** buttons let you move individual permissions between the lists. Note that you cannot select more than one at the same time; this is done to prevent you from making mistakes. You will better understand a given permission if you select that permission in the Assigned Permissions list and press the **Properties** button. You

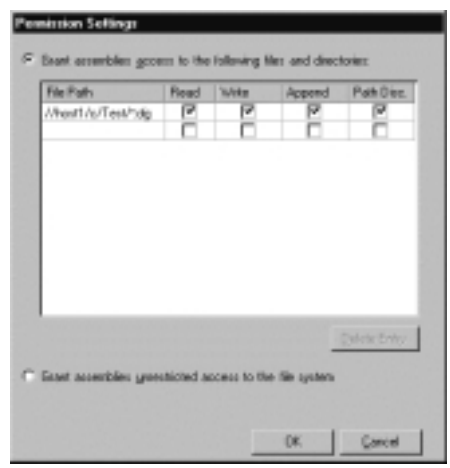
can use the fourth button (**Import**) to load an XML-coded permission set. Now, let's make some modifications to the permission set, because that was the reason to duplicate the permission set:

- Add the *FileIOPermission* to the Assigned Permission list.
- Add the *RegistryPermission* to the Assigned Permission list.
- Modify the *SecurityPermission* properties.

To do so:

1. Select **FileIO** in the Available Permissions list. (Notice that if you have selected a permission in the Assigned Permissions list, this permission stays selected.)
2. Click **Add**. A **Permission Settings** dialog box for the FileIO appears (Figure 5.8). (You can also double-click the permission to add it to the Assigned Permissions list. However, do not double-click an Assigned Permission by accident—this will remove the permission from the Assigned Permission list.) On the Permission Settings dialog box, you are given the option to select between **Grant assemblies access to the following files and directories** and **Grant assemblies unrestricted access to the file system**.

Figure 5.8 Modify the Settings of FileIO Using the Permission Settings Dialog Box

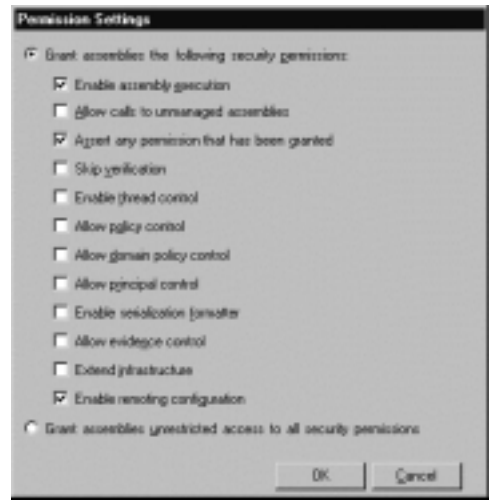


3. Choose the first one, and because it is already selected, we can focus our attention on the empty list window below the option. You may expect

an Add button below the list, especially because there is a **Delete Entry** one. However, there is an auto-add list. You fill in a line, and it is automatically added. Add a second line, and a third empty line will appear.

4. As you saw earlier this chapter, this resembles the way we used *FileIOPermission* and *FileIOPermissionAttribute* to demand and request access to specific files in a specific directory. Go ahead, fill in **"C:\Test*.cfg"**. Surprised that you get an error message? The point is that the field demands that you use UNC names. The advantage is that you can reference to files on other servers in the domain. However, the dialog box checks the existence of the path when you click **OK**, so be sure that the UNC path exists.
5. Fill the File Path with a valid UNC of the machine you are working on, and because we want to give full access, you can check all four boxes. (Note: if you do not check any of the boxes, then this is accepted, because you filled in a File Path. However, if you check the properties of *FileIO* as an assigned permission, you will notice that the line has disappeared—hence, a beta bug!).
6. Click **OK** and you have added a permission to the assigned permission list. You are now ready for the next permission.
7. Double-click the **Registry** permission and a **Permissions Setting** dialog box appears that looks a lot like the one you just saw with **FileIO**. Keep the option **Grant assemblies access to the following registry keys**.
8. Fill the **Key** field with a valid *HKEY* value, such as *HKEY_LOCAL_MACHINE*, and check the **Read** box, so that we can give read permission to the specified Registry tree.
9. Click **OK**, and you have added your second permission to your permission set.
10. The last task is to modify the Security permission. Therefore, select the **Security** permission in the Assigned Permissions list (do not double-click, because that will remove the permission from the list) and click **Properties**.
11. A Permission Settings dialog box (Figure 5.9) appears. You see that the option **Grant assemblies the following security permissions** is selected, together with the properties **Enable assembly execution**, **Assert any permission that has been granted**, and **Enable remoting configuration**.

Figure 5.9 Modify the Settings of Security Using the Permission Settings Dialog Box



12. We also want to grant our security policy the security permission properties. Check **Allow calls to unmanaged assemblies** because we want to make calls to unmanaged code. Also check **Allow principal control** because we want to be able to modify principal settings. Click **OK**, and you are done, for now, with modifying your first permission set.
13. Click **Finish**. You will probably get a warning message stating that you changed your security policy and you have to save it. Up until the point you save the policy, an asterisk (*) will mark the user policy.
14. You can save the policy by right-clicking the **User** runtime security policy and selecting **Save**.

If you want this permission set to also become part of the machine and/or enterprise permission sets, you can simply copy and paste it.

You will also notice two other options: **Reset** and **Restore Policy**. The first resets the policy back to the default setting of the policy. You can try it, but it will wipe out all the changes you made up until now. The latter makes it possible to go back to the previous save. This is possible because for each of the runtime security policies, the settings are saved in an XML-coded file that becomes the current one. Before this happens, it renames the old one with the extension .old. The current one has the extension .cch. The default policy has no extension, so to speak. For the user security policy, you have the following files:

- **security.config** The default security; used by **Reset**.
- **security.config.cch** The current/active policy.
- **security.config.old** The last saved policy version; used by **Restore Policy**.

The enterprise security uses the name enterprisesec.config, and the machine uses the name security.config. This is possible because the user security policy is saved in the user's directory tree in the following folder:

```
Document and Settings\User_Name\Application Data\Microsoft\CLR Security
    config\v1.0.xxxx
```

The enterprise and machine security policies are saved in the following directory:

```
WINNT\Microsoft.NET\Framework\v1.0.xxxx\CONFIG
```

This directory is located by the CLR through the HiveKey:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Catalog42\NetFrameworkv1\
MachineConfigdirectory
```

Because the configuration files are XML-coded, you can open them with a Web browser and examine them. This will give you additional understanding of how the permission sets are set up. This also means that you can modify the default security policies.

Modifying the Code Group Structure

Now that we have created a security permission set, it makes sense to start using it. We can do so by attaching it to a code group. We are going to modify the code groups structure of the user security policy. By default, the user already has a basic structure (Figure 5.10).

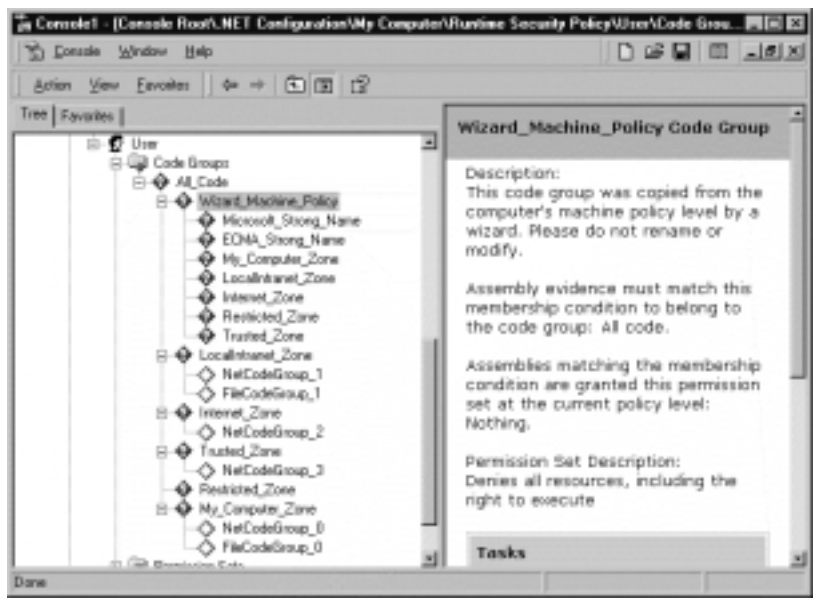
A few things might strike you at first sight:

- There is a code group called *Wizard_Machine_Policy*. The description of this group tells you that a wizard, called the Adjust Security Wizard, copied this group from the computer's policy level and that you should not modify it. This description is not totally true. In fact, if you take a closer look at these code groups, you will see that all groups that end with *_Zone* have a permission set of *Nothing*. This means that you, the user, cannot make use of the permission sets of the machine that are

based on the zone evidence. However, if you are given more permissions based on the zone evidence, this will be toned down by the zone-based permission of the machine policy. The user can have permissions based on zoned evidence that is equal to or less than allowed by the machine. However, you do see zone-based code groups at the same level as the *Wizard_Machine_Policy*, because these are the code groups that are copied from the machine policy.

- The zone-based code groups contain *NetCodeGroup* and *FileCodeGroup*. As the description states, they are generated by the .NET Configuration Tool; hence, the tool we are working with at the moment. The custom code groups are based on XML-code files and can therefore not be edited by the tool. However, you can use the caspol.exe tool to do so. Without going into detail regarding what exactly these groups entail, it suffices to state that they are necessary for you to use the .NET Configuration Tool. If you do not remove or modify them, you might lock yourself out from using this tool.

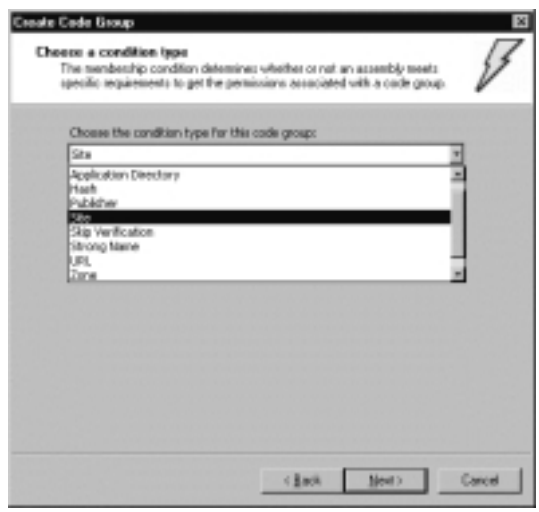
Figure 5.10 The Default Code Group Structure for the User Security Policy



Let's create a small code groups structure that is made up of two code groups directly under the *All_Code* group, and apply our own custom-made permission set *PrivatePermissions* to the *LocalIntranet_Zone* group:

1. If you do not have the MMC with the .NET Management snap-in open, open it now.
2. Expand the tree to **.NET Configuration | My Computer | Runtime Security Policy | User**.
3. Now, expand **Code Groups | All_Code**.
4. Right-click **All_Code** and select **New**; the Create Code Group dialog box appears.
5. You are given two options: **Create a new code Group** and **Import a code group from a XML File**. Use the first option. (Note: For the *NetCodeGroup* and *FileCodeGroup*, the latter is used.)
6. You have to enter at least the **Name** field. For this example, we choose *PrivateGroup_1*. Now, click **Next**.
7. The dialog box shows you a second page called **Choose a condition Type** and has just one field called **Choose the condition type for this code group**. The field has a pull-down menu containing the values from which you can choose. All of these, except the first and last one—All Code and (custom)—are evidence-related (Figure 5.11).

Figure 5.11 Select One of the Available Condition Types for a Code Group



8. Select **Site** from the drop-down menu. A new field, called **Site Name** appears and is related to the **Site** condition. For the sake of the example,

we choose the MSDN Subscribers download site, so we enter the value **msdn.one.microsoft.com** in the site field.

9. Click **Next**, and the third page, called **Assign a Permission Set to the Code Group**, appears.
10. You can choose between the options **Use existing permission set** and **Create a new permission set**. Because the site comes from the Internet, that permission set will do.
11. Select the value **Nothing** from the drop-down menu (Note: The permission set we just made is also part of the list.), and click **Next**.
12. Click **Finish**, and you have created your first code group. While we are at it, let's create the second code group, which will be the child of the code group we just created.
13. Right-click the code group **PrivateGroup_1** and select **New**.
14. Create a new code group named **PrivateGroup_2** and click **Next**.
15. Select the value **Publisher** from the drop-down menu. Below the field, a new box called **Publisher Certificate Details** appears and has to be filled by importing a certificate. You can do this by reading out of a signed assembly using the **Import from Signed File** button. (Note: it should say Import from signed Assembly.) Or, you can import a certificate file, using the **Import from Certificate File** button.
16. For the purpose of this example, we use the certificate from the msdn.one.microsoft.com site. (Note: In case you have forgotten how this is done, you go to a protected site, thus using SSL. You double-click the icon indicating that the site is protected. This opens up the certificate. Go to the **Details** tab and click the **Copy to File** button.)
17. Click the **Import from Certificate File** button, browse to the certificate file (the extension is .cer), and open it. You will see that the field in the certificate box will be filled (Figure 5.12).
18. Click **Next**.
19. Select the existing permission group **LocalIntranet**. We can give more permissions now that we know that the signed assemblies indeed comes from Microsoft MSDN, but also originates from the corresponding Web site.
20. Click **Next**, and then click **Finish**.

Figure 5.12 Importing a Certificate for a Publisher Condition in a Code Group

Before tackling our last task, let's recap what we have done. We were concerned with creating a permission set for signed assemblies that come from the msdn.one.microsoft.com site. So, what if the assembly comes from this Web site but is not signed? It meets the condition of *PrivateGroup_1*, so it will get the permission set of this code group. Because this is Nothing, this would mean that these assemblies are granted no permission. However, because the msdn.one.microsoft.com site comes from the Internet Zone, it also meets the condition of the code group *Internet_Zone*, which grants any assembly from this zone the Internet permission set. Moreover, because a union is taken from all the granted permission sets, these assemblies will still have enough permissions to run.

Why not make the *PrivateGroup_2* a child of *Internet_Zone*, because unsigned assemblies from msdn.one.microsoft.com are granted the Internet permission set anyway? The reason is simple: we only want to give signed assemblies from msdn.one.Microsoft.com additional permission if they also originate from the appropriate Web site. In case such a signed assembly originates from another Web site, we treat it as any other assembly coming from an Internet Zone. The reason for giving *PrivateGroup_1* the Nothing permission set is that it is only there to force assemblies to meet both conditions, and *PrivateGroup_1* is just an intermediate stage to meet all conditions.

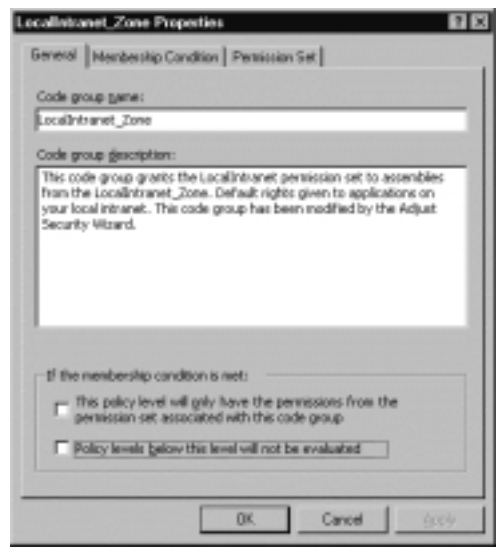
What you have to keep in mind is that we only discussed how the actual permission set is determined at the user security policy level. This will be inter-

sected with the actual permission set determined on the machine level. Moreover, because at the machine level the assembly will be given only the Internet permission set, our signed assembly will wind up with the effective permission set of Internet. Normally, the actual permission set of the enterprise is also taken into the intersection, but because that code group tree has only the *All_Code* code group with full trust, it will play no role in the intersection of this example.

Our last task, replacing a permission set, should be straightforward by now:

1. Right-click the code group *LocalIntranet_Zone* and select **Properties**. The **LocalIntranet_Zone Properties** dialog box appears (Figure 5.13).

Figure 5.13 Setting Attributes in the General Tab of the Code Group Permission Dialog Box



2. Select the **Permission Set** tab.
3. Open the pop-up menu with available permission sets and select **PrivatePermissions**. You will see that the list box will reflect the permissions that make up the *PrivatePermissions* permission set.
4. Click **Apply** and go back to the **General** tab.

On this tab is a frame called **If the membership condition is met**, which shows two options:

- **This policy level will have only the permissions from the permission set associated with this code group**. This refers to the code group attribute *Exclusive*.
- **Policy levels below this level will not be evaluated**. This refers to the code group attribute *LevelFinal*.

Both need some explanation, so let's go back to our msdn.one.microsoft.com example. Suppose you open the Properties dialog box of the *Internet_Zone* code group and check the **Exclusive** option (of course, you have to save it first for it to become active). We received a signed assembly from msdn.one.microsoft.com that also originates from this site. We had established that it would be granted the *LocalIntranet_Zone* permission at the user policy level. But now the **Exclusive** option comes into play. Because our signed assembly also meets the *Internet_Zone* condition, the Internet permission set is valid. The exclusive that is set for the *Internet_Zone* code group forces all other valid permission sets to be ignored by not taking a union of these permission sets. Instead, the permission set with the Exclusive attribute becomes the actual permission set for the user policy level. Because it will be intersected with the Actual permission sets of the other security levels, it also determines the maximum set of permissions that will be granted to the signed assembly. Use this attribute with care, because from all the code groups of which an assembly is a member—hence, meets the condition—only one can have the Exclusive attribute. The CLR determines if this is the case. When the CLR determines that an assembly meets the condition of more than one code group with the Exclusive attribute, it will throw an exception, and it fails to determine the Effective permission set and the assembly is not allowed to execute.

The way in which the *LevelFinal* is handled is more straightforward. Understand that by establishing the effective permission set of an assembly, the CLR evaluates the security policies starting at the highest level (*Enterprise*, followed by *User* and *Machine*). Again, take our MSDN example. We set a *LevelFinal* in the *PrivateGroup_2* code group and removed the Exclusive attribute from *Internet_Zone*. When the effective permission set for a signed assembly from msdn.one.microsoft.com that originates from that Web site has to be established, the CLR starts with determining the actual permission set of the enterprise policy level. This is for *All_Code* Full Trust, effectively taking this policy level out of the intersection of actual permission sets. Now the user policy level gets its turn in establishing the actual permission set. As you know by now, this will be equal to the *LocalIntranet_Zone* permission set. However, the CLR has also

encountered the *LevelFinal* attribute. It refrains from establishing the actual permission set of the machine policy level and intersects the actual permission sets from the enterprise and user policy level. The actual permission set will be equal to *LocalIntranet_Zone*.

Because the machine policy level is not considered, the actual permission set in this case has more permission than in the situation in which the *LevelFinal* attribute has not been set.

Remoting Security

Discussing security between systems always provides a new set of security issues. This is no exception for remoting. Let's start with the communication between systems. If you use an *HttpChannel*, you can make use of the SSL encryption. The *FtpChannel* does not have encryption, but if both servers support IPSec, you are able to create a secured channel through which the *FtpChannel* can communicate.

The next issue is to what extent you trust the other system. Even with a secure channel in place, how do you know that the other system has not been compromised? You need at least a sturdy authentication mechanism in place, and need to avoid the use of anonymous users, although this will not always be possible. At least try to use NTLM or Kerberos for authentication. The latter is a perfect vehicle for handling impersonation between multiple systems. If you need to use anonymous users, you can use IIS as the storefront and let the IIS handle the impersonation. You can also use a proxy to prevent a user from directly accessing your IIS.

The messages that are exchanged should always be signed so you are able to verify the sender and/or origin. Even when you are sure that a message is transported over a secured channel, you are never sure if the message that is put in this channel has been sent out of ill intent.

This chapter has discussed the use of code access and role-based security. The more thoroughly you use this runtime security instrument, the better you can control the remoting security.

Cryptography

There is no subject about security that does not reference cryptography. Although it is an absolute necessity to create a secure environment, it is not the "Holy Grail" of security. This section highlights the cryptography features that come with the .NET Framework. If you already have worked with Windows 2000 Cryptographic Service Providers (CSPs) and/or used the CryptoAPI, you know nearly everything there is to know about cryptography in the .NET Framework.

The most important observation is that the ease of use of crypto functionalities has improved a lot over the way we had to use the CryptoAPI, which only was available for C/C++. An important addition in the design concept of the cryptography namespace is the use of *CryptoStreams*, which make it possible to chain together any cryptographic object that makes use of CryptoStreams. This means that the output from one cryptographic object can be directly forwarded as the input of another cryptographic object without the need of storing the output result in an intermediate object. This can enhance the performance significantly if large pieces of data have to be encoded or hashed. Another addition is the functionality to sign XML code, although only for use within the .NET Framework security system. To what extend these methods comply with the proposed standard RFC 3075 is unclear.

Within the .NET Framework, three namespaces involve cryptography:

- ***System.Security.Cryptography*** The most important one; resembles the CryptoAPI functionalities.
- ***System.Security.Cryptography*** .X509 certificates. Relates only to the X509 v3 certificate used with Authenticode.
- ***System.Security.Cryptography.Xml*** For exclusive use within the .NET Framework security system.

The cryptography namespaces support the following CSP classes that will be matched on the Windows 2000 CSPs, by the CLR. If a CSP is available within the .NET Framework, this does not automatically imply that the corresponding Windows 2000 CSP is available on the system the CLR is running:

- ***DESCryptoServiceProvider*** Provides the functionalities of the symmetric key algorithm Data Encryption Standard.
- ***DSACryptoServiceProvider*** Provides the functionalities of the asymmetric key algorithm Data Signature Algorithm.
- ***MD5CryptoServiceProvider*** Provides the functionalities of the hash algorithm Message Digest 5.
- ***RC2CryptoServiceProvider*** Provides the functionalities for the symmetric key algorithm RC 2 (named after the inventor: Rivest's Cipher 2).
- ***RNGCryptoServiceProvider*** Provides the functionalities for a Random Number Generator.

- ***RSACryptoServiceProvider*** Provides the functionalities for the asymmetric algorithm RSA (named after the inventors Rivest, Shamir, and Adleman).
- ***SHA1CryptoServiceProvider*** Provides the functionalities for the hash algorithm Secure Hash Algorithm 1.
- ***TripleDESCryptoServiceProvider*** Provides the functionalities for the symmetric key algorithm 3DES.

To be complete, short descriptions of symmetric key algorithm, asymmetric key algorithm, and hash algorithm are given. A *symmetric key algorithm* enables you to encrypt/decrypt data that is sent between you and another party. The same key is used to both encrypt and decrypt the data. That is why it is called a symmetric algorithm. This algorithm forces you to exchange the key with your counter party, but this must be done in a way that no other party can intercept this key. Because symmetric key algorithms are often used for a short exchange of data, it is also referred to as *session key algorithm*. For the exchange of session keys, the parties involve use an asymmetric key algorithm.

An *asymmetric key algorithm* makes use of a *key pair.* One is private and is kept under lock and key by the owner, and the other is public and available to everyone. Because the algorithm uses two related but different keys to encrypt and decrypt, it is called an *asymmetric algorithm*, but is also referenced as a *public key algorithm*. The public key is wrapped in a certificate that is a "proof of authenticity," and that certificate has to be issued by an organization that is trusted by all involved parties. This organization is called a *certificate authority* (CA), of which VeriSign is the best known. So, what about using an asymmetric key algorithm to exchange symmetric keys? The best example is two Windows 2000 servers that need to regularly set up connection between both servers on behalf of their users. Each connection—hence, session—has to be secured and needs to use a session key that is unique in relation to the other secured sessions. The servers exchange a session key for every connection. Both have an asymmetric key-pair and have exchanged the public key in a certificate. Therefore, if one server wants to send a session key to the other server, it uses the public key of the other server to encrypt the session key before it sends it. The server knows that only the other server can decrypt the session key because that server has the private key that is needed to decrypt the session key.

A *hash algorithm,* also referred to as a *one-way hash algorithm*, can take a variable piece of data and transform it to a fixed-length piece of data, called a *hash* or

message digest that is nearly always much shorter; for example, 160 bits for SHA-1. *One-way* means that you cannot derive the source data by examining only the digest. Another important feature of the hash algorithm is that it generates a hash that is unique for each piece of data, even if just one bit of data is changed. You can see a hash value as the fingerprint of a piece of data. Let's say, for example, you send someone a plaintext e-mail. How do you and the receiver of the e-mail know that the message was not altered while it was sent? Here is where the message digest comes in. Before you send your e-mail, you apply a hash algorithm on that message, and you send the message and message digest to the receiver. The receiver can perform the same hash on the message, and if both the digest and the message are the same, the message has not been altered. Yes, someone who alters your message can also generate a new digest and obscure his act. Well, that is where the next trick comes in. When you send the digest, you encrypt it with your own private key, of which you know the receiver has the public part. This not only prevents the message from being changed without you and the receiver discovering it, but also confirms to the receiver that the message came from you and only you. How?

Well, let's assume that someone intercepts your message and wants to change it. He has your public key, so he can decrypt your message digest. However, because he doesn't have your private key, he is unable to encrypt a newly generated digest. Therefore, he cannot go forward with his plan to change the e-mail without anyone finding out. Eventually, the e-mail arrives at the receiver's Inbox. He takes the encrypted digest and decrypts it using your public key. If that succeeds, he knows that this message digest must have been sent by you because you are the only one who has access to the private key. He calculates the hash on the message and compares both digests. If they match, he not only knows that the message hasn't been tampered with, but also that the message came from only you because every message has a unique hash. Moreover, because he already established that the encrypted hash came from you, the message must also come from you.

Security Tools

The .NET Framework comes with 10 command-line security tools (Table 5.4) that help you to perform your security tasks. For a more thorough description of these tools, you should consult the .NET Framework documentation.

Table 5.4 Command-Line Security Tools

Name of Tool	Name of Executable	Description
Code Access Security Policy Utility	Caspol.exe	This tool can perform any operation in relation to the code access security policy. Because it can do more than the .NET Configuration Tool we have been using in this chapter, it is important that you familiarize yourself with it.
Certificate Verification Utility	Chktrust.exe	With this tool, you can check a file that has been signed using Authenticode.
Certificate Creation Utility	Makecert.exe	Creates a X.509 certificate for testing purposes. A option you might consider is to install the Certificates Services on Windows 2000, which makes it much easier to create and maintain certificates for development and testing purposes.
Certificate Manager Utility	Certmgr.exe	This utility manages your certificates, certificate trust lists, and so on. Use the Microsoft Management Console with the Certificates snap-in, which enables you to maintain not only your own certificates, but also (if you have the rights) the certificates of your computer and service accounts.
Software Publisher Certificate Test Utility	Cert2spc.exe	This tool creates a software publishers certificate for one or more X.509 certificates.
Permissions View Utility	Permview.exe	This tool enables you to view the requested permissions of an assembly.
PE Verify Utility	Peverify.exe	This tool enables you to verify the type safety of a portable executable file.
Secutil Utility	Secutil.exe	This tool extracts strong name or public key information from an assembly and converts it so that you can use it directly in your code (for example, for a permission demand).
File Signing Utility	Signcode.exe	This tool enables you to sign a PE file with an Authenticode signature. If this utility is called with no command-line options, a Digital Signature Wizard is started.

Continued

Table 5.4 Continued

Name of Tool	Name of Executable	Description
Strong Name Utility	Sn.exe	This tool enables you to sign assemblies with strong names.
Set Registry Utility	Setreg.exe	This tools enables you to set Registry keys for use of public key cryptography. If you call this utility without options, it will just list the settings.
Isolated Storage Utility	Storeadm.exe	This tool enables you to manage isolated storage for the current user.

Securing XML—Best Practices

Just as with HTML documents, digital certificates are the best way in which to secure any document that has to transverse the Internet. Anytime you need to perform a secure transaction over the Internet, a digital certificate should be involved, whether the destination is a browser or an application. Certificates are used by a variety of public key security services and applications that provide authentication, data integrity, and secure communications across nonsecure networks such as the Internet. From the developer's perspective, use of a certificate requires it to be installed on the Web server, and that the HTTPS protocol is used instead of the typical HTTP.

Access to XML and XSL documents on the server can be handled through file access restrictions just like any other file on the server. Unfortunately, if you are performing client-side XSL transformations, this requires that all the files required to perform the transformation be exposed to the Internet for anyone to use. One way to eliminate this exposure is to perform server-side transformation. All XML and XSL documents can reside safely on the server where they are transformed, and only the resultant document is sent to the client.

XML Encryption

The goal of the XML Encryption specification is to describe a digitally encrypted Web resource using XML. The Web resource can be anything from an HTML document to a GIF file, or even an XML document. With respect to XML documents, the specification provides for the encryption of an element, including the start- and end-tags, the content within an element between the

start- and end-tags, or the entire XML document. The encrypted data is structured using the *<EncryptedData>* element that contains information pertaining to encrypting and/or decrypting the information. This information includes the pertinent encryption algorithm, the key used for encryption, references to external data objects, and either the encrypted data or a reference to the encrypted data. The schema as defined so far is shown in Figure 5.14.

Figure 5.14 XML Encryption DTD

```
<!DOCTYPE schema
   PUBLIC "-//W3C//DTD XMLSCHEMA 200010//EN"
http://www.w3.org/2000/10/XMLSchema.dtd
  [
   <!ATTLIST schema xmlns:ds CDATA #FIXED
     "http://www.w3.org/2000/10/XMLSchema">
   <!ENTITY enc "http://www.w3.org/2000/11/temp-xmlenc">
   <!ENTITY enc 'http://www.w3.org/2000/11/xmlenc#'>
   <!ENTITY dsig 'http://www.w3.org/2000/09/xmldsig#'>
  ]>


<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
       xmlns:ds="&dsig;"
       xmlns:xenc="&enc;"
       targetNamespace="&enc;"
       version="0.1"
       elementFormDefault="qualified">

<element name="EncryptedData">
  <complexType>
    <sequence>
      <element ref="xenc:EncryptedKey" minOccurs=0/
          maxOccurs="unbounded"/>
      <element ref="xenc:EncryptionMethod" minOccurs=0/>
      <element ref="ds:KeyInfo" minOccurs=0/>
      <element ref="xenc:CipherText"/>
```

Continued

Figure 5.14 Continued

```
    </sequence>
    <attribute name="Id" type="ID" use="optional"/>
    <attribute name="Type" type="string" use="optional"/>
  </complexType>
</element>

<element name="EncryptedKey">
  <complexType>
    <sequence>
      <element ref="xenc:EncryptionMethod" minOccurs=0/>
      <element ref="xenc:ReferenceList" minOccurs=0/>
      <element ref="ds:KeyInfo" minOccurs=0/>
      <element ref="xenc:CipherText1"/>
    </sequence>
    <attribute name="Id" type="ID"  use="optional"/>
    <attribute name="NameKey" type="string" use="optional"/>
  </complexType>
</element>

<element name="EncryptedKeyReference">
  <complexType>
    <sequence>
      <element ref="ds:Transforms" minOccurs="0"/>
    </sequence>
    <attribute name="URI" type="uriReference"/>
   </complexType>
</element>

<element name="EncryptionMethod">
  <complexType>
    <sequence>
      <any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
```

Continued

Figure 5.14 Continued

```
    <attribute name="Algorithm" type="uriReference" use="required"/>
  </complexType>
</element>

<element name="ReferenceList">
  <complexType>
    <sequence>
      <element ref="xenc:DataReference" minOccurs="0"
        maxOccurs="unbounded"/>
<element ref="xenc:KeyReference" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
  </complexType>
</element>

<element name="DataReference">
  <complexType>
    <sequence>
      <any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="URI" type="uriReference" use="optional"/>
  </complexType>
</element>

<element name="KeyReference">
  <complexType>
    <sequence>
      <any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="URI" type="uriReference" use="optional"/>
  </complexType>
</element>

<element name="CipherText">
```

Continued

Figure 5.14 Continued

```
  <complexType>
    <choice>
      <element ref="xenc:CipherText1"/>
      <element ref="xenc:CipherText2"/>
    </choice>
  </complexType>
</element>

<element name="CipherText1" type="ds:CryptoBinary">

<element name="CipherText2">
  <complexType>
    <sequence>
      <element ref="ds:transforms" minOccurs="0"/>
    </sequence>
  </complexType>
  <attribute name="URI" type="uriReference" use="required"/>
</element>

</schema>
```

The schema is quite involved in describing the means of encryption. The following described elements are the most notable of the specification.

The *EncryptedData* element is at the crux of the specification. It is used to replace the encrypted data, whether the data being encrypted is within an XML document or the XML document itself. In the latter case, the *EncryptedData* element actually becomes the document root. The *EncryptedKey* element is an optional element containing the key that was used during the encryption process. *EncryptionMethod* describes the algorithm applied during the encryption process, and is also optional. *CipherText* is a mandatory element that provides the encrypted data. You might have noticed that the *EncryptedKey* and *EncryptionMethod* are optional—the nonexistence of these elements in an instance is the sender making an assumption that the recipient knows this information.

The processes of encryption and decryption are straightforward. The data object is encrypted using the algorithm and key of choice. Although the specification is open to allow the use of any algorithm, each implementation of the specification should implement a common set of algorithms to allow for interoperability. If the data object is an element within an XML document, it is removed along with its content and replaced with the pertinent *EncryptedData* element. If the data object being encrypted is an external resource, a new document can be created with an *EncryptedData* root node containing a reference to the external resource. Decryption follows these steps in reverse order: parse the XML to obtain the algorithm, parameters, and key to be used; locate the data to be encrypted; and perform the data decryption operation. The result will be a UTF-8 encoded string representing the XML fragment. This fragment should then be converted to the character encoding used in the surrounding document. If the data object is an external resource, then the unencrypted string is available to be used by the application.

There are some nuances to encrypting XML documents. Encrypted XML instances are well-formed XML documents, but might not appear valid when validated against their original schema. If schema validation is required of an encrypted XML document, a new schema must be created to account for those elements that are encrypted. Figure 5.15 contains an XML instance that illustrates the before and after effects of encrypting an element within the instance.

Figure 5.15 XML Document to Be Encrypted

```
<?xml version="1.0"?>
<customer>
      <firstname>John</firstname>
      <lastname>Doe</lastname>
      <creditcard>
            <number>4111111111111111</number>
            <expmonth>12</expmonth>
            <expyear>2000</expyear>
      </creditcard>
</customer>
```

Now, let's say we want to send this information to a partner, but we want to encrypt the credit card information. Following the encryption process laid out by the XML Encryption specification, the result is shown in Figure 5.16.

Figure 5.16 XML Document after Encryption

```
<?xml version="1.0"?>
<customer>
      <firstname>John</firstname>
      <lastname>Doe</lastname>
      <creditcard>
<xenc:EncryptedData
xmlns:xenc='http://www.w3.org/2000/11/temp-xmlenc' Type="Element">
           <xenc:CipherText>AbCd….wXYZ</xenc:CipherText>
          </xenc:EncryptedData>
      </creditcard>
</customer>
```

The encrypted information is replaced by the *EncryptedData* element, and the encrypted data is located within the *CipherText* element. This instance of *EncryptedData* does not contain any descriptive information regarding the encryption key or algorithm, assuming the recipient of the document already has this information. There are some good reasons why you would want to encrypt at the element level considering the XLink and XPointer supporting standards, which enable users to retrieve portions of documents (although there is a debate as to restricting encryption to the document level). You might want to consolidate a great deal of information in one document, yet restrict access only to a subsection. In addition, encrypting only sensitive information limits the amount of information to be decrypted. Encryption and decryption are expensive operations. Although encryption is an important step in securing your Internet-bound XML, there are times when you might want to ensure that you are receiving information from whom you think you are. The W3C is also in the process of drafting a specification to handle digital signatures.

XML Digital Signatures

The XML Digital Signature specification is a fairly stable working draft. Its scope includes how to describe a digital signature using XML and the XML-signature namespace. The signature is generated from a hash over the canonical form of the manifest, which can reference multiple XML documents. To canonicalize something is to put it in a standard format that everyone generally uses. Because the signature is dependent upon the content it is signing, a signature produced from a

non-canonicalized document could possibly be different from that produced from a canonicalized document. Remember that this specification is about defining digital signatures in general, not just those involving XML documents—the manifest may also contain references to any digital content that can be addressed or even to part of an XML document.

To better understand this specification, knowing how digital signatures work is helpful. Digitally signing a document requires the sender to create a hash of the message itself, and then encrypt that hash value with his own private key. Only the sender has that private key and only he can encrypt the hash so that it can be unencrypted using his public key. The recipient, upon receiving both the message and the encrypted hash value, can decrypt the hash value knowing the sender's public key. The recipient must also try to generate the hash value of the message and compare the newly generated hash value with the unencrypted hash value received from the sender. If both hash values are identical, it proves that the sender sent the message, as only the sender could encrypt the hash value correctly. The XML specification is responsible for clearly defining the information involved in verifying digital certificates.

XML digital signatures are represented by the *Signature* element, which has the following structure where "?" denotes zero or one occurrence, "+" denotes one or more occurrences, and "*" denotes zero or more occurrences. Figure 5.17 shows the structure of a digital signature as currently defined within the specification.

Figure 5.17 XML Digital Signature Structure

```
<Signature>
      <SignedInfo>
        (CanonicalizationMethod)
        (SignatureMethod)
        (<Reference (URI=)? >
          (Transforms)?
          (DigestMethod)
          (DigestValue)
        </Reference>)+
      </SignedInfo>
      (SignatureValue)
    (KeyInfo)?
    (Object)*
   </Signature>
```

The *Signature* element is the primary construct of the XML Digital Signature specification. The signature can envelop or be enveloped by the local data that it is signing, or the signature can reference an external resource. Such signatures are detached signatures. Remember, this is a specification to describe digital signatures using XML, and no limitations exist as to what is being signed. The *SignedInfo* element is the information that is actually signed. The *CanonicalizationMethod* element contains the algorithm used to canonicalize the data, or structure the data in a common way agreed upon by most everyone. This process is very important for the reasons mentioned at the beginning of this section. The algorithm used to convert the canonicalized *SignedInfo* into the *SignatureValue* is specified in the *SignatureMethod* element. The *Reference* element identifies the resource to be signed and any algorithms used to preprocess the data. These algorithms can include operations such as canonicalization, encoding/decoding, compression/inflation, or even XSLT transformations. The *DigestMethod* is the algorithm applied to the data after any defined transformations are applied to generate the value within *DigestValue*. Signing the *DigestValue* binds resources content to the signer's key. The *SignatureValue* contains the actual value of the digital signature.

To put this structure in context with the way in which digital signatures work, the information being signed is referenced within the *SignedInfo* element along with the algorithm used to perform the hash (*DigestMethod*) and the resulting hash (*DigestValue*). The public key is then passed within *SignatureValue*. There are variations as to how the signature can be structured, but this explanation is the most straightforward. There you go—everything you need to verify a digital signature in one nice, neat package! To validate the signature, you must digest the data object referenced using the relative *DigestMethod*. If the digest value generated matches the *DigestValue* specified, the reference has been validated. Then, to validate the signature, obtain the key information from the *SignatureValue* and validate it over the *SignedInfo* element.

As with encryption, the implementation of XML digital signatures allows the use of any algorithms to perform any of the operations required of digital signatures, such as canonicalization, encryption, and transformations. To increase interoperability, the W3C does have recommendations for which algorithms should be implemented within any XML digital signature implementations.

You will probably see an increase in the use of encryption and digital signatures when both the XML Encryption and XML Digital Signature specifications are finalized. They both provide a well-structured way in which to communicate each respective process, and with ease of use comes adoption. Encryption will

ensure that confidential information stays confidential through its perilous journey over the Internet, and digital signatures will ensure that you are communicating with whom you think you are. Yet, both these specifications have some evolving to do, especially when they are used concurrently. There's currently no way to determine if a document that was signed and encrypted was signed using the encrypted or unencrypted version of the document. Typically, these little bumps find a way of smoothing themselves out…over time.

NOTE

You can write your own code to perform XSL transformations on the server, or you can use the XSL ISAPI extension to automatically transform the XML page that includes a reference to the XSL style sheet. Some of the advantages to using the ISAPI filter are automatic selection and execution of style sheets on the server, style sheet caching for improved performance, and the option to allow the "pass through" of the XML for client-side processing. To learn more about the XSL ISAPI Extension, visit http://msdn.microsoft.com/xml/general/sxlisapifilter.asp.

Summary

Positioning the .NET Framework as a distributed application environment, Microsoft was well aware that they had to pay attention to how an application can be secured, due to the great risks that distributed security incorporate. That is why they introduced a scalable but rights- and permission-driven security mechanism: scalable because you can as much own your own designed and customized permissions, and rigid because it is always, even if the application takes no notice of permissions. To add to that, the CLR will check the code on type safety (it checks whether the code is trying to stick its nose in places it does not belong) during the JIT compilation.

The .NET Common Language Runtime (CLR) will always perform a security check—called code access security—on an assembly if it wants to access a protected resource or operation. To prevent an assembly from obscuring its restricted permissions by calling another assembly, the CLR will perform a security stack walk. It checks every assembly in a calling chain of assemblies to see if every single one has this permission. If this is not the case, the assembly is not given access to this protected resource or operation.

What permissions an assembly is granted and what permission an assembly requests is controlled in two ways. The first is controlled by code groups that grant permissions to an assembly based on the evidence it presents to the CLR. The assembly itself controls the latter. A secure conscious assembly requests only the permissions it needs, even if the CLR is willing to grant it more permissions. By doing this, the assembly insures itself from being misused by other code that wants to make use of its permission set. A code group hierarchy has to be set up by an administrator, which he can do at different security policy levels: enterprise, user, and machine.

To establish the effective set of permissions, the CLR uses a straightforward and robust method: it determines all valid permission sets based on the evidence an assembly presents per security policy level, and the actual permission set per policy level is the union of the valid permission set. The CLR does this for all the policy levels and intersects the actual permission set to determine the effective permission set of an assembly.

Added to the code access security, the CLR still supports role-based security, although its implementation differs slightly from what you were accustomed to with COM. Every executing thread has a security context called *principal* that references the identity of the user. The principal is also used for impersonation of the executing user. The principal comes in a few forms: based on Windows users

and its authentication, generic and can be controlled by custom-made authentication services; and a base form that enables you to custom-make your own principal and identity. The code can reference the principal to check if the user has a specific role.

Still, the most important security feature is security policies, which allow you to create code groups and build your own permission set that can be enriched with custom permissions. The custom permissions can be added to the .NET Framework without opening up the security system, provided that you make no security mistakes in the coding of the permissions.

As can be expected from every framework that relies on security, the .NET Framework comes with a complete set of cryptography functionalities, equal to what we had with the CryptoAPI, only the ease of use has improved a lot and is no longer dependent on C/C++. To control cryptographic functionalities, such as certificates and code signing, the .NET Framework has a set of security utilities that enable you to control and maintain the security of your application during its development and deployment process.

We need to rely on .NET's security because the XML security is so weak. After all, XML is meant to be just a simple ASCII file for data transfer. In a way, the security of an XML document should not really be left to XML, but rather to the programmer. However, the W3C does have plans to provide several crypto recommendations for XML, but, like any other mathematical algorithm, it is only a matter of time before the encryption is cracked. Your best bet—and your users'—when using XML is to secure it by using a combination of .NET's internal security classes with some decent encrypting.

Solutions Fast Track

The Risks Associated with Using XML

- ☑ Anything and everything on the Internet is vulnerable. Expose only data and code that is absolutely necessary.
- ☑ If information is not meant to be seen, it is much safer to transform the XML document to exclude the sensitive information prior to delivering the document to the recipient, rather than encrypt the information within the document.
- ☑ XSL is a complete programming language, and at times might be more valuable than the information contained within the XML it transforms.

When you perform client-side transformations, you expose your XSL in much the same way that HTML is exposed to the client.

.NET Security as a Viable Alternative

- ☑ Permissions are used to control the access to protected resources and operations.
- ☑ Principal is the security context that is attached to every executing thread in the CLR. It also holds the identity of the user, such as Windows account information, and the roles that user has. It also contributes to the capability of the code to impersonate.
- ☑ Authentication and authorization can be controlled by the application itself or rely on external authentication methods, such as NTLM and Kerberos. Once Windows has authorized a user to execute CLR-based code, the code has to control all other authorization that is based on the identity of the user and information that comes with assemblies, called *evidence*.
- ☑ Security policy is what controls the entire CLR security system. A system administrator can build policies that grant assemblies permissions access to protected resources and operations. This permission granting is based on evidence that the assemblies hand over to the CLR. If the rules that make up the security policy are well constructed, it enables the CLR to provide a secure runtime environment.
- ☑ Type safety is related to the prevention of assembly code to reach into memory/storage of other applications. Type safety is always checked during JIT compilation and therefore before the code is even loaded into the runtime environment. Only code that is granted the SkipVerification permission can bypass type safety checking, unless this is turned off altogether.

Code Access Security

- ☑ Code access security is based on granting an assembly permission and enforcing that it can never gain more permissions. This enforcing is done by what is known as *security stack walking*. When a call is made to a protected resource or operation, the assembly that the CLR demanded from the assembly has a specific permission. However, instead of checking

only the assembly that made the call, the CLR checks every assembly that is part of a calling chain. If all these assemblies have that specific permission, the access to the protected resource/operation is allowed.

- ☑ To be able to write secure code, it is possible to refrain from permissions that are granted to the code. This is done by requesting the necessary permissions for the assembly to run, whereby the CLR gives the assembly only these permissions, under the reservation that the requested permissions are part of the permission set the CLR was willing to grant the assembly anyway. By making your assemblies request a limited permission set, you can prevent other code from misusing the extended permission set of your code. However, you can also make optional requests, which allow the code to be executed even if the requested permission is not part of the granted permission set. Only when the code is confronted with a demand of having such a permission, it must be able to handle the exception that is thrown, if it does not have this permission.
- ☑ The demanding of a caller to have a specific permission can be done using declarative and imperative syntax. Requesting permissions can only be done in a declarative way. Declarative means that it is not part of the actual code, but is attached to an assembly, class, or method using a special syntax enclosed with brackets (**<>**). When the code is compiled to the intermediate language (IL) or a portable executable (PE), these demands/requests are extracted from the code and placed in the metadata of the assembly. This metadata is read and interpreted by the CLR before the assembly is loaded. The imperative way makes the demands part of the code. This can be sensible if the demands are conditional. Because a demand can always fail and result in an exception being thrown by the CLR, the code has to be equipped for handling these exceptions.
- ☑ The code can control the way in which the security stack walk is performed. By using *Assert*, *Deny*, or *PermitOnly*, which can be set with both the declarative and imperative syntax, the stack walk is finished before it reaches the end of the stack. When CLR comes across an *Assert* during a stack walk, it finishes with a Succeed. If it encounters a *Deny*, it is finished with a Fail. With the *PermitOnly*, it succeeds only if the checked permission is the same or is a subset of the permission defined with the *PermitOnly*. Every other demand will fail at the *PermitOnly*.

- ☑ Custom permissions can be constructed and added to the runtime system.

Role-Based Security

- ☑ Every executing thread in the .NET runtime system has an identity that is part if the security context, called *principal*.
- ☑ Based on the principal, role-based checks can be performed.
- ☑ Role-based checks can be performed in a declarative, imperative, and direct way. The direct way is by accessing the principal and/or identity object and querying the values of the fields.

Security Policies

- ☑ A security policy is defined on different levels: enterprise, user, machine, and application domain. The latter is not always used.
- ☑ A security policy has permission sets attached that are built in—such as *FullTrust*, *Internet*, or *Custom Made*. A permission set is a collection of permissions. By grouping permissions, you can easily address them, only using the name of the permission set.
- ☑ The important part of the policy is the security rules, called *code groups*; these groups are constructed in a hierarchy.
- ☑ A code group checks the assembly based on the evidence it presents. If the assembly's evidence meets the condition, the assembly is regarded as a member of this code group and is successively granted the permissions of the permission set related to the code group. After all code groups are checked, the permission sets of all the code groups or which the assembly is a member are united to an actual permission set for the assembly at that security level.
- ☑ The CLR performs this code group checking on every security level, resulting in three or four actual permission sets. These are intersected to result in the effective permission set of permissions granted to the assembly.
- ☑ Remoting limits the extent to which the security policy can be applied. To create a secure environment, you need to secure remoting in such a way that access to your secured CLR environment can be fully controlled.

Cryptography

- ☑ The .NET Framework comes with a cryptography namespace that covers all necessary cryptography functionalities that are at least equal to the CryptoAPI that was used up until now.
- ☑ Using the cryptography classes is much easier than using the CryptoAPI.

Security Tools

- ☑ The .NET Framework comes with a set of security tools that enable you to maintain certificates, sign code, create and maintain security policies, and control the security of assemblies.
- ☑ Two comparable tools enable you to maintain code access security. caspol.exe (Code Access Security Policy Utility) has to be operated from the command-line interface. The .NET Configuration Tool comes as a snap-in for the Microsoft Management Console (MMC) and is therefore more intuitive and easier to use than caspol.exe is.

Securing XML—Best Practices

- ☑ Use existing methods of security to protect your XML. HTTPS works with your XML in the same way it does with HTML.
- ☑ Try to keep everything on the server. Perform your XSL transformation on the server, thus only sending HTML or relevant XML to the client.
- ☑ The goal of the XML Encryption specification (currently in working-draft form) is to describe a digitally encrypted Web resource using XML. The specification provides for the encryption of an element including the start- and end-tags, the content within an element between the start- and end-tags, or the entire XML document. The encrypted data is structured using the *<EncryptedData>* element.
- ☑ The XML Digital Signature specification is a fairly stable working draft. Its scope includes how to describe a digital signature using XML and the XML-signature namespace. The signature is generated from a hash over the canonical form of the manifest, which can reference multiple XML documents.

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** and click on the **"Ask the Author"** form.

Q: I want to prevent an overload of security stack walk; how can I control this?

A: This can indeed become a major concern if it turns out that the code accesses a significant number of protected resources and/or operations, especially if they happen in a long calling-chain. The only way to prevent this from happening is to put in a *SecurityAction.Assert* just before a protected resource/operation is called. This implies that you need a thorough understanding of when a stack walk—hence, demand—is triggered and on what permission this stack walk will be performed. By just placing an *Assert*, you create an uncontrolled security hole. What you can do is the following, which can be applied in the situation in which you make a call to a protected resource, but do this from within a loop-structure. You can also use it in a situation in which you call a method that makes a number of calls to (different) protected resources/operations that trigger the demand for the same type of permission.

The only way to prevent a number of stack walks is to place an imperative assertion on the permission that will be demanded. Now you know that the stack walk will be stopped in its tracks. To close the security hole you just opened, you place an imperative demand for the permission you asserted in front of the assertion. If the demand succeeds, you know that in the other part of the calling-chain, everything is OK in regard to this permission. Moreover, because nothing will change if you check a second or third time, you can save yourself from a lot of unnecessary stack walks. Think about a 1000-fold loop: You just cleared your code from doing redundant 999 stack walks.

Q: When should I use the imperative syntax, and when should I use the declarative?

A: First, make sure that you understand the difference in the effect they take. The imperative syntax makes a demand, or override for that matter, on part of your code. It is executed when the line of code that holds the demand/override is encountered during runtime. The declarative syntax brings these demands and

overrides right into the metadata of the assembly. During the load phase of the assembly, the metadata is extracted and interpreted, meaning that the CLR already takes action on this information. If a stack walk takes place, the CLR can handle overrides much quicker than if they would occur during execution, thus the imperative way. However, demands should only be made at the point they are really necessary. Most of the time, demands are conditional—think about whether the demand is based on a role-based security check. If you would make a demand declarative for a class or method, it will be trigger a stack walk every time this class or method is referenced, even if demands turn out to be not needed. To recap: Make overrides declarative and place them in the header of the method, unless all methods in the class need the assertion; then, you place it in the class declaration. Remember that an assembly cannot have more than one active override type. If you cannot avoid this, you need to use declarative overrides anyway. Make demands imperative and place them just before you have to access a protected resource/operation.

Q: How should I go about building a code group hierarchy?

A: You need to remember four important issues in building a code group hierarchy:

- An assembly cannot be a member of code groups that have conflicting permissions; for example, one with unrestricted *FileIOPermission* and one with a more restricted *FileIOPermission*.
- The bigger the code group hierarchy, the harder it is to maintain.
- The larger the number of permission sets; the harder it is to maintain them.
- The harder it is to maintain code groups and permissions sets, the more likely it is that they contain security holes.

Anyhow, the best approach is the largest common denominator. Security demands simplicity with as few exceptions as possible. Before you start creating custom properties sets, convince yourself that this is absolutely necessary. Nine out of 10 times, one of the built-in permission sets suffices. The same goes for code groups—most assemblies will fit nicely in a code group based on their zone identity. If you conclude that this will not do, add only code groups that are more specific than the zone identity, like the publisher identity, but still apply to a large group of assemblies. Use more than one level in the code group hierarchy only if it is absolutely necessary to check on more

than one membership condition—hence, identity attribute. Add a permission set to the lowest level of the hierarchy only and apply the Nothing permission set to the parent code groups.

Take into account that the CLR will check on all policy levels, so check if you have to modify the code group hierarchy of only one policy level, or that this has to be done on more levels. Remember, the CLR will intersect the actual permission sets of all the policy levels.

Q: How do I know when to use an element versus an attribute when defining the structure of my XML?

A: It is very hard to define catchall rules to determine when to use an element versus an attribute. Remember, though, that you can do very little validation with attributes other than making sure that they exist. For the most part, if there is any doubt, use an element to describe your content.

Q: Are there any XML editors out there?

A: Yes, quite a few, one of which is XML Notepad by Microsoft. The one we personally prefer to use is XML Spy. You might have a small learning curve with the user interface, but it is by far the best XML editor available when considering the price. Sometimes, though, nothing beats XML Notepad when you need something down and dirty.

Q: Do I always have to define a schema for my XML document?

A: No, you don't always need a schema. Schemas are great for when you have to do validation—typically when exchanging XML documents over the Internet. Performing validation all the time might seem like a great idea, but it is a very expensive operation that can bog down a Web server. When shooting out XML to the Web, you typically don't need a schema, although it is a great way to document your XML.

Q: How can I use XSL to make my applications completely browser independent?

A: XSL is a tool you can use to transform XML to HTML. You can create several style sheets. Each can be especially suited for a particular browser, and depending on the browser of the client, you can transform the XML using the respective style sheet. This not only allows you to support Netscape and Internet Explorer, but also allows you to support almost any Internet-enabled device, from handhelds to cell phones.

Chapter 6

Web Development Using XML and ASP.NET

Solutions in this chapter:

- Reviewing the Basics of the ASP.NET Platform
- Reading and Parsing XML
- 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 Online

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

Microsoft has done a great job of bringing ASP and their older languages into the twenty-first century with .NET. ASP.NET is now a full-fledged object-oriented Web application development platform, and has seen many improvements; the main improvement to ASP with .NET is its capability to use the .NET languages and not have to rely on either JScript or VBScript.

The following are some key points of ASP.NET:

- ASP.NET is a key part of the wider Microsoft .NET initiative, Microsoft's new application development platform.
- .NET is both an application architecture to replace the Windows DNA model, and a set of tools, services, applications, and servers based on the .NET Framework and the Common Language Runtime (CLR).
- Rather than just being ASP 4 or an incremental upgrade, ASP.NET is a complete rewrite from the ground up, using all of the advanced features .NET makes available.
- ASP.NET can take advantage of all that .NET has to offer, including support for around 20 or more .NET languages from C# to Perl .NET, and the full set of .NET Framework software libraries.
- Web applications written in ASP.NET are fast, efficient, manageable, scalable, and flexible, and, above all, easy to understand and to code!
- Components and Web applications are all compiled .NET objects written in the same languages, and they offer the same functionality, so there is no need to leave the ASP environment for purely functional reasons.
- You'll have less need for third-party components. With a few lines of code, ASP.NET can talk to XML, serve as or consume a Web service, upload files, "screen scrape" a remote site, or generate an image.

Reviewing the Basics of the ASP.NET Platform

With the .NET Framework and ASP.NET, Microsoft has not just shown itself to be a contender in Web development technologies, but many commentators also believe Microsoft has taken the lead. ASP.NET is well equipped for any task you want to put to it, from building intranets to e-business or e-commerce mega-sites. Microsoft has been very careful to include the functionality and flexibility developers require, while maintaining the easy-to-use nature of ASP:

- With ASP.NET you now have a true choice of languages. All the .NET languages have access to the same foundation class libraries, the same type of systems, equal object orientation and inheritance capabilities, and full interoperability with existing COM components.
- You can use the same knowledge and code investment for everything from Web development to component development or enterprise systems, and developers do not have to be concerned about differences in APIs or variable type conversions, or even deployment.
- ASP.NET incorporates all the important standards of our time, such as XML and SOAP. In addition, with ADO.NET and the foundation class libraries, they are arguably easier to implement than in any other technology, including Java.
- An ASP.NET programmer still only needs a computer with Notepad and the ability to FTP to write ASP code, but now with the .NET Framework command-line tools and the platform's XML-based configuration, this is truer than before!
- Microsoft has included in the .NET Framework an incredibly rich feature set of library classes, from network-handling functions for dealing with Transmission Control Protocol/Internet Protocol (TCP/IP) and Domain Name System (DNS), to XML data and Web services, to graphic drawing.
- In the past, the limitations of ASP scripting meant components were required for functionality reasons, not just for architectural reasons. ASP.NET has access to the same functionality and uses the same languages in which you would create components, so now components are an architectural choice only.
- A .NET developer is shielded from changes in the underlying operating system and API, as the .NET technologies deal with how your code is implemented; and with the Common Type System, you don't have to worry whether the component you are building uses a different implementation of a string or integer to the language in which it will be used.

NOTE

The remainder of this chapter assumes you have previous knowledge of ASP.NET; if you need more help with ASP.NET in general, please feel free to pick up a copy of Syngress' *ASP.NET Web Developer's Guide* (ISBN: 1-928994-51-2).

Reading and Parsing XML

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 can 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 can 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 can 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 document (this document can be found in the source code folder for Chapter 6 on the companion Solutions Web site for the book www.syngress.com/solutions). The code for this example and its output is shown in Figures 6.1 and 6.2, respectively. The code shown in Figure 6.2 is available on the CD. 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 6.2) and its output carefully. 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 6.1 Truncated Output of the *XmlTextReader1.aspx* Code

Figure 6.2 XmlTextReader1.aspx

```
<%@ Page Language = "VB" Debug ="True" %>
<%@ Import Namespace="System.Xml" %>
<Script runat="server">
Sub Page_Load(sender As Object, e As EventArgs)
```

Continued

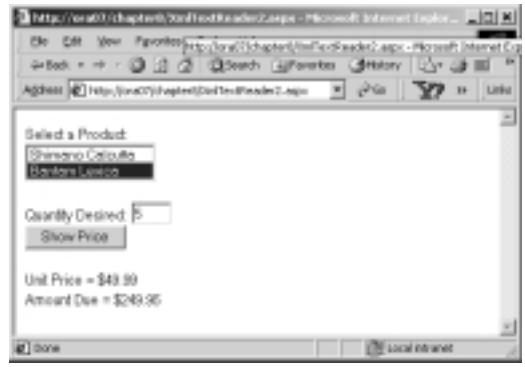
Figure 6.2 Continued

```
  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
  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-space contained in an XML document. 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 6.3. 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 array list 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 6.3 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 white-space, 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 "unclick" event code of the command button "Show Price." We will create a second *XmlTextReader* object based on the catalog2.xml file. Of course, we can 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()* might take us to a "whitespace" node, and thus we have applied a *MoveToContent()* to get to the ListPrice node. Figure 6.4 shows an excerpt of the relevant code. The complete code is available in the XmlTextReader2.aspx file located in the folder for the source code for Chapter 6 on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 6.4 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.

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 can 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*, *WriteAttributes*, *WriteAttributeString*, *WriteComment*, *WriteElementString*, *WriteElementString*, *WriteEndAttribute*, *WriteEndDocument*, *WriteState*, and *WriteStartDocument*.

Generating an XML Document Using *XmlTextWriter*

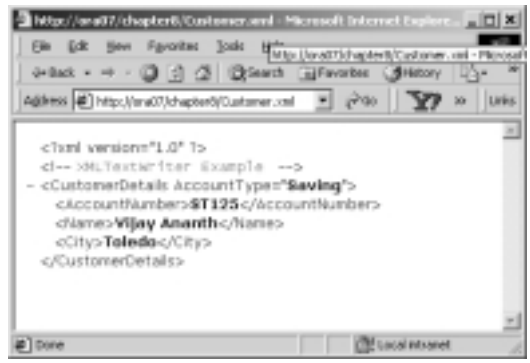
In this section, we will collect user-given data via an .aspx page, and write the information in an XML file. Figure 6.5 shows the runtime view of the application. On the click event of the "Create XML File," the application will create the XML file (in the disk) and display it back in the browser as shown in Figure 6.6.

Figure 6.5 Output of 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 6.6 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 employ the *Response.Redirect(Server.MapPath("Customer.xml"))* to display the XML documents information in the browser. Figure 6.7 shows the complete code for the application. Both Customer.xml and XmlTextWriter1.aspx files are available on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 6.7 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"/>  
<asp:TextBox id="txtAcno" runat="server" width="50" _
text=" ST124" /><br/>
<asp:Label id="lblName" Text="Name :" runat="server" />  
<asp:TextBox id="txtName" runat="server" width="100" text="Vijay
Ananth"/><br/>
<asp:Label id="lblCity" Text="City :" runat="server"/>  
<asp:TextBox id="txtCity" runat="server" width="100" text="Toledo"/><br/>
<asp:Button id="cmdWriteXML" Text="Create XML File" runat="server"
onclick="writeXML"/>
```

Continued

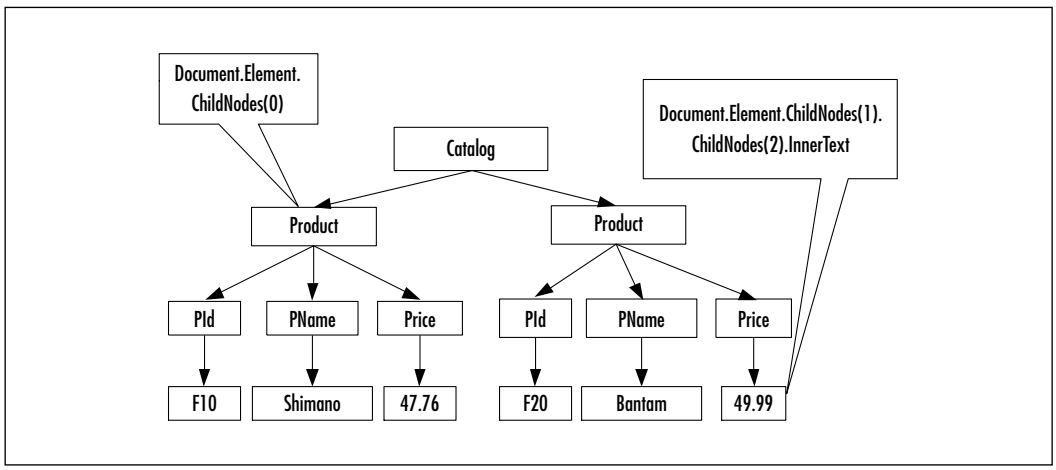
Figure 6.7 Continued

```
<br></form>

<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"))

 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 DOM 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 6.8). 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 6.8 Node Addressing Techniques in an XML DOM Tree



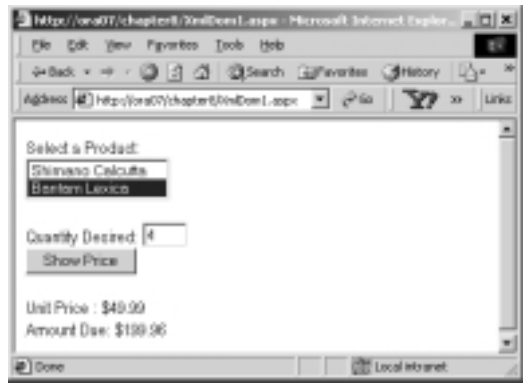
An *XmlDocument* object can be loaded from an *XmlTextReader.* Once it is loaded, we can navigate via the nodes of its tree using numerous methods and properties. Some of the frequently used members include *DocumentElement* (root of the tree), *ChildNodes* (all children of a node), *FirstChild*, *LastChild*,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

If needed, we can 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 DOM. Figure 6.9 shows the output of the code.

Figure 6.9 Output of the *XmlDocument* Object Example



Let's go through the process of loading the *XmlDocument* (DOM tree). There are a number of different ways to load an *XMLDocument* object; we will load it using an *XmlTextReader* object. We 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, which 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
```

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 *SelectedIndex* value will match the corresponding child's ordinal position in the *Catalog* (*DocumentElement*). Figure 6.10 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 companion Solutions Web site for the book (www.syngress.com/solutions).

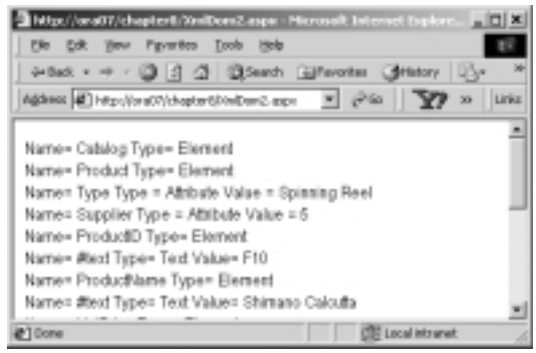
Figure 6.10 Partial Listing of XmlDom1.aspx

```
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 comprised 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 the DOM tree and display the information contained in each of its nodes. Figure 6.11 shows the output of this exercise.

Figure 6.11 Parsing an *XmlDocument* Object



We will develop two subprocedures:

- **DisplayNode(node As XmlNode)** It will receive a node and check if it is a terminal node. If it is, this subprocedure will print its contents. If the node is not a terminal node, then the subprocedure will check if the node has any attributes; if there are, it will print them.
- **TravelDownATree(tree As XmlNode)** It will receive a tree, and at first it will call the *DisplayNode* procedure. Then, it will pass the subtree 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.

Figure 6.12 shows the complete code listing. The code is also available in the file named XmlDom2.aspx on the companion Solutions Web site for the book. 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.

SYNGRESS
syngress.com

Figure 6.12 The Complete Code *XmlDom2.aspx*

```
<%@ Page Language = "VB"  Debug ="True" %>
<%@ Import Namespace="System.Xml" %>
<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"))
```

Continued

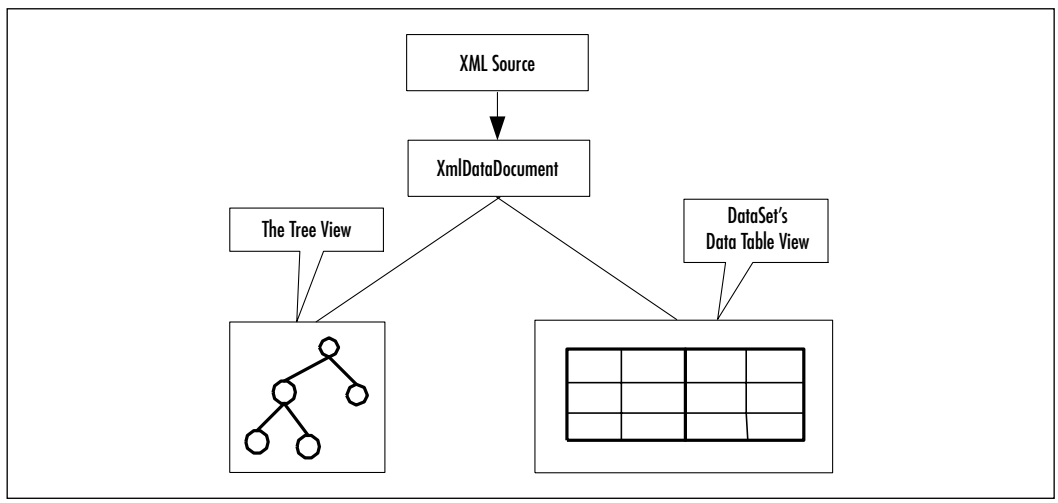
Figure 6.12 Continued

```
    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/>")
       Next
    End If
  End If
 End Sub
</Script>
```

Using the *XmlDataDocument* Class

The *XmlDataDocument* class is an extension of the *XmlDocument* class, and more or less behaves 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 6.13 shows the two views of an *XmlDataDocument*. Because these views are drawn from the same *DataDocument* object, they are automatically synchronized. That means that any changes in either of them will change the other.

Figure 6.13 Two Views of an *XmlDataDocument* Object



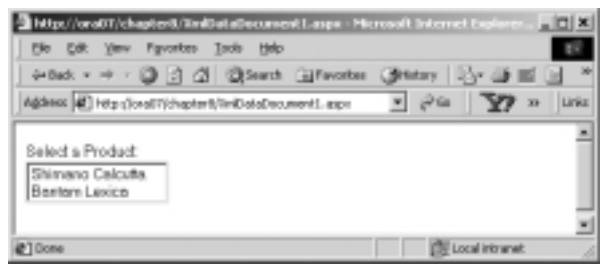
In this section, we will provide three examples:

- 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. Figure 6.14 shows the output of this example. The code for this application is listed in Figure 6.15, and is also available in the file named XmlDataDocument1.aspx on the companion Solutions Web site for the book.

Figure 6.14 Output of XmlDataDocument1.aspx



The *XmlDataDocument* is a pleasant object with which to work. 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 are probably wondering 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.

SYNGRESS
syngress.com

Figure 6.15 XmlDataDocument1.aspx

```
<%@ Page Language = "VB"  Debug ="True" %>
<%@ Import Namespace="System.Xml" %>
<html><head></head><body><form runat="server">
Select a Product: <br/>
```

Continued

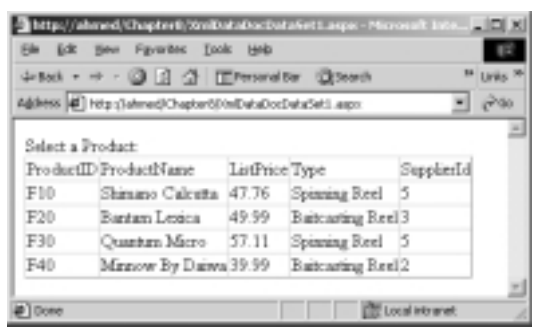
Figure 6.15 Continued

```
<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 (Figure 6.17) is available on the companion Solutions Web site for the book. Figure 6.16 shows the output of this example.

Figure 6.16 Output of *XmlDataDocument DataSet* View Example



If we want to process the XML data as relational data, we first need to load the schema of the XML document. We have generated the following schema for the Catalog3.xml using VS.NET. Figure 6.17 shows the schema specification (also available on the companion Solutions Web site for the book).

SYNGRESS
syngress.com

Figure 6.17 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"
        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. Figure 6.18 shows the complete listing of the code. The code is also available in the XmlDataDocDataSet1.aspx file on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 6.18 Complete Listing for XmlDataDocDataSet1.aspx

```
<%@ Page Language = "VB"  Debug ="True" %>
<%@ Import Namespace="System.Xml" %>
<%@ Import Namespace="System.Data" %>
<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
```

Continued

Figure 6.18 Continued

```
      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 an *XmlDataDocument* Object

In many instances, an XML document might 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 6.19. It is also available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.19 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>
                  <Type>Savings</Type>
                  <Balance>1500.00</Balance>
            </Account>
            <Account>
                  <Type>Checking</Type>
                  <Balance>111.11</Balance>
          </Account>
          <Account>
```

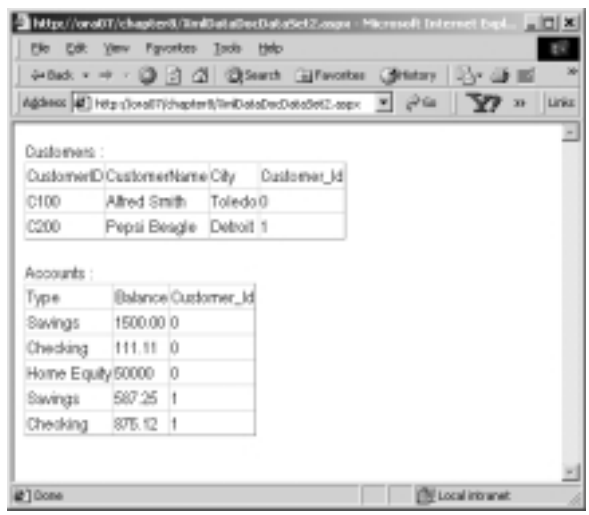
Continued

Figure 6.19 Continued

```
                <Type>Home Equity</Type>
                <Balance>50000</Balance>
            </Account>
        </Customer>
        <Customer>
            --- --- ---
            --- --- ---
        </Customer>
</Bank>
```

If we load the XML document in Figure 6.19 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 data grids as shown in Figure 6.20.

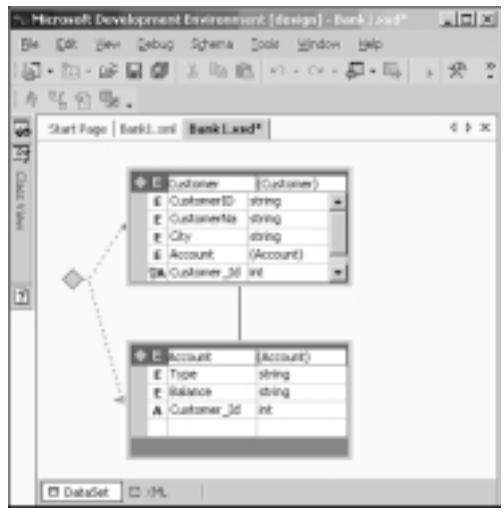
Figure 6.20 Displaying Customer and Accounts Data in Two Data Grids



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 one-to-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* data table. Simultaneously, it inserts the appropriate values of the foreign keys in the *Account* data table. Figure 6.21 shows the *DataSet* view of the generated schema.

Figure 6.21 *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 6.22 shows an excerpt of the generated schema for our XML file. The complete schema is available in a file named Bank1.xsd on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.22 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" />
                <xsd:field xpath="@Customer_Id" />
</xsd:keyref>
```

In the preceding 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 aforementioned objectives.

Figure 6.23 shows the complete listing of the application. It is also available in the xmlDataDocDataSet2.aspx file on the companion Solutions Web site for the book (www.syngress.com/solutions). The code is pretty straightforward. We have loaded two data grids from two data tables of the data set, associated with the *XmlDataDocument* object.

Figure 6.23 Complete Listing for 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()
       '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 illustrated how an *XmlDataDocument* object maps nested XML elements into multiple *DataTable*s. 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, 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 might not be very convenient either. To alleviate these problems, 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).

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 glance, the *XPath* query syntax might appear very complex. However, with a certain amount of practice, you might 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 6.24, and is also available on the companion Solutions Web site for the book (ww.syngress.com/solutions). It contains data about various accounts. We will use this XML document to illustrate our *XPath* queries.

Figure 6.24 Bank 2.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 can 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. 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 preceding 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.*

NOTE

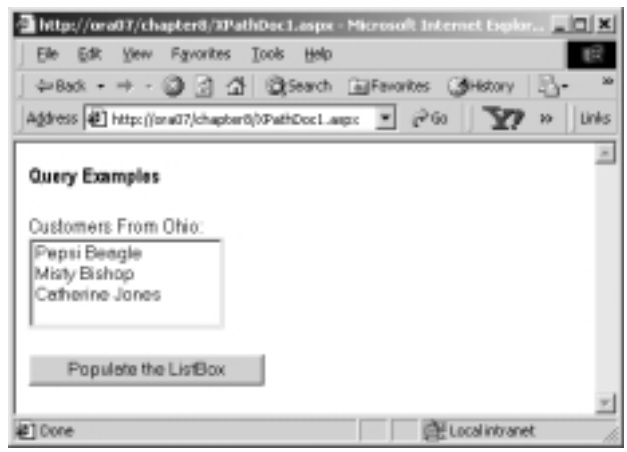
We found the http://staff.develop.com/aarons/bits/xpath-builder/ site to be very good in learning *XPath* queries interactively.

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*.

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 6.24). We will load a list box with the names of customers who are from Ohio. Figure 6.25 shows the output of this application. Figure 6.26 shows the complete code for this application. The code is also available in the XPathDoc1.aspx file on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.25 Using the *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")
```

The **Bank/Account[child::State='OH']/Name** search expression returns the Name nodes from the Account node-set whose state is "OH." To get the

value inside a particular name node, we need to use the *Current.Value* property of the *Iterator* object. Thus, the following code loads our list box:

```
While (myIter.MoveNext())
        lstName.Items.Add(myIter.Current.Value)
End While
```

SYNGRESS
syngress.com

Figure 6.26 Complete Code XPathDoc1.aspx

```
<%@ Page Language="VB" Debug="True"%>
<%@ Import Namespace="System.Xml"%>
<%@ Import Namespace="System.Xml.XPath"%>
<%@ Import Namespace="System.Xml.Xsl"%>
<html><head></head><body>
<form runat="server"><h4>
Query Examples</h4>
Customers From Ohio:<br/>
<asp:ListBox id="lstName1" runat="server"
     width="150" rows="4"/>  
<br/><br/>
<asp:Button id="cmdDetails" Text="Populate the ListBox"
    runat="server"  onClick="showNames"/><br/>
</form></body></html>
<Script Language="vb" runat="server">
 Sub showNames(s As Object, e As EventArgs)
   Dim Doc As New XPathDocument(Server.MapPath("Bank2.xml"))
   Dim myNav As XPathNavigator
   myNav=Doc.CreateNavigator()
   Dim myIter As XPathNodeIterator
   myIter=myNav.Select("Bank/Account[child::State='OH']/Name")
   While (myIter.MoveNext())
     lstName1.Items.Add(myIter.Current.Value)
   End While
  End Sub
</Script>
```

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. Figure 6.27 shows a partial listing of the Bank3.xml. The complete code is available on the companion Solutions Web Site for the book (www.syngress.com/solutions).

Figure 6.27 Bank3.xml

```
<Bank>
   <Account AccountNo="A1112">
      <Name>Pepsi Beagle</Name>
      <Balance>1200.89</Balance>
      <State>OH</State>
   </Account>
   --- --- ---
   --- --- ---
</Bank>
```

The *Account* element of the XML document in Figure 6.27 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. Figure 6.28 shows the output of the application. The application has been developed in an .aspx file named XPathDoc2.aspx. Figure 6.29 shows the complete listing. The code is also available on the companion Solutions Web site for the book (www.syngress.com/solutions).

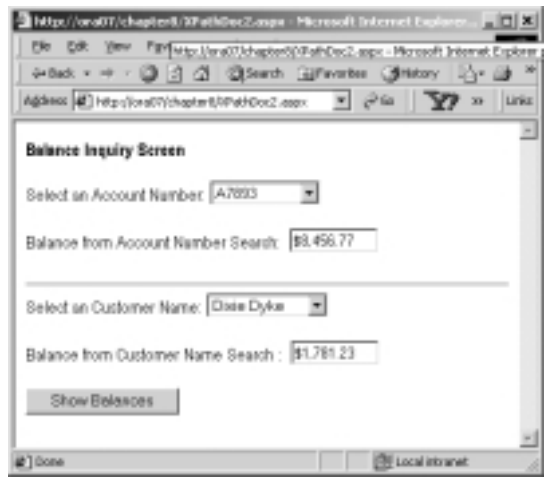
To search for a particular value of an attribute (e.g., of an account number), we have used the following expression:

```
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 6.28 The Output for XPathDoc2.aspx



We need to call the *MoveNext* method of the *Iterator* object in order to get to the balance node. The following expression illustrates the construct:

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

SYNGRESS
syngress.com

Figure 6.29 Complete Code XPathDoc2.aspx

```
<%@ Page Language="VB" Debug="True"%>
<%@ Import Namespace="System.Xml"%>
<%@ Import Namespace="System.Xml.XPath"%>
<%@ Import Namespace="System.Xml.Xsl"%>
<html><head></head><body>
<form runat="server"><h4>
Balance Inquiry Screen</h4>
Select an Account Number:
<asp:DropdownList id="cboAcno" runat="server" width="100" />  
<br/><br/>
Balance from Account Number Search: 
<asp:Textbox id="txtBalance1" runat="server" width="80" />  
<br/><br/><hr/>
Select an Customer Name:
```

Continued

Figure 6.29 Continued

```
<asp:DropdownList id="cboName" runat="server" width="110" />  
<br/><br/>
Balance from Customer Name Search : 
<asp:Textbox id="txtBalance2" runat="server" width="80" />  
<br/><br/>

<asp:Button id="cmdDetails" Text="Show Balances" runat="server"
    onClick="showNames"/><br/>
</form></body></html>
<Script Language="vb" runat="server">
 Sub Page_Load(s As Object, e As EventArgs)
  If Not Page.IsPostBack Then
   Dim myDoc As New XPathDocument(Server.MapPath("Bank3.xml"))
   Dim myNav As XPathNavigator
   myNav=myDoc.CreateNavigator()
   Dim myIter As XPathNodeIterator
   ' Populate the DropDownList with Account Number values
   myIter=myNav.Select("//@*")  ' Load all attributes
   While (myIter.MoveNext())
    cboAcno.Items.Add(myIter.Current.Value)
   End While

   ' Populate the  DropDown list with the name values
   myIter=myNav.Select("/Bank/Account/Name")
   While (myIter.MoveNext())
    cboName.Items.Add(myIter.Current.Value)
   End While
  End If
 End Sub

 Sub showNames(s As Object, e As EventArgs)
   'Get the value of the selected Item
   Dim accNo As String = cboAcno.SelectedItem.Text.Trim()
```

Continued

Figure 6.29 Continued

```
    Dim accName As String = cboName.SelectedItem.Text.Trim()
    Dim myDoc As New XPathDocument(Server.MapPath("Bank3.xml"))
    Dim myNav As XPathNavigator
    myNav=myDoc.CreateNavigator()
    Dim myIter As XpathNodeIterator

   '  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

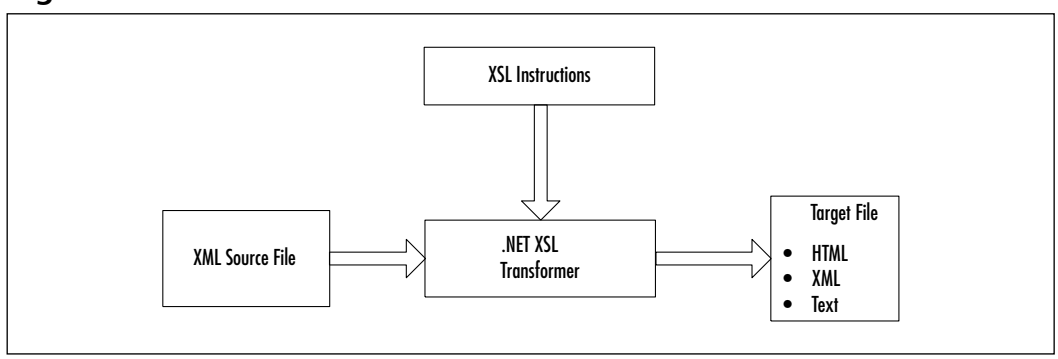
Extensible Stylesheet Language Transformations (XSLT) is the transformation component of the XSL specification by W3C (www.w3.org/Style/XSL). It is essentially a template-based declarative language that can be used to transform an XML document to another XML document, or to documents of other types (e.g., 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 (IE) 5.5 and later 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, IE 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 ascending order of the unit price. Further, we might 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 can either develop relevant codes in a programming language such as C#, or we can 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 kinds 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. Figure 6.30 illustrates this process.

Figure 6.30 XSL Transformation Process

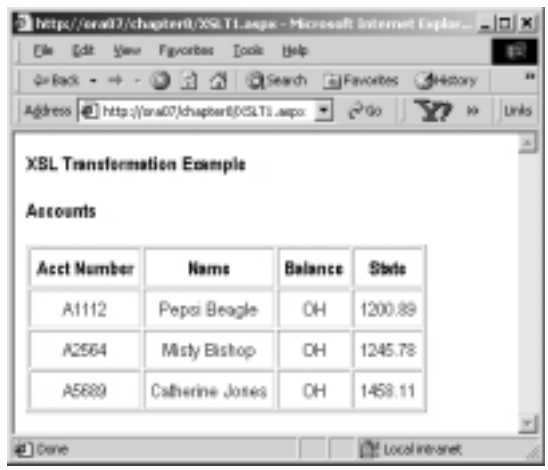


In this section, we will demonstrate certain selected features of XSLT through some examples. The first example will apply XSLT to transform an XML document to an HTML document. We know that IE can automatically transform an XML document to an HTML document and can display it on the screen in collapsible 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 an 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 6.27) document. The extracted data will be finally displayed in an HTML table. Figure 6.31 shows the output of the application.

Figure 6.31 Transforming an XML Document to an HTML Document



If we need to use XSLT, we must first develop the XSLT style sheet (i.e., 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 at (@) 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 6.32, and is also available on the companion Solutions Web site for the book (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 6.33, and is also available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.32 Complete Code for XSLT1.xsl

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0"
        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 6.33 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 </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 6.34, and is also available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.34 An Order Received from a Customer in XML Format (OrderA.xml)

```
<?xml version="1.0" ?>
<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 6.35. The OrderB.xml file is also available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.35 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>
    <Sku>P 25-16:3</Sku>
    <Description>50 GPM Pump</Description>
    <Quantity>10</Quantity>
  </Item>
</Order>
```

The objective of this example is to automatically transform OrderA.xml (Figure 6.34) to OrderB.xml (Figure 6.35). The output of this application is shown in Figures 6.36 and 6.37.

Figure 6.36 Transformation of an XML Document to Another XML Document

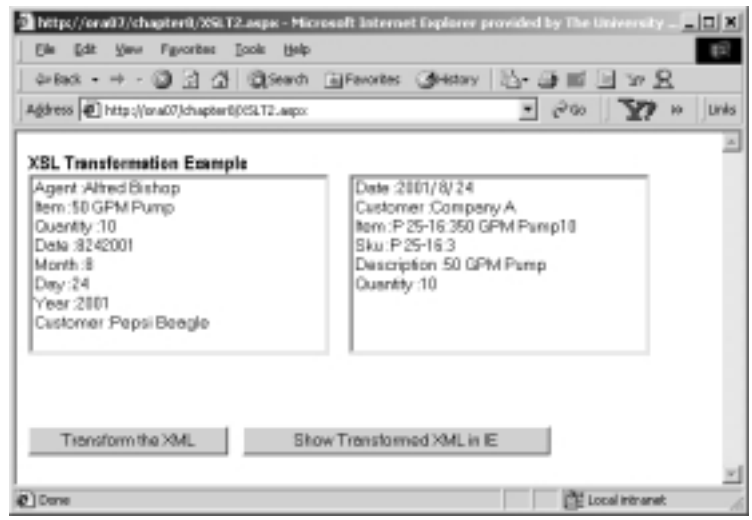
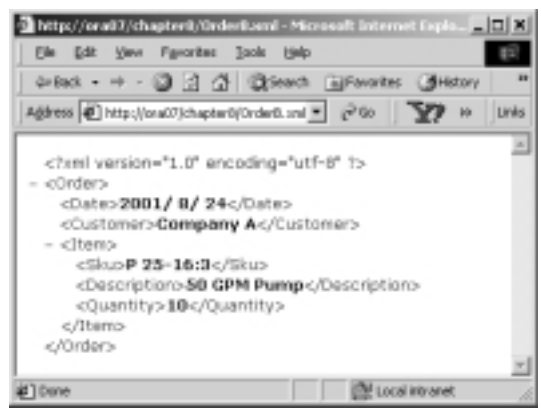


Figure 6.37 The Target XML File as Displayed in Internet Explorer



We have developed an XSLT file (shown in Figure 6.38) 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 6.38, and is also available in the order.xsl file on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 6.38 Complete Listing for order.xsl

```
<?xml version="1.0" ?>
<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>
```

Subsequently, we have developed the XSLT2.aspx file to employ the XSLT code in the order.xsl file to transform the OrderA.xml to OrderB.xml. The complete listing of the .aspx file is shown in Figure 6.39, and is also available on the companion Solutions Web site for the book (www.syngress.com/solutions). The transformation is performed in the *ShowTransformed()* subprocedure of our .aspx file. In this code, the *Transform* method of an *XSLTransform* object is used to transform and generate the target XML file.

Figure 6.39 Complete Listing for XSLT2.aspx

```
<%@ Page Language="VB" Debug="True"%>
<%@ Import Namespace="System.Xml"%>
<%@ Import Namespace="System.Xml.XPath"%>
<%@ Import Namespace="System.Xml.Xsl"%>
<%@Import Namespace="System.IO"%>
<html><head></head><body><form runat="server">
<b>XSL Transformation Example </b><br/>
<asp:ListBox id="lstInitial" runat="server" rows="9"
    width=250/>   
<asp:ListBox id="lstFinal" runat="server" rows="9" width=250/><br/><br/>
<br/><br/>
<asp:Button id="cmdTransform" Text="Transform the XML" runat="server"
       onClick="showTransformed" />  
<asp:Button id="cmdDisplayTgt" Text="Show Transformed XML in IE"
    runat="server" onClick="showTarget" />
</form></body></html>

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

 Sub Page_Load(sender As Object, e As EventArgs)
   If Not Page.IsPostBack Then
    Dim myDoc As New XPathDocument(Server.MapPath("OrderA.xml"))
    Dim myNav As XPath.XPathNavigator
    Dim myIterator As XPath.XPathNodeIterator
    ' Set nav object
```

Continued

Figure 6.39 Continued

```
    myNav = myDoc.CreateNavigator()
    ' Iterate through all the attributes of the descendants
    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"))
 ' Tranform 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
```

Continued

Figure 6.39 Continued

```
  myIterator =myNav.Select("/Order")
  myIterator=myNav.SelectDescendants(XPathNodeType.Element,false)
  myIterator.MoveNext()
  While myIterator.MoveNext()
   ' 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 Online

Databases are used to store and manage an 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. We might 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 pieces of software to exchange data among themselves. In this context, the XML forms a bridge between ADO.NET and other applications. Since XML is integrated into the .NET Framework, the data transfer using XML is much 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, rely heavily on the 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*.

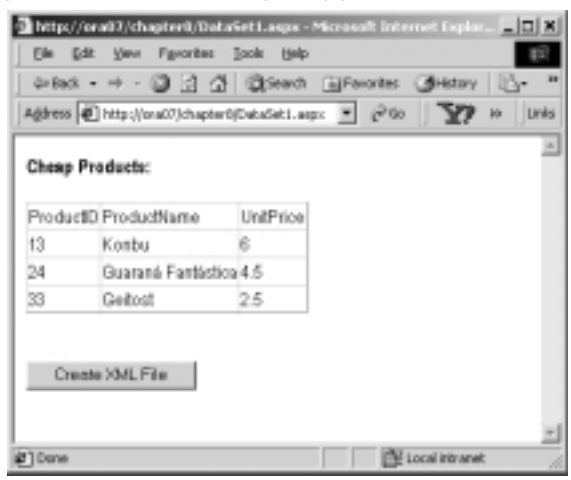
In this context, we will provide two simple examples. In the first example, we will create a *DataSet* from a SQL query, and write its contents as an XML

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 product data of our fishing products to a client. In earlier days, we would have sent the data as a text file. However, in the .NET environment, we can instead develop an XML document very quickly 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 can use any technology (e.g., 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 might 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 6.40). We have developed the application in an .aspx file named DataSet1.aspx. The complete listing of the .aspx file is shown in Figure 6.41 and is also available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 6.40 Output for 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.

SYNGRESS
syngress.com

Figure 6.41 Complete Listing for 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")
```

Continued

Figure 6.41 Continued

```
   myGrid.DataSource=myDataSet.Tables(0)
   myGrid.DataBind
   Session("sessDs")=myDataSet
 End If
End Sub

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 6.40) 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 6.42. The application has been developed in an .aspx file named DataSet2.aspx. The complete code for this application is shown in Figure 6.43, and is also available on the companion Solutions Web site for the book (www.syngress.com/solutions). The code is self-explanatory.

Figure 6.42 Output of DataSet2.aspx Application

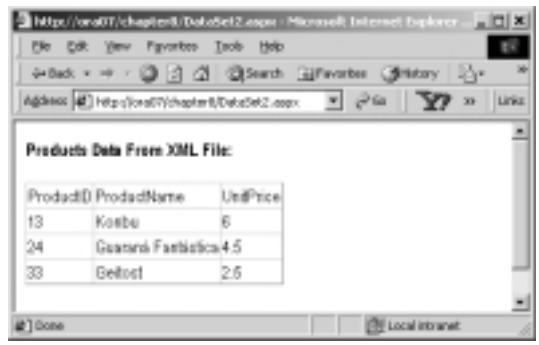


Figure 6.43 Complete Listing for DataSet2.aspx

```
<%@ 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

ASP has come a long way in a very short time. It is not difficult to see why it is so popular, when the languages are so easy to learn and novice developers do not need any special software or platform knowledge, just Notepad and their current desktop operating system. Contrast this against, say, Java Server Pages, where the language can be tricky for new programmers, and the application server installation can seem daunting. The playing field has been leveled; developers now have the freedom to choose the languages that suit them, and each .NET language has equal access to the full .NET functionality and abilities.

Since we can use so many different .NET languages in ASP.NET, we can also use the .NET Framework without any problems.

Solutions Fast Track

Reviewing the Basics of the ASP.NET Platform

- ☑ ASP.NET is part of the wider Microsoft .NET initiative.
- ☑ .NET is a set of tools, services, applications, and servers based on the .NET Framework and Common Language Runtime (CLR).
- ☑ VBScript support has been dropped in favor of VB.NET. The CLR enables you to use a choice of full-fledged, object-oriented, and event-driven server-compiled languages for the first time.
- ☑ .NET languages are compiled using an intermediate language and then into machine-specific code, so language differences are now more a matter of style and personal preference rather than functionality and performance. Objects can interact and inherit from components written in any language.
- ☑ ASP.NET pages are built with (and are) .NET components, providing all the benefits of an object-oriented approach.
- ☑ Web forms introduce a new Visual Basic forms-style way of looking at Web pages, allowing for server-side, event-driven coding and true separation of layout and logic with code behind. .NET form controls maintain session state, and the controls properties are available to the ASP code without resorting to querying the request object.

- ☑ The functionality available has been increased to encompass such exciting features as building dynamic images on-the-fly, browser-based file upload, and network services without the need for third-party components.
- ☑ You can now distribute code and applications easily and effectively with .NET Web services and standards-based protocols.
- ☑ Deployment, including server configuration, is mostly just a matter of transferring files with configuration implemented with Extensible Markup Language (XML) files. Now you do not need to register and unregister components.
- ☑ Mission-critical services now have increased support, with load balancing and several state management options, including the ability to store state information in an SQL Server database.

Reading and Parsing XML

- ☑ 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," the document is written using UTF-8 format.

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.
- ☑ *XmlDocument* class implements both the W3C specifications (Core Levels 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 Online

- ☑ 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** and click on the **"Ask the Author"** form.

Q: Why so much emphasis on the Web? Can't I use XML on the desktop as well?

A: Yes, you can use XML on the desktop. However, one of the main goals of .NET is to properly connect the desktop with the Internet and not suffer any setback due to server type, programming language, and so on. As you might have noticed as well, ASP.NET can be thought of as a Web wrapper for desktop code. This helps ensure that what you see online will be mostly reproducible offline.

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 difference in the specifications, one of the major features was the namespaces in XML being added, which was unavailable in prior versions. 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: 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; *XPath* does. 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 contain *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']**
- *XPath* 1.0 Alternative: **descendant::Account[child::State='OH']**

Which of the preceding alternatives should 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 *XPath* offers more compact search string.

Chapter 7

Creating an XML.NET Guestbook

Solutions in this chapter:

- Functional Design Requirements of the XML.NET Guestbook
- Adding Records to the Guestbook
- Viewing the Guestbook
- Advanced Options for the Guestbook Interface

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

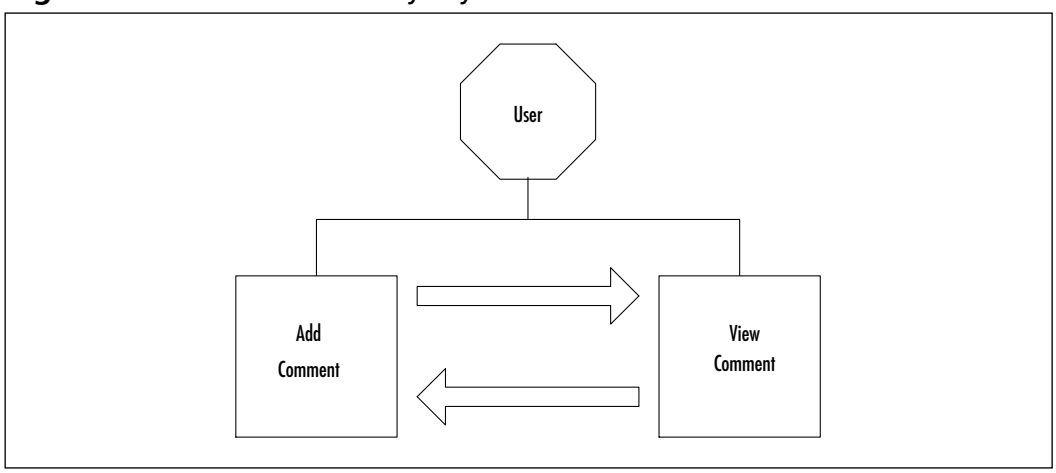
Introduction

Your first case study is a simple online guestbook application, completely coded in ASP.NET. You are going to need to provide the basic functions through this guestbook, namely the ability to do the following:

- Enable guests to enter messages.
- Display all messages on one page.
- Show author, e-mail address of author, and comment from the author of the message.

The flowchart in Figure 7.1 shows the user interaction process that you want to achieve.

Figure 7.1 Basic Functionality Layout



In essence, the user will come to the site and decide if he or she wants to view previous messages or add new ones. The user will be redirected to the view comments page after filling out a new message, or the user viewing the messages has the option to fill out a message.

NOTE

On the companion Solutions Web site for the book (www.syngress.com/solutions) for Chapter 7 are two folders for this chapter, representing two ways of going about this guestbook: one is labeled "basic," and the other is labeled "advanced." We are going to explore both of these.

All of these functions need to be kept as compact as possible. Our backend needs to store the following information for every message that is left on the guest book:

- Name
- E-mail
- Subject Line
- Actual Comment

The Name, E-mail, Subject Line, and Actual Comment need to be required fields, and you need to provide validation for the e-mail field. In addition, you need to provide the user with an easy-to-use interface. A basic interface would consist of the user being able to do the following:

- Choose between adding a new entry and viewing previous entries
- Properly locate the corresponding text areas for the entry points
- Have real-time validation take place where needed
- Reply to a comment left by a user via e-mail

Functional Design Requirements of the XML Guestbook

Several guest books are already available online for download, but most require either a Microsoft Access database or an SQL Server database for storing the guest book entries and other information pertinent to that guestbook. While both of these tools provide their own strengths and weaknesses, you want to provide an application that is small, quick, and able to stand-alone without requiring a separate application to make it work. This type of thought also implies that the application will be small and easy to transfer, if needed. You also need to keep an eye on the code and keep it as small as possible. You need to be able to write directly to the database and read from the database with as little code as possible. Just because you are trying to make the code portable doesn't mean you need to make the code bloat!

So, if you are not going to use a traditional database (such as Microsoft Access or SQL Server), then what can you use that won't kill the application

requirements? Previously, we talked about a technology that is turning into a strong database alternative, called XML. XML will enable you to use a text-based approach to your database that does not rely on any ODBC connections or even any server (although your code will pretty much lock you into a server that uses ASP.NET). Moreover, through an XML schema you can define how your XML "row" will look and what each value must contain.

With your backend solution set at XML, you need to determine how you are going to work with the XML file. The logical choice is the *System.XML* namespace, but you can actually find a faster method by using the XML tools that accompany the *System.Data* namespace. Even though *System.XML* is the more powerful than *System.Data* when it comes to XML, you simply don't need to rely on so much coding to see your results.

NOTE

The choice of *System.Data* over *System.XML* does *not* mean that *System.XML* is in any way inefficient. It simply means that, as programmers, we sometimes have to choose between a solution that requires more time but is more flexible, and a solution that is quicker but more rigid. *System.XML* is more flexible with XML than *System.Data* will ever be, but all you need for this case study is just to be able to read and write to an XML file.

Constructing the XML

Even though *System.Data* is viewed more or less as a method of working with traditional database connections, such as a SQL database or an Access database, it can also work with XML data, provided the XML has an inline schema that it can match the data against; almost like looking at the table structure first and then the data within it.

The file gbook.xml shown in Figure 7.2, and in the Basic directory on the companion Solutions Web site for the book (www.syngress.com/solutions) displays the XML code with which we will be working.

Figure 7.2 gbook.xml (Basic Version)

```
01: <gbook>
02: <xsd:schema id="gbook"
03:              targetNamespace=""
04:              xmlns=""
05:              xmlns:xsd="http://www.w3.org/2001/XMLSchema"
06:              xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
07:
08: <xsd:element name="gbook"
09:               msdata:IsDataSet="true">
10:      <xsd:complexType>
11:     <xsd:choice maxOccurs="unbounded">
12:     <xsd:element name="gbooky">
13:     <xsd:complexType>
14:     <xsd:sequence>
15:          <xsd:element name="Name" type="xsd:string" minOccurs="0" />
16:      <xsd:element name="Chrono" type="xsd:string" minOccurs="0" />
17:      <xsd:element name="Email" type="xsd:string" minOccurs="0" />
18:      <xsd:element name="Comments" type="xsd:string" minOccurs="0" />
19:   </xsd:sequence>
20:   </xsd:complexType>
21:   </xsd:element>
22:   </xsd:choice>
23:   </xsd:complexType>
24: </xsd:element>
25:
26: </xsd:schema>
27: </gbook>
```

Lines 1 and 26 have the root tags for the XML file. In this example, we are using "gbook," but you can use anything. Lines 2 through 6 are one line in which we used whitespace to organize the attributes in order for the tag to be more

readable. The *targetNamespace* and *xmlns* attributes in the *<xsd:schema>* tag are left blank, since both the *targetNamespace* and *xmlns* are inline. The *xsd* attribute is pointing to the current schema, and the special Microsoft attribute *msdata* points to a Microsoft data compatibility namespace.

NOTE

If you want more information on the XSD and MSDATA attributes, you can find documentation for XML schemas online at http://msdn.microsoft.com/library and www.w3.org/XML/Schema.html.

Lines 8 through 24 construct the element that will store the data. When the data is entered into the corresponding .aspx file, it will format the data within the XML per the data outline within the schema. In this case, a sample entry in our guestbook will appear as the following:

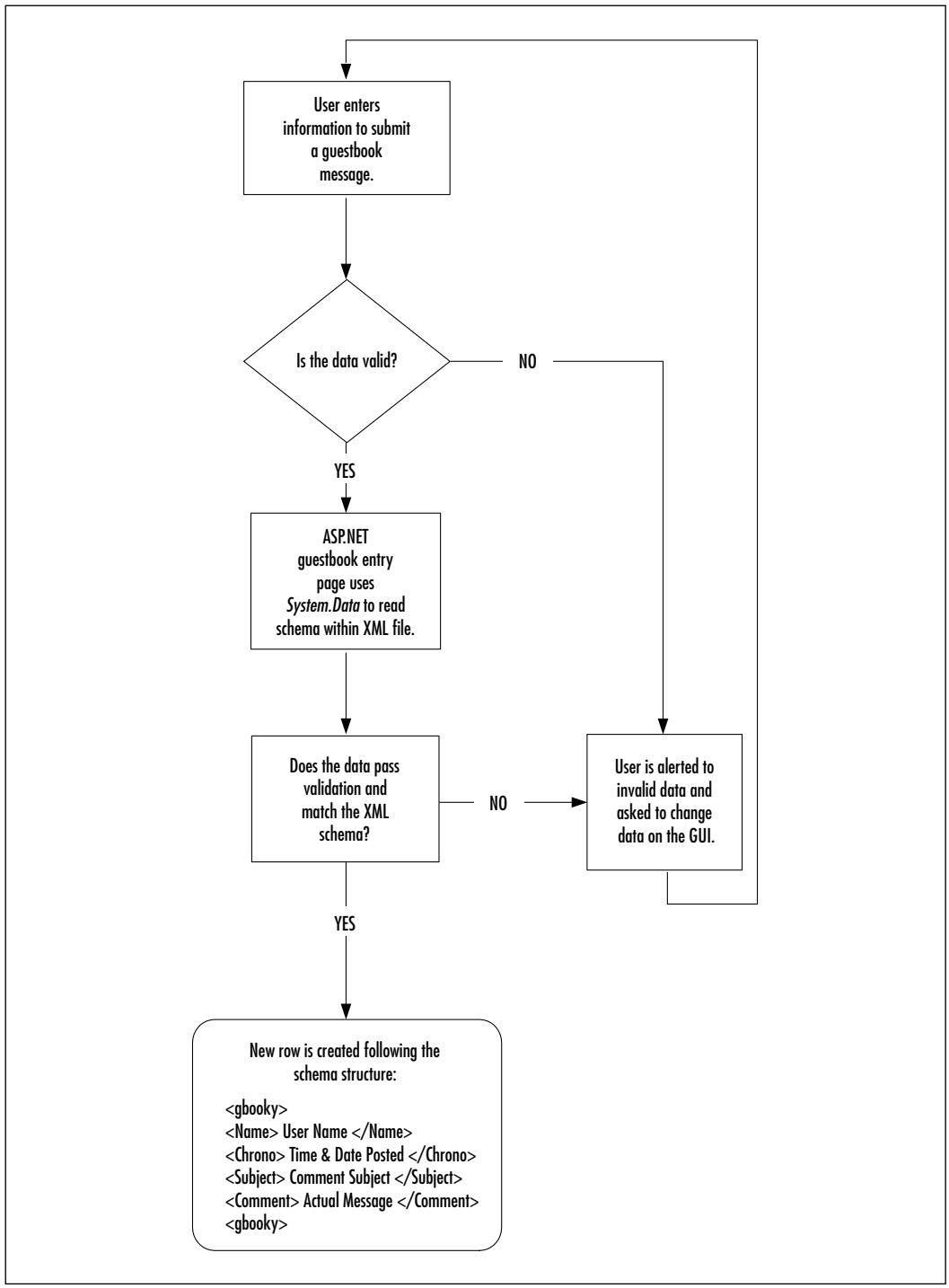
```
<gbooky>
    <Name>Jon Ortiz</Name>
    <Chrono>Time Posted</Chrono>
    <Email>somewhere@overthereainbow.com</Email>
    <Comments>Hola!</Comments>
</gbooky>
```

This information will be created by your application through the *System.Data* namespace. In order to be able to do so, *System.Data* matches the information input to the inline schema and creates the appropriate record. Now that you have set up the "template," you can get started with the code that adds records. Refer to Figure 7.3 for the logic behind the XML file.

Adding Records to the Guestbook

Any veteran ASP developers are going to notice in this section a distinct change. Remember in desktop applications that you formed your GUIs using a form? Well, in ASP.NET, the form has been brought to Web development and is referred to as a *panel*. You are going to work with your code inline for just this chapter so that you can get a good grasp of what a panel looks like and how it works within ASP.NET.

Figure 7.3 Creating a Record Using the XML Schema



There are no real differences between using a form for desktop applications and a panel for online applications. Many of the same subs are intact, such as *OnLoad*, and *panel* can reference any item within the panel, just like in desktops. A great place to view the panel in ASP.NET is within the UI for adding guestbook records. Your file will be called add.aspx, and is the code is shown in Figure 7.4 (note that some lines wrap), and in the Basic directory on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 7.4 Sample ASPX Code add.aspx (Basic Version)

```
01: <%@ Page Language="VB" EnableSessionState="False"%>
02:
03: <%@ Import Namespace="System.IO" %>
04: <%@ Import Namespace="System.Data" %>
05: <html>
06: <head>
07: <title>Add Entry</title>
08: </head>
09: <script language="VB" runat="server" >
10: <!-- event handling code here-->
11: </script>
12: </head>
13: <body topmargin="0" leftmargin="0" rightmargin="0" marginwidth="0"
    marginheight="0">
14: <br>
15: <br>
16: <h3 align="center">Guestbook Post Page.</h3>
17: <br>
18: <asp:label id="err" text="" style="color:#FF0000" runat="server" />
19: <asp:Panel id=pnlAdd runat=server>
20: <form action="add.aspx" runat=server>
21:     <table border="0"  width="80%" align="Center">
22:     <tr>
23:     <td><b>Sign-in My GuestBook</b></td>
24:     <td> </td>
25:     </tr>
```

Continued

Figure 7.4 Continued

```
26:     <tr>
27:     <td>Name :</td>
28:     <td><asp:textbox text="" id="Name" runat="server" /
    ><asp:RequiredFieldValidator ControlToValidate=Name display=static
    runat=server>*</asp:RequiredFieldValidator></td>
29:     </tr>
30:     <tr>
31:     <td>E-Mail :</td>
32:     <td><asp:textbox text="" id="Email" runat="server"/>
    <asp:RequiredFieldValidator ControlToValidate=Email display=static
             runat=server> *</asp:RequiredFieldValidator>
    <asp:RegularExpressionValidator runat="server"
             ControlToValidate="Email"
             ValidationExpression="[\w-]+@([\w-]+\.)+[\w-]+"
             Display="Static"
             Font-Name="verdana" Font-Size="10pt">Please enter a valid
             e-mail address</asp:RegularExpressionValidator>
33:     </td>
34:     </tr>
35:     <tr>
36:     <td>Comments :</td>
37:     <td><asp:Textbox textmode=multiline id="Comments" columns="25"
        rows="4" runat="server" />
38:     </td>
39:     </tr>
40:     <tr>
41:     <td colspan="2" >
42:     <asp:Button Text="Submit Post" onClick="AddClick" runat="server"
    /></td>
43:     </tr>
44: </table>
```

Continued

Figure 7.4 Continued

```
45: </form>
46: </asp:Panel>
47:
48: <asp:Panel id=pnlThank visible=false runat=server>
49: <p align=center><b>Thank you for posting in my Guestbook!</b><br>
50: <a href="viewbook.aspx">Click here </a> to view GuestBook.
51: </p>
52: </asp:Panel>
53: </body>
54: </html>
```

It might look daunting at first, but it really is quite simple. Remember that in ASP.NET, you first should declare the language the page is going to be using. While it is redundant, since the language declaration on the <*script*> tag determines the actual language use, it is still a good coding practice to get into. Lines 2 through 4 declare the namespaces that you are going to use—*System*, *System.IO*, and *System.Data*. Lines 5 through 8 just display the HTML code that needs to be in every HTML page.

You then hit the script tag that controls the Submit button event (lines 9 through 10). For now, it's just a placeholder for the code you'll be adding in later. Notice that the code is placed at the head of the HTML file, which means that it will be processed before anything else. You'll look at the Submit button event after you dissect this portion of the ASP.NET page.

Understanding the *pnlAdd* Panel

On line 19 of Figure 7.4, *pnlAdd* is declared; it is the name of the panel that contains the programming code displaying the messages and text boxes that the user will be viewing on the page, in order to enter the guestbook entry data; (e.g., the name area, the name entry textbox, the e-mail area, the e-mail entry textbox, the comment area, the comment entry textbox, and the Submit button). In other words, it is your run-of-the-mill HTML form but with ASPX. In reality, there are only two "normal" form objects; the name text box is your standard text object, and the comment area is your standard multi-line text box.

The e-mail area, however, is another story. Take a look at the behemoth of a line that you'll find in line 32:

```
<asp:textbox text="" id="Email" runat="server"/
    ><asp:RequiredFieldValidator ControlToValidate=Email display=static
    runat=server> *</asp:RequiredFieldValidator>
    <asp:RegularExpressionValidator runat="server"
            ControlToValidate="Email"
            ValidationExpression="[\w-]+@([\w-]+\.)+[\w-]+"
            Display="Static"
            Font-Name="verdana" Font-Size="10pt">Please enter a valid
    e-mail address</asp:RegularExpressionValidator>
```

Starting from the top, you find your standard *ASPcontrol* declaration as a text box with its default text set to empty and an ID of "E-mail." Right after it comes the ASP control declaration for *RequiredFieldValidator* set to validate the control labeled *E-mail* and with a static display. You then implement two types of validation to the field. The first validation is through the *RegularFieldValidator* control:

```
<asp:RequiredFieldValidator ControlToValidate=Email display=static
    runat=server>This is required.</asp:RequiredFieldValidator>
```

All you are doing here is a quick check to see if the field is empty. If the user skips the field and leaves it empty, then a little message in red shows up saying that "This is required." You don't have to use that text, but it works for this example. Our second round of validation begins right after that line with the more intense *RegularExpressionValidator* object:

```
<asp:RegularExpressionValidator runat="server"
            ControlToValidate="Email"
            ValidationExpression="[\w-]+@([\w-]+\.)+[\w-]+"
            Display="Static"
            Font-Name="verdana" Font-Size="10pt">Please enter a valid
    e-mail address</asp:RegularExpressionValidator>
```

You first set the object to bind itself to the *Email* control. It will be analyzing the contents within the e-mail object to see if it falls under the validation expression that it has been given; in this case, it checks to see that an at (@) symbol as well as a dot (.) is present within the string. You might want to read up on *RegEx* to fully understand what variables can be used with regular expressions.

Developing & Deploying…

Stricter E-Mail Validation

The method of e-mail validation demonstrated in this chapter is not the only option available to you. There is a stricter method for e-mail validation that would only enable the user to input a .com, .org, .edu, .mil, .gov, or .net:

```
ValidationExpression = "^[\w-]+@[\w-]+\.(com|net|org|edu|mil|gov)$"
```

Adding a Thank-You Panel with *PnlThank*

All you are doing here is declaring a panel that will show up after a successful guestbook entry has been added to the XML file. The link in order to view the guestbook is declared and set. Very simple and very quick, to the point, starting on line 48 (Figure 7.4):

```
<asp:Panel id=pnlThank visible=false runat=server>
<p align=center><b>Thank you for posting in my Guestbook!</b><br>
<a href="viewbook.aspx">Click here </a> to view GuestBook.
</p>
</asp:Panel>
```

Exploring the Submit Button Handler Code

Now that you have established your design and layout, you can take a look at the code that actually handles the addition of new entries into the guestbook. The basic functionality of this code is to react to the Submit button when pressed, and write the necessary items to the XML file. Figure 7.5 walks you through an overview of the Submit button code. The complete source code for Figure 7.5 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 7.5 Submit Button Handler Code for add.aspx (Basic Version)

```
01:     Sub AddClick(Sender As Object, E As EventArgs)
02:
03:          Try
04:          Dim dataFile as String = "gb/gbook.xml"
05:
06:          'the next line wraps
07:          Dim fin as New FileStream (Server.MapPath(dataFile),
    FileMode.Open,FileAccess.Read,FileShare.ReadWrite)
08:
09:          'this line also wraps
10:           Dim fout as New FileStream (Server.MapPath(dataFile),
    FileMode.Open,FileAccess.Write,FileShare.ReadWrite)
11:
12:          Dim guestData as New DataSet()
13:          Dim newRow as DataRow
14:          err.Text = ""
15:          guestData.ReadXml(fin)
16:          fin.Close()
17:          newRow = guestData.Tables(0).NewRow()
18:            newRow("Name")=Name.Text
19:          newRow("Chrono")=DateTime.Now.ToString()
20:          newRow("Email")=Email.Text
21:          newRow("Comments")=Comments.Text
22:          guestData.Tables(0).Rows.Add(newRow)
23:          guestData.WriteXml(fout, XmlWriteMode.WriteSchema)
24:          fout.Close()
25:          pnlAdd.Visible=false
26:          pnlThank.Visible=true
27:
28:          Catch edd As Exception
29:          err.Text="Error writing file at: " & edd.ToString()
30:
31:          End Try
32:
33: End Sub
34: </script>
```

Migrating…

Online Forms

As you have noticed and learned throughout this book, ASP.NET enables programmers to use Web forms, which can be described as the VB6.0 desktop form. In this particular example, your *AddClick* sub would be placed within the *OnClick()* event for whatever button you wanted to use as your trigger for this action. One other little trick is to view each "panel" as a small form within the browser window, with their own "hide" and "show" features.

Line 1 starts you off with your VB code, declaring itself a code segment that is run on the server-side and written using VB. Line 1 uses an ASP.NET form subnamed *AddClick*; this code segment will be providing all of the functionality of the Submit button.

On line 3, you start taking advantage of one of VB's newest and very useful features, error trapping. Your try/catch segment starts out by declaring a variable to store the location of your XML file, which can be any directory. You can just assume that for this example it's in the *gb* directory on the root folder of the site. With the location of the file stored, you can open up a *FileStream* object to open and process the XML file for you. *FileStream* needs to know the actual location (not the virtual location) of the file, so you use *Server.MapPath()* to return the actual location of the file to your *FileStream* object, which you can then open (*FileMode.Open*) and start reading (*FileAccess.Read*). You can also tell *FileStream* how to handle other events, such as sharing; by telling *FileStream* to allow read/write sharing of the file (*FileShare.ReadWrite*), you don't have to worry about your XML file suffering from any file locking, which would prevent any other user from editing the file and giving the user a nasty error.

With your XML file stored within the *fout* object (line 10 in Figure 7.5), you can start to create the object that will handle parsing the data, *DataType,* and properly formatting it and writing to the XML file, *DataRow.* Specifically, *DataType* will handle reading the information and transforming it into a table format. *DataRow* will then use the information stored within your *DataType* object to create a new row with the columns that it finds within the *DataType* object. In other words, when *DataType* reads your XML file, it will see the root element *gbook* as your table, *gbooky* as your rows, and all the information within *gbooky* as columns. It will write the information out accordingly to the XML file. It will know what it's

writing since it's using the inline schema (Figure 7.2) to write to the file per the schema, using the *WriteXML* class of the *DataType* object and having it write the stream matching the *XMLSchema* (*XMLWriteMode.WriteSchema*). You then hide the panel that contains the text boxes and Submit button, and make the panel that contains the "Thank You" message. Figures 7.6 and 7.7 show the basic add.aspx file before and after filling out a new entry.

Figure 7.6 Before Adding a New Entry

Developing & Deploying…

File Locking

File locking is a basic response to multiple users trying to read and modify the same file at around the same time. We say at "around" the same time because file locking will take place if the file is accessed at the same time, or if access is attempted after someone already has access to it. By preventing multiple users from reading and writing the file, you avoid file corruption and constant backup restorations. File locking allows a temporary "lock" to be placed to the file that allows for changes to be made one after the other without damaging the integrity of the file.

Figure 7.7 After Adding a New Entry

Viewing the Guestbook

One line of actual ASPX code—that's about as simple as it gets, and is done just by using the built-in XML server control. You should know that ASP.NET has several controls built in to facilitate many different HTML functions, such as displaying radio buttons and handling forms, which allows ASP.NET to generate items fairly on-the-fly. XML is no exception to this rule.

Displaying Messages

Here is our one-line masterpiece, as shown in Figure 7.8. In essence, all we did to get the sample output shown in Figure 7.8 was just to tell the ASP.NET XML control to read the data in gbook.xml, and to transform it according to the XSL information in gbook.xsl. It is shown in Figure 7.9 and can be found in the *gb* folder in the Basic directory on the companion Solutions Web site for the book (www.syngress.com/solutions). Figure 7.10 shows us the output.

Figure 7.8 viewplain.aspx (Basic Directory)

```
01: <html>
02: <head>
03: <title>XML Control Test</title>
04: </head>
05: <body bgcolor="#000000">
06: <!- - line 7 wraps - ->
07: <asp:xml id="gbook" DocumentSource="gb/gbook.xml"
    TransformSource="gb/gbook.xsl" runat="server"/>
08: </body>
09: </html>
```

NOTE

If you have no other recourse but to use XSL to also generate your hyperlinks, the fastest workaround to this is to simply add the *<a>* element with an attribute of *href* and nesting the *e-mail* element.

Figure 7.9 gbook.xsl

```
01: <?xml version="1.0"?>
02:
03: <!-- this line wraps -->
04: <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    version="1.0">
05:
06: <xsl:template match="/">
07:
08: <xsl:for-each select="gbook/gbooky">
09: <table width = "400">
10: <!- - this line wraps - ->
11: <tr><font color="#FFFFFF" face="Arial Black"><xsl:value-of
    select="Name"/></font></tr>
```

Continued

Figure 7.9 Continued

```
12:
13: <!- - this line wraps - ->
14: <tr><font color="#FFFFFF" face="Arial Black"><br /><xsl:value-of
    select="Chrono"/></font></tr>
15:
16: <!- - this line wraps - ->
17: <tr><font color="#FFFFFF" face="Arial Black"><br /><xsl:value-of
    select="Email"/></font></tr>
18:
19: <!- - this line wraps - ->
20: <tr><font face="Arial, Helvetica, sans-serif" size="2"
    color="#C7B29A"><p><xsl:value-of
    select="Comments"/></p><p></p></font></tr>
21:
22: </xsl:for-each>
23: </table>
24: </xsl:template>
25: </xsl:stylesheet>
```

Figure 7.10 Viewing Basic Guestbook Entries

Advanced Options for the Guestbook Interface

Now that you have a good understanding of a guestbook and how it works, you can try to do something you haven't done yet—actually make it look cool! Just because you are working with ASP.NET does not mean that you cannot use its new tricks to come up with some really jazzy items and tweak your XML a bit. Let's start by looking at your guestbook entry page.

Manipulating Colors and Images

Clearly, this is a design point and not a very strong showing of ASP.NET. However, how you design your page is just as vital as how you design a graphical user interface. In this example, we made the design pleasing to the eye, and we try to use a couple of design techniques to lure the user's eye to the proper areas on the add screen. While these are basic points, it's a good idea to keep the following in mind:

- Is the area visible on most monitors? (Start at 800 x 600 resolution.)
- Will the user be able to understand what to do?
- If the user cannot easily figure out what to do, should an easy-to-find help link be visible, or should you perhaps change the design?

One of the nice things about asp.net controls is that you can still use tags with them. In fact, this second version of the add entry page (Figure 7.12) looks so nice because we're using a Cascading Style Sheet (CSS) script with it (on the companion Solutions Web site for the book [www.syngress.com/solutions] as gbook.css in the Advanced directory). Another part of this new design that you haven't seen before are the emoticons. Emoticons add a little bit of interactivity to the guestbook by enabling users to pick an image that reflects their "feelings" at the time of posting. You will have to add a couple of changes to the XML file and to the add.aspx file, as well as to the view.aspx file in order to display the images. Figure 7.11 shows you how the new add.aspx page will look before entering a message, and Figure 7.12 shows the page after entering a message.

Figure 7.11 add.aspx before Entering a Comment (Advanced Version)

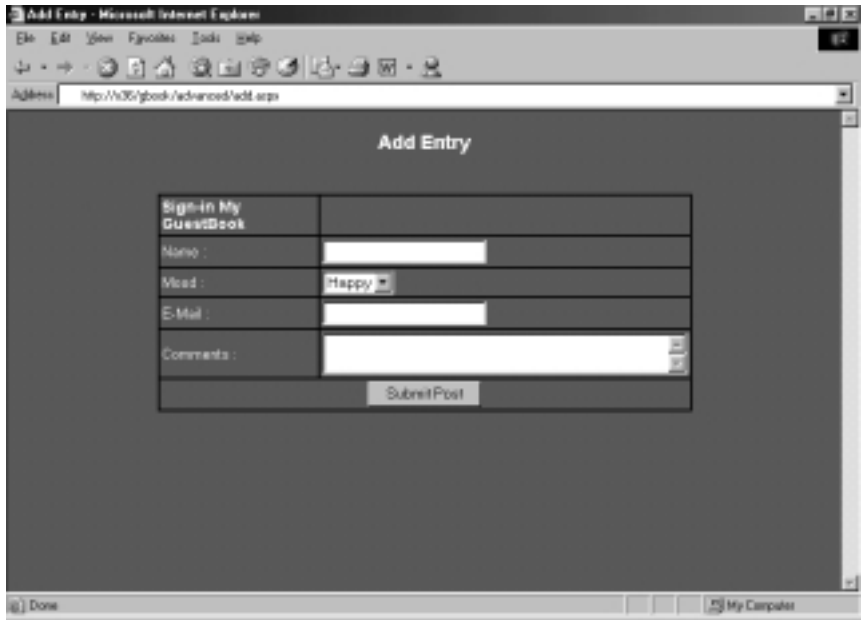
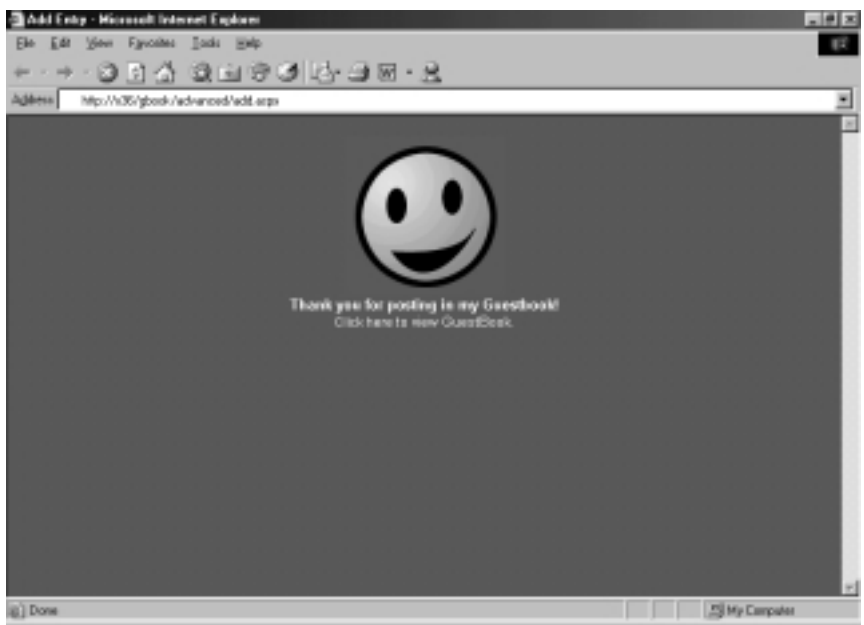


Figure 7.12 add.aspx after Entering a Comment (Advanced Version)



Line 16 in Figure 7.13 reflects the change from the previous XML code; all that happened was just to create a new element of *img* underneath the

complex type *gbook*. Your code will read this value and assign the correct image for it. For right now, all you are doing is just preparing the inline schema to support the value so that when you store the data it will know where to put it. The complete source code for Figure 7.13 is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 7.13 gbook.xml (Advanced Version)

```
01: <gbook>
. . .
14:     <xsd:sequence>
15:          <xsd:element name="Name" type="xsd:string" minOccurs="0" />
16:      <xsd:element name="Emoticon" type="xsd:string" minOccurs="0" />
17:      <xsd:element name="Email" type="xsd:string" minOccurs="0" />
18:      <xsd:element name="Comments" type="xsd:string" minOccurs="0" />
19:      <xsd:element name="DateTime" type="xsd:string" minOccurs="0" />
20:   </xsd:sequence>
```

Now for your code; first, you have to add the new row to your Submit button handler at the top in order to include the new *Emoticon* element within the XML (Figure 7.14). The advanced version of the add.aspx submit handler code is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 7.14 Your Changed add.aspx Submit Handler Code (Advanced Version)

```
10:     Sub AddClick(Sender As Object, E As EventArgs)
11:
12:          Try
13:          Dim dataFile as String = "gb/gbook.xml"
14:
15:          'the next line wraps
16:          Dim fin as New FileStream (Server.MapPath(dataFile),
    FileMode.Open,FileAccess.Read,FileShare.ReadWrite)
```

Continued

Figure 7.14 Continued

```
17:
18:            'this line also wraps
19:            Dim fout as New FileStream (Server.MapPath(dataFile),
    FileMode.Open,FileAccess.Write,FileShare.ReadWrite)
20:
21:            Dim guestData as New DataSet()
22:            Dim newRow as DataRow
23:            err.Text = ""

24:            guestData.ReadXml(fin)
25:            fin.Close()
26:            newRow = guestData.Tables(0).NewRow()
27:            newRow("Name")=Name.Text
28:            newRow("Emoticon")=Emoticon.Value
29:            newRow("Chrono")=DateTime.Now.ToString()
30:            newRow("Email")=Email.Text
31:            newRow("Comments")=Comments.Text
32:            guestData.Tables(0).Rows.Add(newRow)
33:            guestData.WriteXml(fout, XmlWriteMode.WriteSchema)
34:            fout.Close()
35:            formPanel.Visible=false
36:            thankPanel.Visible=true
37:
38:            Catch edd As Exception
39:            err.Text="Error writing file at: " & edd.ToString()
40:
41:            End Try
42:
43: End Sub
44: </script>
```

The final change to your add entry is an option button for the image selection; you can add this code anywhere in the add.aspx within the display area. We set ours right after the name.

```
<tr>
<td>Mood :</td>
<td><select id="Emoticon" runat="server">
<option Value="01.gif">Happy</option>
<option Value="02.gif">Sad</option>
<option Value="03.gif">Cute</option>
<option Value="04.gif">Ugly</option>
</select>
</td>
</tr>
```

Modifying the Page Output

You don't really want to display the same boring, old structured output, so try using some tables to break things up a bit. You are going to take a look at this code a bit differently by starting with the page load code (Figure 7.15). The source code for the advanced version of the view.apsx code is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 7.15 view.aspx (Advanced Version)

```
Sub Page_Load(Src As Object, E As EventArgs)
 Dim ds As New DataSet
 Dim fs As New FileStream(Server.MapPath("gb\gbook.xml"),
FileMode.Open)
 ds.ReadXml(fs)
 gbook.DataSource = ds.Tables(0).DefaultView
 gbook.DataBind()
 fs.close()
End Sub
```

You are telling the server that when the page loads (before *anything* else is processed, including HTML), create a dataset (*ds*) and a filestream (*fs*) to the XML file. Then, you tell the dataset (*ds*) to read the XML file and bind the information to the *gbook* object with the information contained in the dataset. You close the *filestream* and finish your initialization code. Your display code has undergone some major changes as well (see Figure 7.16; note that some lines wrap). The complete source code is available on the companion Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 7.16 Your Changed Display Code add.aspx (Advanced Version)

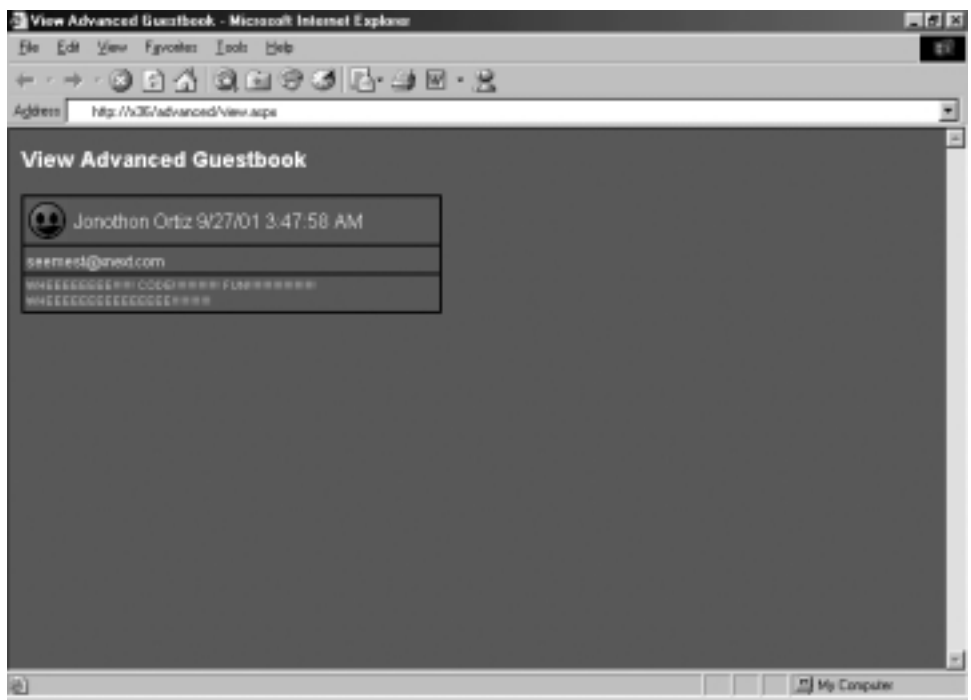
```
01: <%@ Page Language = "VB" Debug="true" %>
02: <%@ Import Namespace="System.IO" %>
03: <%@ Import Namespace="System.Data" %>
04: <html>
05: <script language="VB" runat="server">
      . . .
06: </script>
07:
08: <body>
09: <h3>Advanced Guestbook</h3>
10: <ASP:Repeater id="gbook" runat="server">
11: <headertemplate>
12: <table width="350" style="font: 12pt Arial">
13: </headertemplate>
14:
15: <itemtemplate>
15:   <tr>
16:     <%# DataBinder.Eval(Container.DataItem, "Name") %>
17:     <img src="<%# DataBinder.Eval(Container.DataItem, "Emoticon")
    %>" >
18:     <%# DataBinder.Eval(Container.DataItem, "Chrono") %>
19:   </tr>
20:   <tr>
21:     <a href="mailto: <%# DataBinder.Eval(Container.DataItem,
    "Email") %>"><%# DataBinder.Eval(Container.DataItem, "Email") %></a>
22:   </tr>
23:   <tr>
24:     <%# DataBinder.Eval(Container.DataItem, "Comments") %>
25: </tr>
26: </itemtemplate
27:
28: <footertemplate>
29: </table>
30: </footertemplate>
31:
32: </ASP:Repeater>
33: </body>
34: </html>
```

NOTE

This code has had all of the graphical changes stripped; if you want to see the code as the screenshots display it, please check the code on companion Solutions Web site for the book (www.syngress.com/solutions).

Instead of using the *asp:xml* server control, you are using the *Repeater* control and a *DataSource*. Lines 2 and 3 have the two namespaces that you are going to need for your script tag. *System.IO* handles the *Filestream* object, and *System.Data* handles the *DataSource* object. The information acquired from the *Page_Load* sub will generate the information that is bound to the *Repeater* object. The *Repeater* object (*id="gbook"*) will read the information bound to it, write the header, and then repeat the sequence within the item template until it finishes; then the footer will be written and the *asp:repeater* object will close. Line 17 shows your only change to the *Repeater* by adding the link to the image stored by the image tag. The preceding code plus the graphical add-ons give you the happy result as seen in Figure 7.17.

Figure 7.17 view.aspx + graphics (Advanced Version)



Summary

Well, we started with basically nothing and finished with something that is not only useful, but can be pleasing to the eye as well. Hopefully, this chapter introduced some concepts that are useful, not only to your hobby programming but also in your work.

XML and ASP.NET can work well together in a variety of ways: from simple reading and writing, to proper design and look. Using a combination of either the *System.Data* namespace and the ASP server objects, you can create a single-line parsing .aspx page or a more robust page with tables, rows, columns, and different colors and graphics. In order to achieve the best performance available, the *System.Data* namespace requires an inline schema within the XML file, which the *System.Data* namespace can reference against when reading or writing XML.

ASP server objects themselves are very flexible in that they can be standalone and provide an area to insert inline ASPX code. In the Advanced guestbook, you made heavy use of the inline functions, wrapping table rows and columns around them to provide a view that was readable. In addition, by using an inline function you were able to receive the correct image file associated for an emoticon, by placing it within the <*image*> html tag. Combined with Cascading Style Sheets (CSS), this method proved capable and provided ample room to grow.

Solutions Fast Track

Functional Design Requirements of the XML.NET Guestbook

- ☑ XML enables you to use an interface that is both universally read and universally accessed. You do not have to use bulky components such as SQL or Access databases for simple—and even some complicated—database solutions.
- ☑ XML provides a schema to use with XML in order to provide validation for data.

Adding Records to the Guestbook

- ☑ When working with the *System.Data* namespace and planning to write XML, you need to make sure that you have a properly validated inline XML schema, or else the code will not work.

- ☑ Even though you can use the XML schema to help determine certain validation points, it is better to have the ASP.NET provide the validation of certain entries, such as e-mail, due to the powerful use of regular expressions.

Viewing the Guestbook

- ☑ Using *System.Data* can provide a fast, efficient forward-only read and write solution that is perfect for reading and writing to XML files that are not dependant on heavy node interaction, and that just need information added to them.
- ☑ Cascading Style Sheets (CSS) provide a way to create a more pleasing guestbook without having to change any code structure.

Advanced Options for the Guestbook Interface

- ☑ The ASP.NET controls are very versatile and efficient. Keep in mind that by combining them with CSS, their obvious lack of visual aids is easily bypassed for a true eye-candy feel.
- ☑ The *asp:repeater* object needs to have a *<headertemplate>*, an *<itemtemplate>*, and a *<footertemplate>* within it to function.
- ☑ The only part of the *asp:repeater* object that actually repeats is the *<itemtemplate>* section.

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** and click on the **"Ask the Author"** form.

Q: Why does the *add.aspx* code need the inline XML schema?

A: *Add.aspx* uses the schema to retrieve the way it needs to write the data to the XML file in the proper order. Say that instead of *name* before *e-mail*, you had *e-mail* before *name*; *add.aspx* would write the row with the *e-mail* field first instead of the *name* field.

Q: Why won't the simple guestbook show?

A: .NET expects *www.w3.org/1999/XSL/Transform* as the XSLT namespace. This does limit you a bit, since the working draft version is much better than the 1999 version.

Q: I get an error that says, "compilation error, (*addClick* or *Page_Load*) is not part of asp:(add.aspx or viewbook.aspx)". What does that mean?

A: Unfortunately, some of the error handling for ASP.NET still needs tweaking; this is a perfect example. When running the aspx page, it will spit out errors when it finds them within the ASP objects, but is not very good at reporting errors within the subs located within the <*head*> tag. When you see these errors, check the code and try again.

Chapter 8

Creating a Message Board with ADO and XML

Solutions in this chapter:

- Setting Up the Database
- Designing Your Application
- Designing the User Interface
- Setting Up General Functions
- Building the Log-In Interface
- Designing the Browsing Interface
- Creating the User Functions
- Building the Administrative Interface

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

This case study details the process necessary to design and implement your own message board using ADO.NET and XML. First, we will go through the process necessary to create the data structures in MS Access and SQL Server. We will analyze our application and break down the data into small pieces in order to represent them in a database. Next, we will determine the best way to design our application and go through the design of all the classes we will use to power the message board, and determine what methods and properties each class should contain.

Once the data analysis is done, we are going to develop our classes that will represent the core "business objects" in our application. These objects will be the guts of dotBoard and are what we will use in our user interface (UI) to allow our users to interact indirectly with the data in our database. The last step we will perform is creating the UI itself and allow users to interact with our message board.

One major point we should realize, however, is that no matter how large a project this message board seems, it is in fact incredibly simple once broken down into its smaller pieces. In fact, as you delve deeper and deeper into .NET, you will notice how much simpler it is to build most applications. With the right programming practices and .NET as your technology of choice, you can build complex applications in a much more efficient manner than some of the older technologies in existence.

Setting Up the Database

Setting up the database is one of the most important parts of any application. How do you represent your ideas in a structured, well-formed way? The first and most important step is to break down what you know you want your application to do, analyze those tasks, and then extract the important parts.

A message board has several distinguishable elements once you start to analyze it. The first and most obvious is that you need to store information on subjects and threads. If you've ever looked at a message board before, you'll notice that it's broken down into three levels. The first level is a general heading, describing what it is going to contain. We'll call this level *Board*. Board can contain any number of *Threads*. Finally, a Thread contains any number of *Messages*. The last area of data involved is data representative of a message board user. Users do not fit into this three-level hierarchy, but instead are a distinct part of each level. This very distinct hierarchy is a great place to start when defining your data.

The first thing to do is break down the type of information that describes a Board. This can be done many different ways: brainstorm, use your vast knowledge of all things data, or actually go to some bulletin boards on the Web and take a look at the kind of information they capture and display. This last option is probably the easiest, as there are numerous examples on the Web to look at.

That said, let's go through a Board and determine what type of information a Board uses. Our board will have the following information displayed on the user interface: name, description, list of threads, thread count, post count, the moderator, and some type of unique identifier. Not all of those fields need their values to be saved in the database. For example, the thread count and post count can be easily retrieved at runtime by just calculating them. That leaves Name, Description, Moderator, and a Unique Identifier.

Threads are less complex than Boards. A Thread contains the following fields: board ID, subject, post count, creator, and some type of unique identifier. The board ID should be created by a relationship between the two tables, and post count can be retrieved at runtime. That leaves subject and creator.

A *Post* is comprised of the following fields: subject, body, creator, thread ID, and some type of unique identifier. All of these must be captured in order to successfully represent a Post.

Finally, the last item we must capture is the user data. Most bulletin boards you visit do not allow anonymous posting. That is, in order to post, you must have your own user data. Our message board will function the same way. This makes it much easier to write your SQL statements and preserve database integrity. User information will contain the following fields: a unique identifier, username, password, first name, last name, e-mail address, whether or not this user is an administrator, and whether or not this user has been banned from posting.

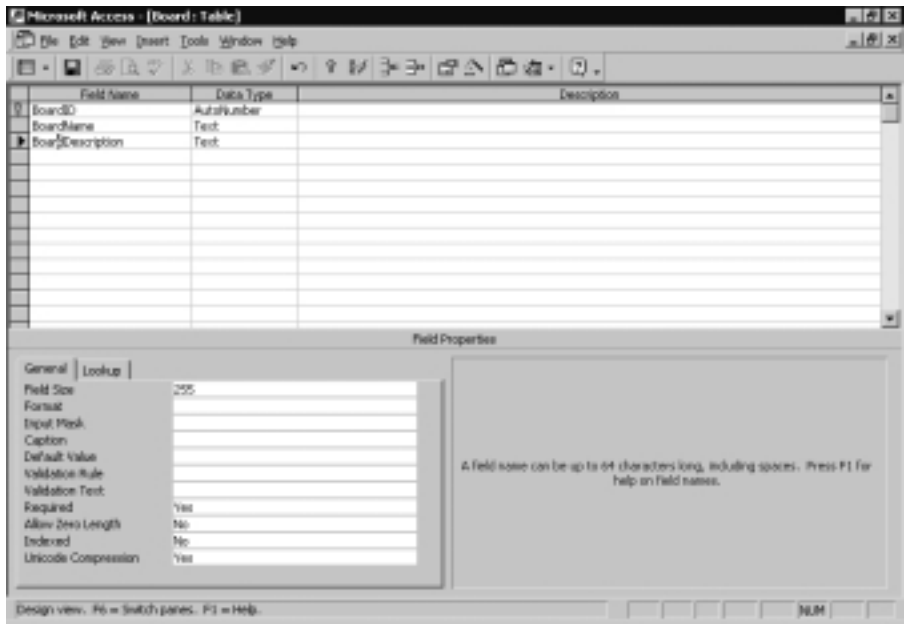
MS Access Database

Setting up your Access database is a pretty quick process. You can use the dotBoard .mdb file located on the companion Web Site for the book (www.syngress.com/solutions), or follow the steps provided next. If you want to create your own database, open up Microsoft Access (either 97 or 2000), and create a new database called dotBoard.mdb.

The Microsoft Access database is rather straightforward. As described previously, the Board table (Figure 8.1) will contains four fields: BoardID, BoardName, BoardDescription, and ModeratorID. BoardID should be an **AutoNumber**, with Indexed set to **Yes (No Duplicates)**, and should also be the primary key.

BoardName is a **Text** field, with **Required** set to **Yes**, and with a **Field Size** of 100. BoardDescription is a **Text** field, with **Field Size** set to the maximum access allows, which is 255.

Figure 8.1 The Board Table



The Threads table will also contain four fields: ThreadID, ThreadSubject, CreatorID, and BoardID. ThreadID should be an **AutoNumber**, with **Indexed** set to **Yes (No Duplicates)**, and should also be the primary key. ThreadSubject is a **Text** field, with **Field Size** set to the maximum access allows, which is 255. CreatorID is a **Number**, with **Field Size** set to **Long Integer**, and **Required** set to **Yes**. BoardID is a **Number**, with **Field Size** set to **Long Integer**, and **Required** set to **Yes** (Figure 8.2).

The Posts table will contain six fields: PostID, PostSubject, PostBody, CreatorID, ThreadID, and PostDate. PostID should be an **AutoNumber**, with **Indexed** set to **Yes (No Duplicates)**, and should also be the primary key. PostSubject is a **Text** field, with **Field Size** set to the maximum access allows, which is 255. PostBody is a **Memo** field with **Required** set to **Yes.** CreatorID is a **Number**, with **Field Size** set to **Long Integer**, and **Required** set to **Yes**. ThreadID is a **Number**, with **Field Size** set to **Long Integer**, and **Required** set to **Yes**. PostDate will be a Date/Time field with a **Default Value** of **Now()** and **Required** set to **Yes**. See Figure 8.3 for the Posts table.

Figure 8.2 The Threads Table

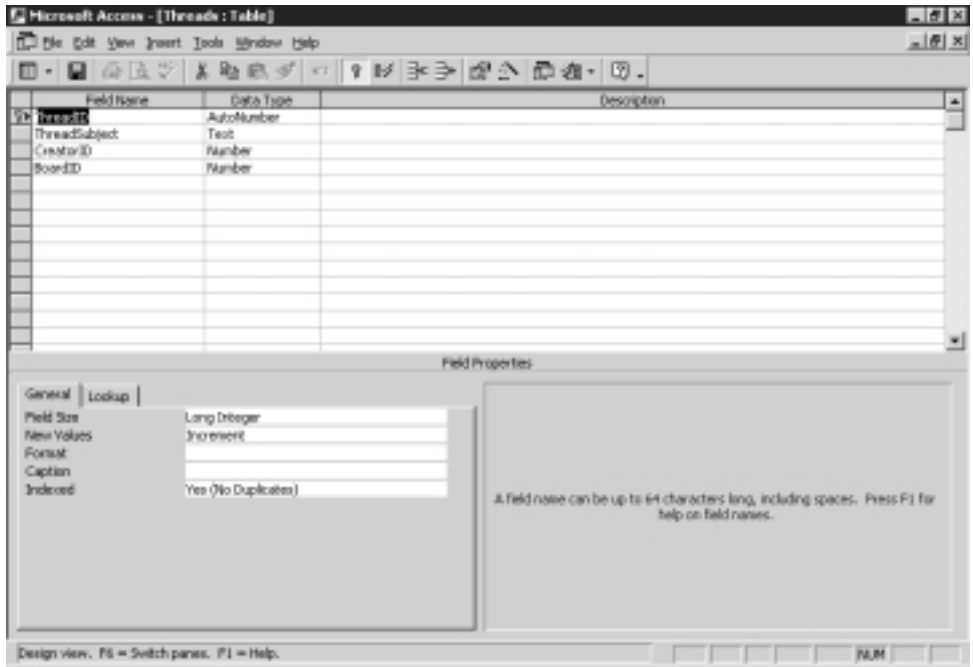
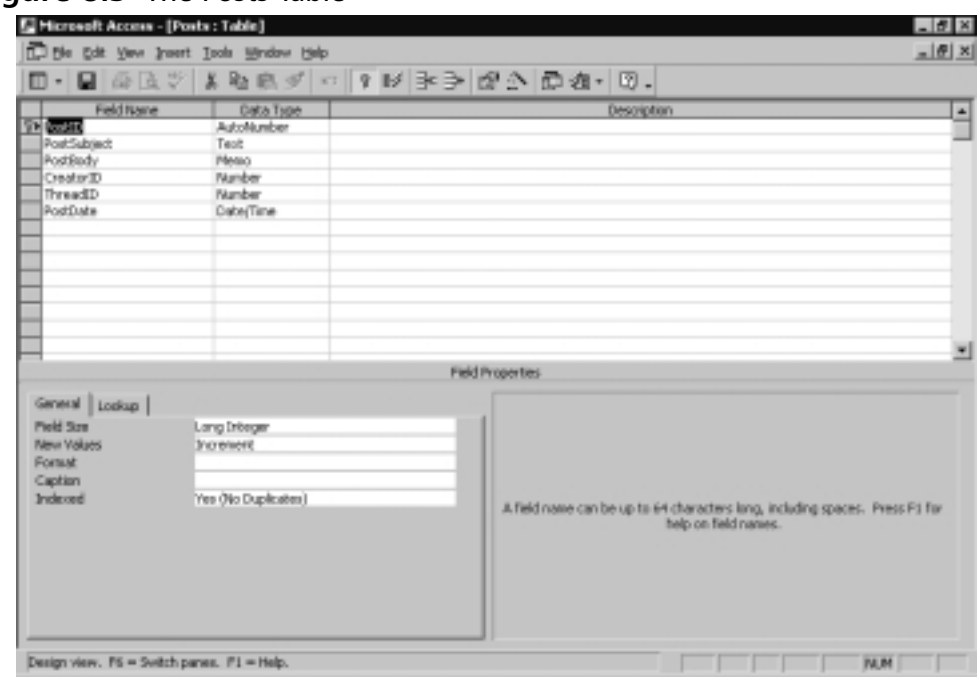
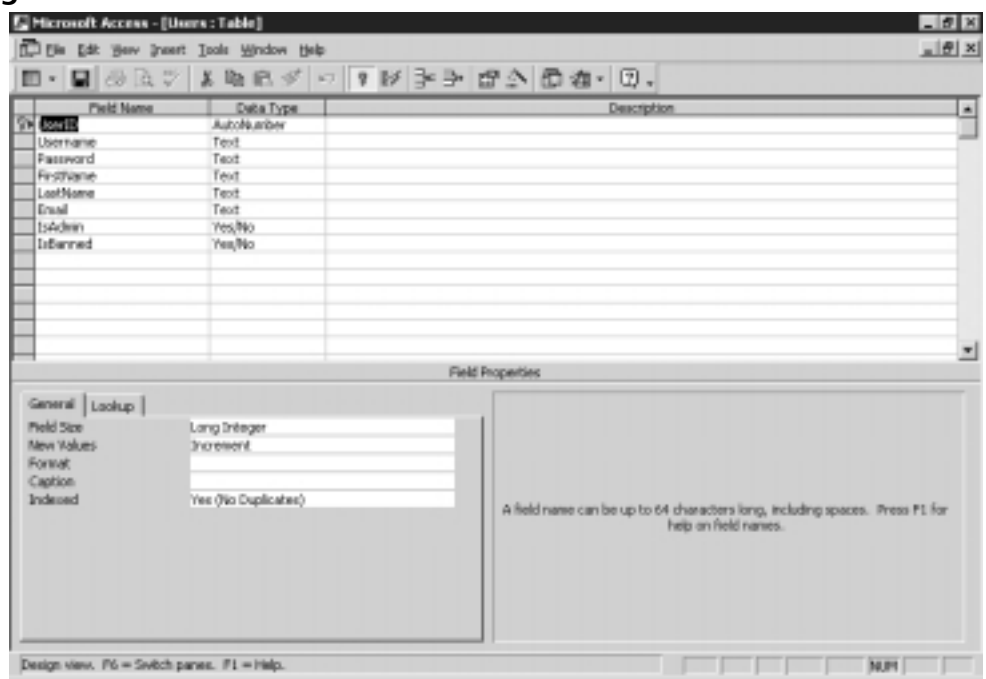


Figure 8.3 The Posts Table



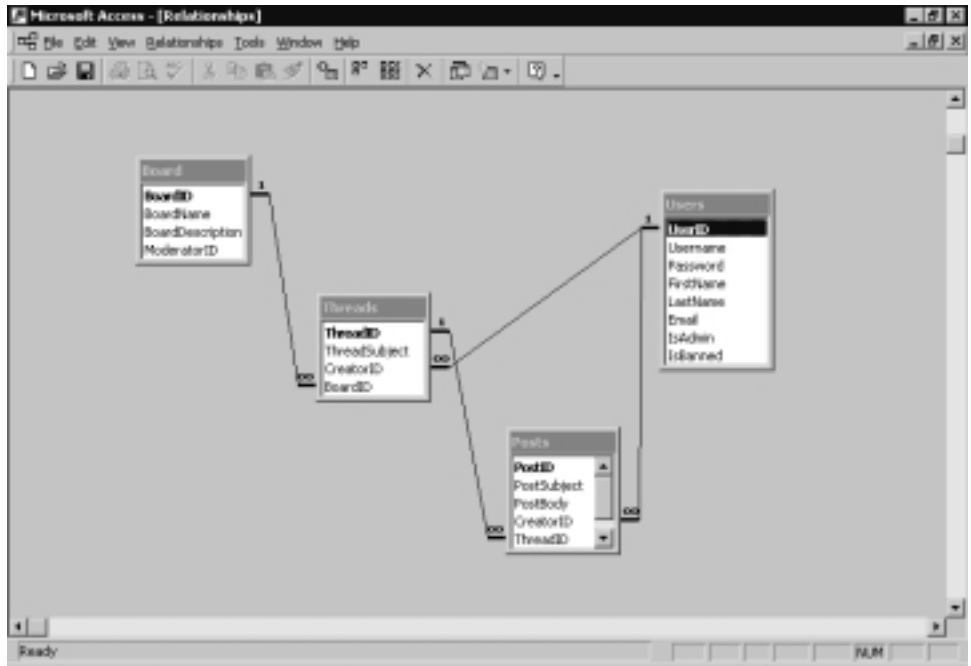
The Users table will contain eight fields: UserID, Username, Password, FirstName, LastName, Email, IsAdmin, IsBanned. UserID should be an **AutoNumber**, with **Indexed** set to **Yes (No Duplicates)**, and should also be the primary key. Username is a **Text** field, with its **Field Size** set to **50** and **Required** set to **Yes**. Password is a **Text** field, with its **Field Size** set to **50** and **Required** set to **Yes**. FirstName is a **Text** field, with its **Field Size** set to **100** and **Required** set to **Yes**. LastName is a **Text** field, with its **Field Size** set to **200** and **Required** set to **Yes**. Email is a **Text** field, with its **Field Size** set to **255** and **Required** set to **Yes**. IsAdmin and IsBanned are **Yes/No** fields, with **Format** set to **True/False** and **Required** set to **Yes**. See Figure 8.4 for the Users table.

Figure 8.4 The Users Table



The last step is to define the relationships between the tables. Posts relates to Threads on ThreadID. Threads relates to Board on BoardID. Users relates to Posts on CreatorID, to Threads on CreatorID, and Board on ModeratorID (Figure 8.5).

Figure 8.5 The Relationships between Tables



SQL Server Database

Setting up a SQL Server database is rather effortless, especially since you can let the database do everything for you by executing a SQL script. The only thing you need to do is open up your **SQL Enterprise Manager**, navigate to the server you want to create your database on, and open up the **Databases** node. Right-click the **Databases** node and select **New Database**. Name your database **dotBoard**, and select **OK**.

The only other action you need to take is to open **SQL Query Analyzer**, and execute the SQL Script shown in Figure 8.6 (which can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions) called dotBoard Setup.sql).

Figure 8.6 SQL Server Database Creation Script (dotBoard Setup.sql)

```
CREATE TABLE [dbo].[Board] (
      [BoardID] [int] IDENTITY (1, 1) NOT NULL ,
      [BoardName] [varchar] (100) COLLATE SQL_Latin1_General_CP1_CI_AS
          NOT NULL ,
```

Continued

Figure 8.6 Continued

```
      [BoardDescription] [varchar] (255) COLLATE SQL_Latin1_General_
          CP1_CI_AS NOT NULL
) ON [PRIMARY]
GO
CREATE TABLE [dbo].[Posts] (
      [PostID] [int] IDENTITY (1, 1) NOT NULL ,
      [PostSubject] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS
          NOT NULL ,
      [PostBody] [text] COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL ,
      [CreatorID] [int] NOT NULL ,
      [ThreadID] [int] NOT NULL ,
      [PostDate] [datetime] NOT NULL DEFAULT getDate()
) ON [PRIMARY] TEXTIMAGE_ON [PRIMARY]
GO
CREATE TABLE [dbo].[Threads] (
      [ThreadID] [int] IDENTITY (1, 1) NOT NULL ,
      [ThreadSubject] [varchar] (255) COLLATE SQL_Latin1_General_
          CP1_CI_AS NOT NULL ,
      [CreatorID] [int] NOT NULL ,
      [BoardID] [int] NOT NULL
) ON [PRIMARY]
GO
CREATE TABLE [dbo].[Users] (
      [UserID] [int] IDENTITY (1, 1) NOT NULL ,
      [Username] [varchar] (50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT
          NULL ,
      [FirstName] [varchar] (100) COLLATE SQL_Latin1_General_CP1_CI_AS
          NOT NULL ,
      [LastName] [varchar] (200) COLLATE SQL_Latin1_General_CP1_CI_AS NOT
          NULL ,
      [Email] [varchar] (255) COLLATE SQL_Latin1_General_CP1_CI_AS NOT
          NULL ,
      [IsAdmin] [bit] NOT NULL ,
```

Continued

Figure 8.6 Continued

```
      [IsBanned] [bit] NOT NULL
) ON [PRIMARY]
GO
ALTER TABLE [dbo].[Board] WITH NOCHECK ADD
      CONSTRAINT [PK_Board] PRIMARY KEY  CLUSTERED
      (
            [BoardID]
      )  ON [PRIMARY]
GO
ALTER TABLE [dbo].[Posts] WITH NOCHECK ADD
      CONSTRAINT [PK_Posts] PRIMARY KEY  CLUSTERED
      (
            [PostID]
      )  ON [PRIMARY]
GO
ALTER TABLE [dbo].[Threads] WITH NOCHECK ADD
      CONSTRAINT [PK_Threads] PRIMARY KEY  CLUSTERED
      (
            [ThreadID]
       )  ON [PRIMARY]
GO
ALTER TABLE [dbo].[Users] WITH NOCHECK ADD
      CONSTRAINT [PK_Users] PRIMARY KEY  CLUSTERED
      (
            [UserID]
      )  ON [PRIMARY]
GO
ALTER TABLE [dbo].[Users] WITH NOCHECK ADD
      CONSTRAINT [DF_Users_IsAdmin] DEFAULT (0) FOR [IsAdmin],
      CONSTRAINT [DF_Users_IsBanned] DEFAULT (0) FOR [IsBanned]
GO
ALTER TABLE [dbo].[Posts] ADD
      CONSTRAINT [FK_Posts_Threads] FOREIGN KEY
```

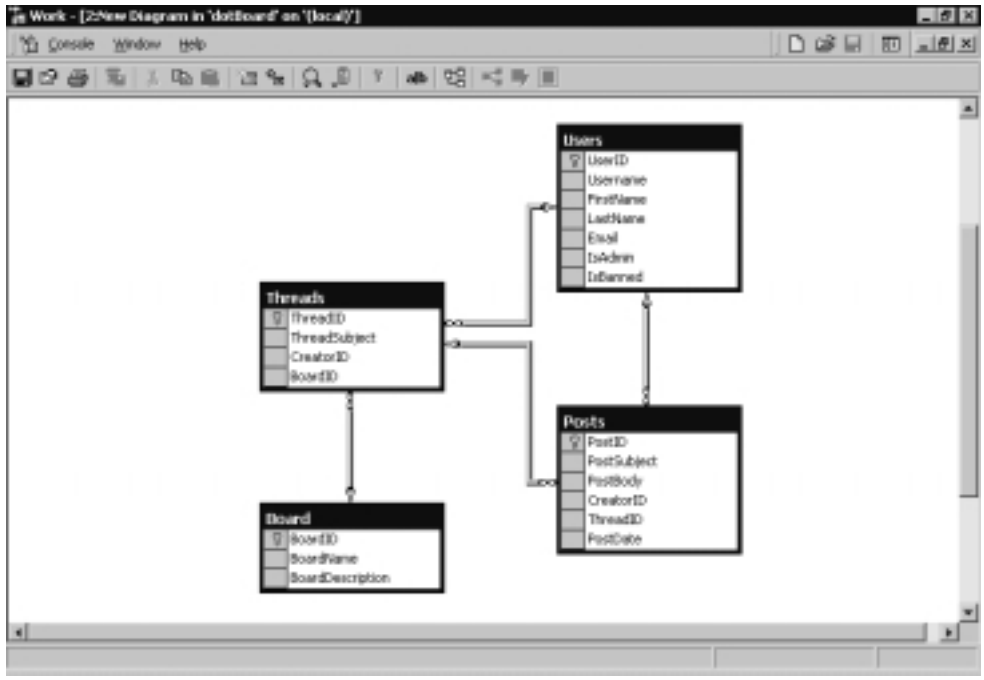
Continued

Figure 8.6 Continued

```
     (
          [ThreadID]
     ) REFERENCES [dbo].[Threads] (
          [ThreadID]
      ) ON DELETE CASCADE  ON UPDATE CASCADE ,
     CONSTRAINT [FK_Posts_Users] FOREIGN KEY
     (
          [CreatorID]
     ) REFERENCES [dbo].[Users] (
          [UserID]
     )
GO
ALTER TABLE [dbo].[Threads] ADD
     CONSTRAINT [FK_Threads_Board] FOREIGN KEY
     (
          [BoardID]
     ) REFERENCES [dbo].[Board] (
          [BoardID]
     ) ON DELETE CASCADE  ON UPDATE CASCADE ,
     CONSTRAINT [FK_Threads_Users] FOREIGN KEY
     (
          [CreatorID]
     ) REFERENCES [dbo].[Users] (
          [UserID]
     )
GO
```

Finally, go back to your SQL Enterprise Manager, navigate to your database, and select the **Diagrams** node. Right-click and select **New Database Diagram**. Click **Next**, then select our four database tables, and click **Next**. The diagram should be created automatically for us, and should look a lot nicer than the MS Access version (Figure 8.7).

Figure 8.7 The SQL Server Diagram



Designing Your Application

When designing an application, there are two possible main routes. The first is the procedural approach (anyone familiar with ASP 3.0 and earlier who did not use COM objects to handle logic knows exactly what I mean): the simple "Page1" does this, "Page2" does this, and so on. You have a set of "top-down" ASP scripts (that is, your code starts at the top and executes until it hits the bottom), with functions including files, which make up your application. There is technically nothing wrong with this approach, as there are many large-scale applications written exactly this way.

Your other choice is to take a more object-oriented (OO) approach. In an OO world, you create a set of classes and interfaces that make up the core of your application. Using these classes, you create a user interface that will define what an average user would consider your "application." This approach allows the designer of the classes to encapsulate a good majority of the code and logic behind the application, without exposing it to whomever might be building the user interface (and likely, the same person will be building both). An OO approach also allows your application to be used in a variety of ways, and would

allow someone to build multiple user interfaces on top of the exact same set of classes.

Both approaches have their merits and flaws. With the procedural approach, you will be stuck in ASP.NET for your user interface, and if you want to "copy" logic from one place to another, you either have to create globally scoped functions, or copy and paste code. The procedural approach does tend to be a bit easier to create, though, because you do not have the additional overhead of having to create classes to handle your logic and data. The object-oriented approach effectively encapsulates your entire application into a small set of classes, grouped into an assembly .dll file, which can be created from another application and used. This allows you to hand another developer your assembly file, and let him or her go about building the actual user interface without you ever needing to know what the user interface was. The drawback to building an application in an OO manner is that whomever is developing the classes needs to take a lot of care to get it done properly, and be able to build it in such a manner as to not tie it exclusively to one type of user interface.

When deciding whether dotBoard should be procedural or object-oriented, take into account these things: First and foremost, you need to be able to maintain your code. If your code is modularized into multiple functions and organized very well, then the procedural approach doesn't seem too bad. However, if your code is placed haphazardly throughout your application, finding your bugs and improving code at a later date might be harder. If your code is organized into a set of classes and public interfaces (the methods and procedures that an application can "see"), it is typically easier to maintain your code, as each piece of each class is a very small piece of the application as a whole, and making changes won't likely take a large amount of code.

The other thing you should think about is that dotBoard is being written in VB.NET. For anyone who has built an application in straight ASP, you would probably be more comfortable with the procedural approach. For anyone who has built an application in ASP and created VB COM objects, you would most likely be more comfortable with the object-oriented approach, but feel some trepidation about speed and performance issues. For the master gurus out there who are building C++ ATL COM objects and using them in ASP, you might scoff at VB.NET and think you would rather stick with C++ ATL COM. Well, all of you have very valid points. A straight procedural approach is generally looked down upon in a professional environment, VB COM objects in ASP are typically regarded as slow and frequently memory- and processor-intensive, and well, nobody can read the C++ ATL code anyway, so it doesn't count!

Seriously, though, every point made is very valid about every technology discussed. That's where VB.NET comes in. The .NET runtime is remarkably fast. The Just-In-Time (JIT) compilation of your code only happens the first time it is executed; so, after that initial execution, your code runs incredibly fast until you change it (at which point the JIT compilation happens again). VB.NET is also a fully object-oriented language. It provides developers with every good OO technique available, and it is actually quite easy to write OO applications with it.

All that said, it's pretty obvious that dotBoard should be an object-oriented application. Don't worry if you've never written any OO code before. Object-oriented techniques are relatively easy to implement, and even if you don't think you've ever used any objects before, you probably have (especially if you've done any development in ASP).

Designing Your Objects

Now that we've decided on object orientation, we need to analyze our application and determine what our objects will "look like." At this point, you might say, "we've already done that while analyzing and building our database," and you'd be right. We have already done that. Half of our design work is now already done! The only other part we have to do is map the data we've already analyzed to VB.NET types and group them accordingly. We're going to do that with the wonder of UML (Unified Modeling Language). If you don't know what UML is, don't worry; all we're using it for here is to show you some pretty pictures of what our classes are going to look like. Please note that all of these objects and all files can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Creating Your Data Access Object

To make it easier for each of your objects to have access to the database, we're going to create a singular data access object that does everything for you. We're going to call this class *DataControl*, and it is going to be comprised of solely shared methods. A shared method means you do not need to create an instance of a *User* object to call it. *DataControl* will contain two Shared methods, *GetDataSet* and *ExecuteNonQuery*. *GetDataSet* returns a *DataSet*, and *ExecuteNonQuery* executes a SQL statement and returns nothing. This class is pretty straightforward, and is shown in Figure 8.8 (likewise, the complete source code for DataControl.vb can be found on the companion Solutions Web site for the book).

SYNGRESS
syngress.com

Figure 8.8 DataControl.vb

```
Imports System.Data
Imports System.Data.OleDb
Imports System.Web
Imports System.Configuration
Imports System.Collections.Specialized

Public Class DataControl
    Public Shared Function GetDataSet(ByVal SQL As String) As DataSet
        Dim connectionString As String
        Dim settings As ConfigurationSettings
        Dim appSettings As NameValueCollection

        appSettings = settings.AppSettings()
        connectionString = appSettings.Item("ConnectionString")

        Dim connection As New OleDbConnection(connectionString)
        connection.Open()

        Dim adapter As New OleDbDataAdapter(SQL, connection)
        Dim myData As New DataSet()

        adapter.Fill(myData)
        adapter.Dispose()
        connection.Close()

        Return myData
    End Function

    Friend Shared Sub ExecuteNonQuery(ByVal SQL As String)
        Dim connectionString As String
        Dim settings As ConfigurationSettings
        Dim appSettings As NameValueCollection

        appSettings = settings.AppSettings()
```

Continued

Figure 8.8 Continued

```
        connectionString = appSettings.Item("ConnectionString")

        Dim connection As New OleDbConnection(connectionString)
        connection.Open()

        Dim myCommand As New OleDbCommand()
        myCommand.Connection = connection
        myCommand.CommandText = SQL
        myCommand.CommandType = CommandType.Text
        myCommand.ExecuteNonQuery()
        'clean up
        connection.Close()
        myCommand.Dispose()
        connection.Dispose()
    End Sub
End Class
```

You see that the two methods in *DataControl* are in fact, rather simple. These functions connect to the database and do a specific function (execute SQL scripts and one returns a *DataSet*). The one thing to note is that the connection string is being retrieved from the *ConfigurationSettings.AppSettings*. These are dynamic settings that the .NET runtime gives you access to. When you're running an ASP.NET application, they are located in the *web.config* file. In another type of application, they are located in *ProjectName.exe.config*. That's it for our Data object. The next step is to take a look at our *User* object.

Designing the *User* Class

When we looked at the user information when thinking about the database, we discovered a number of fields that needed to reside in the *User* table. Luckily, all our classes will be structured in a way to nearly match the database; the *User* class is no exception. The only difference is that this *User* is a VB.NET class and not a database table.

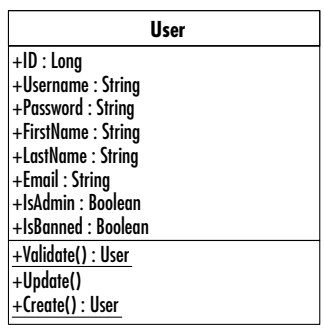
There are four basic types of users: *Guests*, *Registered Users*, *Administrators*, and *Moderators*. All of these should be represented when we build our *User* class.

Again, you might say something like "but this is an object-oriented application, and if we have multiple types of one object, shouldn't they be separate?" Again, you would be right. There are three types of users. All have similar properties; the only difference is that some do certain things that others can't. For example, a Registered User in a bulletin board would have the ability to post threads and messages, whereas a Guest user would not. A Registered User would also have the ability to edit his or her profile and edit his or her messages, whereas a Guest user would not be able to. An Administrator would have the ability to do everything a Registered User could, except globally. A Moderator can modify posts and threads in boards to which he or she has Moderator privileges.

Now that we've identified the multiple types of users, we need to determine if we should have multiple types of users in our application. A Guest can only browse a bulletin board, as no security is necessary for browsing. A Registered User can create and edit posts, and modify his or her profile. An Administrator can do anything he or she wants to the bulletin board. A Moderator can do what a Registered User can, and can act like an Administrator on the board to which he or she is given moderation rights.

You might want to build some neat OO objects here, but all these things can be accomplished through a single *User* class. Take a look at Figure 8.9.

Figure 8.9 The *User* Object Diagram



You see that our *User* object will have the exact same fields as our database table, which is named exactly the same. This makes it a bit easier to remember which field in the object matches up to which field in the database. The other thing you should notice is the three items at the bottom of the diagram: *Create*, *Validate*, and *Update*. All are methods the *User* object will have. *Update()* will update the user's details and save them to the database. *Validate* is a shared method

of the *User* class, and can be used to perform all user validation. Create is also a shared method, and can be used to create a brand new user in the database.

That's it. That's the whole *User* object. Not much to it, is there? It has a Boolean field to signify whether or not it is an Administrator, and each Board object will store the ID of the Administrator of that Board, so the *User* object doesn't have to. The only other thing to mention is Guest users—a Guest user will just be a User that is Nothing. That is, if you are currently a guest in the application, you won't have a *User* object created for you. Let's take a look at the code involved to create this *User* object in Figure 8.10. The complete source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.10 The Basics (User.vb)

```
Public Class User
    Private mUsername As String
    Private mPassword As String
    Private mFirstName As String
    Private mLastName As String
    Private mUserID As Long
    Private mIsAdmin As Boolean
    Private mEmail As String
    Private mUserID As Long
End Class
```

That part is clear enough. We declare the *User* class, and the private variables necessary to represent each user. Next, declare the public properties for each of these private variables as shown in Figure 8.11. The complete source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.11 Public Properties (User.vb)

```
    Public WriteOnly Property Password() As String
        Set(ByVal Value As String)
            MPassword = Value
        End Set
    End Property
```

Continued

Figure 8.11 Continued

```
    Public ReadOnly Property ID() As Long
        Get
            Return mUserID
        End Get
    End Property

    Public Property LastName() As String
        Get
            Return mLastName
        End Get
        Set(ByVal Value As String)
            mLastName = Value
        End Set
    End Property

    Public Property FirstName() As String
        Get
            Return mFirstName
        End Get
        Set(ByVal Value As String)
            mFirstName = Value
        End Set
    End Property

    Public Property Username() As String
        Get
            Return mUsername
        End Get
        Set(ByVal Value As String)
            mUsername = Value
        End Set
    End Property

    Public Property IsAdmin() As Boolean
```

Continued

Figure 8.11 Continued

```
        Get
            Return mIsAdmin
        End Get
        Set(ByVal Value As Boolean)
            mIsAdmin = Value
        End Set
    End Property

    Public Property IsBanned() As Boolean
        Get
            Return mIsBanned
        End Get
        Set(ByVal Value As Boolean)
            mIsBanned = Value
        End Set
    End Property

    Public Property Email() As String
        Get
            Return mEmail
        End Get
        Set(ByVal Value As String)
            mEmail = Value
        End Set
    End Property
```

With that out of the way, let's look at the methods the *User* object will have. As we saw earlier, there will be three methods: *Validate*, *CreateUser*, and *Update*. *Validate* is a shared method, which will give a developer the ability to validate and return a valid *User* object, or throw an exception. *CreateUser* is also a shared method that gives the developer the ability to create a new *User* object. Finally, *Update* will allow a developer to update the private fields in the *User* object and commit them to the database. This will be for tasks such as saving passwords and updating e-mail addresses. Let's take a look at the first

method, *Validate*, in Figure 8.12. Figure 8.12. The complete source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.12 The *Validate* Method (User.vb)

```
Public Shared Function Validate(ByVal username As String, _
    ByVal password As String) As User
    If password.Equals("") Then
      Throw New ArgumentException("You must enter a password.")
    Else
      Dim myData As DataSet = DataControl.GetDataSet("SELECT * " & _
          "FROM [Users] WHERE [UserName] = '" & username & "'")
        If myData.Tables(0).Rows.Count <= 0 Then
            Throw New ArgumentException("Username does not exist.")
        Else
          If CBool(myData.Tables(0).Rows(0)("IsBanned")) = True Then
              Throw New Exception("User is banned")
            Else
              If password <> _
                  CStr(myData.Tables(0).Rows(0)("Password")) Then
                      Throw New ArgumentException("Invalid password")
              Else
                  Return New User(myData.Tables(0).Rows(0))
              End If
           End If
        End If
    End If
End Function
```

The *Validate* method accepts a username and a password as parameters, and attempts to verify that those parameters are a valid combination for a registered user. If the password is empty, it throws an *ArgumentException*. If, while looking up the username, it finds that the username is not present in the database, it again throws an *ArgumentException*. If the username exists, but the user is banned, then it throws an *ArgumentException*. If the username exists, the user is

not banned, and the password passed in was incorrect, once again it throws an *ArgumentException*. Finally, if the username is valid and the password is correct, it returns a new *User* object, passing in the first *DataRow* to the *User* constructor.

At this point, you're probably wondering why we haven't discussed the constructor of the *User* object. Well, wait no longer! Here's the code for the *User* object constructor in Figure 8.13. The complete source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.13 Constructors (User.vb)

```
    Public Sub New(ByVal userId As Long)
        Dim myData As DataSet
        myData = DataControl.GetDataSet("SELECT * FROM Users " & _
"WHERE UserID = " & Me.mUserID)

        If myData.Tables(0).Rows.Count <= 0 Then
            Throw New ArgumentException("The requested user " & _
         does not exist.")
        Else
            inflate(myData.Tables(0).Rows(0))
        End If

        myData.Dispose()
    End Sub

    Public Sub New(ByVal row As DataRow)
        inflate(row)
    End Sub
```

There are two constructors here. The second constructor is what the *Validate* method called. That constructor forwards the *DataRow* on to another method called *inflate*, which will be discussed in a moment. The first constructor accepts a user ID as a parameter. This user ID is synonymous with the *UserID* field in the User table. The constructor looks up the user based on the user ID. If that user

ID is not found, it throws an *ArgumentException*. If the user ID *is* found, it forwards the first *DataRow* in the *DataSet* to the *fillData* method in Figure 8.14. The complete source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.14 The *fillData* Method (User.vb)

```
Private Sub inflate(ByVal row As DataRow)
    Me.mUsername = CStr(row("Username"))
    Me.mFirstName = CStr(row("FirstName"))
    Me.mLastName = CStr(row("LastName"))
    Me.mIsAdmin = CBool(row("IsAdmin"))
    Me.mEmail = CStr(row("Email"))
    Me.mUserID = CLng(row("UserID"))
    Me.mPassword = CStr(row("Password"))
End Sub
```

As you can see, the *inflate* method accepts a *DataRow* as a parameter, and populates all the private fields with values from the database. This is frequently called "inflating" your objects; hence, the appropriately named subroutine. The other thing to notice is that inflate is a private subroutine. This is because you don't want any objects outside of the current *User* object to have access to this method. It does "utility" work on the object, and is unnecessary for any other object to call this method.

Now that we've discussed how to validate and return a valid *User* object, let's move on to creating users. Any user can have any username. The only restriction is that no two users can have the same username. This is because if you had two users with the same username, the only way to identify which one you wanted is to have some other type of unique identifier. Unfortunately, people can typically remember names and usernames much better than they can some (relatively) random number. So, in order to keep this username unique, you have to manually check. If you were a database administrator, you would probably insist on creating a unique index on the username field in the database, which is completely reasonable. If you feel you need the extra "security" in place to make sure the same username isn't taken twice, go ahead and put it in there, but it's in the *CreateUser* method as well, which we will now take a look at in Figure 8.15. The complete

source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.15 The *CreateUser* Method (User.vb)

```
Public Shared Function CreateUser(ByVal userName As String, _
    ByVal password As String, _
    ByVal firstName As String, _
    ByVal lastName As String, ByVal email As String) As User

    Dim sql As String
    Dim myData As DataSet

    sql = "SELECT userName FROM Users WHERE userName = '" & _
         userName & "'"
    myData = DataControl.GetDataSet(sql)
    If myData.Tables(0).Rows.Count <= 0 Then
        'this username has not been taken
        sql = "INSERT INTO [Users] ([Username], [Password], " & _
            "[FirstName], [LastName], " & _
            "[Email], [IsAdmin], [IsBanned]) VALUES ('" & userName & _
            "','" & password & "','" & firstName & "','" & lastName & _
            "','" & email & "',0,0)"
        DataControl.ExecuteNonQuery(sql)
        Return User.Validate(userName, password)
    Else
        'this username has already been taken
        Throw New ArgumentException("The username is already taken")
    End If
End Function
```

First, the *CreateUser* function scans the database to see if the request username already exists. If it does, it throws an *ArgumentException*. If the username doesn't exist, it builds a SQL statement to insert a new row into the user table and executes it. Finally, it calls the *Validate* method and returns the result.

Debugging…

Creating Console Applications to Test Your Progress

Visual Studio .NET gives us an easy way to test and debug our applications, without actually needing to have a decent user interface to look at—they call it a Console Application. Sure, Console Applications are useful by themselves when you don't need a UI, but when you are building a relatively large application and you don't want to get yourself confused trying to build the UI and the classes at the same time, consider using a Console Application to debug your project.

Go ahead and try it.

1. Add a new Console Application to your project.
2. Add a reference to your **dotBoardObjects** project to the Console Application.
3. Set your new Console Application as the start-up project.
4. Start putting in some code to test the classes you've written—maybe something like this:

```
Dim myUser As User
myUser = User.CreateUser("myuser", "mypassword", "joe", _
    "blow", "joe.blow@email.com")
Console.WriteLine(myUser.FirstName)
Console.WriteLine("Press enter to finish")
Console.ReadLine()
```

Before you run this, put a break point on the line that creates a user.

5. Step through the code using **F8** (if you set up your Visual Studio to use the Visual Basic Profile), and watch as the execution moves into the *User* class you created. You can step through your application and watch as every line of code is executed. If an error pops up, stop your application, fix the error, and run the application again.

You should use and abuse this technique as much as possible. Not only does it allow you to test and debug your classes, but it also does it without your needing to build a UI at the same time you build the objects.

The last method to discuss is the *Update* method. This method updates the database with the current state of the object. See Figure 8.16 for the *Update* method. The complete source code for User.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.16 The *Update* Method (User.vb)

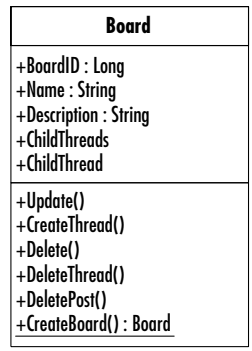
```
Public Sub Update()
    Dim sql As String
    sql = "UPDATE [Users] SET [Password] = '" & mPassword & _
        "', [FirstName] = '" & mFirstName & _
        "', [LastName] = '" & mLastName & _
        "', [Email] = '" & mEmail & "'"
    If Me.IsAdmin = True Then
        sql = sql & ", [IsAdmin] = 1"
    Else
        sql = sql & ", [IsAdmin] = 0"
    End If
    If Me.IsBanned = True Then
        sql = sql & ", [IsBanned] = 1"
    Else
        sql = sql & ", [IsBanned] = 0"
    End If
    sql = sql & " WHERE [UserID] = " & mUserID.ToString()
    DataControl.ExecuteNonQuery(sql)
End Sub
```

Again, this method is rather simple; it generates a SQL statement to update the database. The If statements are there to insert the correct Boolean value into the database instead of "True" or "False." Finally, after building the SQL statement, it executes it and exits the method.

Designing the *Board* Class

Now that we've designed the *User* class, let's take a look at the *Board* class. Many of the concepts in the *User* class will be taken from the *Board* class. That is, the *Board* class will mimic the Board table in the database, and will have a couple of similarly named methods as in the *User* class. Let's take a look at a UML diagram of the *Board* class in Figure 8.17.

Figure 8.17 The *Board* Class



Just by looking at this diagram, you can see that the *Board* class has much more functionality than the *User* class. Notice the four fields from the Board table: *BoardID*, *Name*, and *Description*. Just like the *User* class, these are directly representative of what exists in the database. The other two fields you shouldn't recognize. *ChildThreads* returns a list of the Threads that exist in this Board. *ChildThread* is a property that accepts a *ThreadID* to return a specific Thread that is directly located in a specific Board.

The methods available to a *Board* object should be somewhat self-explanatory. The *Update* method does exactly what the *User* class *Update* method did: updates the database with the private fields in the database. The *Delete* method deletes the Board from the database. *DeleteThread* deletes a specific Thread from the database. *DeletePost* deletes a specific Post that is located somewhere in this Board. *CreateThread* creates a new Thread and adds it to the private list of Threads in this Board. Like the *User* class, the *Board* class has a way to create new Boards, called *CreateBoard*. Let's start by showing the basics of the *Board* class in Figure 8.18. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.18 Private Fields and Public Properties (Board.vb)

```
Public Class Board
    Private mBoardID As Long
    Private mName As String
    Private mDescription As String
    Private myThreads As ThreadList
```

Continued

Figure 8.18 Continued

```
    Public ReadOnly Property ChildThread(ByVal threadId As Long) As _
        Thread
        Get
            'lookup the correct thread
            Dim i As Integer
            For i = 0 To Me.ChildThreads.Count - 1
                Dim myThread As Thread = Me.ChildThreads.Item(i)
                If myThread.ID = threadId Then
                    Return myThread
                End If
            Next i
            'if we've gotten to this point, there is no thread
            'with that ID in this board. throw an exception
            Throw New ArgumentException("Thread does not exist")
        End Get
    End Property

    Public ReadOnly Property ChildThreads() As ThreadList
        Get
            Return myThreads
        End Get
    End Property

    Public ReadOnly Property ID() As Long
        Get
            Return mBoardID
        End Get
    End Property

    Public Property Name() As String
        Get
            Return mName
        End Get
        Set(ByVal Value As String)
            mName = Value
```

Continued

Figure 8.18 Continued

```
        End Set
    End Property

    Public Property Description() As String
        Get
            Return mDescription
        End Get
        Set(ByVal Value As String)
            mDescription = Value
        End Set
    End Property
```

The public properties in this class are a little more complex than the properties in the *User* class. The public properties for the private fields are easy to understand, but *ChildThread* and *ChildThreads* are a bit more complex, as is the private *myThread* variable. Let's start with *myThread*, which is defined as being of type *ThreadList*. If you're familiar with the *System.Collections* namespace, you'll definitely notice that this is not one of the built-in .NET collections. ThreadList is actually a custom list that wraps an *ArrayList*, which will be discussed a bit later. For now, just accept the fact that this list collects all the Threads in a given Board.

The *ChildThreads* property returns the private *myThreads* variable. The *ChildThread* property accepts a *ThreadID* as a parameter, and looks up that *ThreadID* in the *myThreads* list. It loops through the list, and compares the ID of the Thread in the list with the *ThreadID* passed in. If it finds a match, it returns that Thread; otherwise, it throws an *ArgumentException*. Again, *ThreadList* will be discussed later, but for now, let's move on to the shared *CreateBoard* method, as shown in Figure 8.19. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.19 The *CreateBoard* Method (Board.vb)

```
Public Shared Function CreateBoard(ByVal name As String, _
    ByVal description As String, _
    ByVal creator As User) As Board

    Dim sql As String
```

Continued

Figure 8.19 Continued

```
    Dim myData As DataSet

    If creator.IsAdmin = True Then
        sql = "SELECT BoardName FROM [Board] WHERE [BoardName] = '" & _
                name & "'"
        myData = DataControl.GetDataSet(sql)
        If myData.Tables(0).Rows.Count <= 0 Then
            'this board name does not already exist.
            sql = "INSERT INTO [Board] ([BoardName], " & _
                "[BoardDescription], " & _
                ") VALUES ("
            sql &= "'" & name & "','" & description & _
                "')"
            'create the board
            DataControl.ExecuteNonQuery(sql)
            'return the board
            Return New Board(name)
        Else
            'board name already exists
            Throw New Exception("This board name already exists")
        End If
    Else
        Throw New Exception("Only admins may create boards")
    End If
End Function
```

The first step in this method is to check to see if the user that is requesting that a new Board be created is an admin. If the user is not an admin, it throws an exception. If the user is an admin, it then checks to see if a Board with that name has already been created. Like the username field in the *User* class, the name field in the *Board* class should be unique. This makes it easier to manage your Boards and to make sure they're named appropriately. If the Board name already exists, it throws an exception; otherwise, it generates the SQL statement necessary to create a Board. It then executes the SQL statement and returns a new *Board* object based on the Board name. Let's take a look at the Board constructor,

shown in Figure 8.20, to see what it does. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.20 Constructor (Board.vb)

```
Public Sub New(ByVal name As String)
    Dim sql As String
    Dim myData As DataSet
    sql = "SELECT * FROM [Board] WHERE [BoardName] = '" & _
        name & "'"
    myData = DataControl.GetDataSet(sql)

    If myData.Tables(0).Rows.Count > 0 Then
        Me.inflate(myData.Tables(0).Rows(0))
    Else
        Throw New Exception("Board does not exist")
    End If
End Sub

Private Sub inflate(ByVal myRow As DataRow)
    mName = CStr(myRow("BoardName"))
    mDescription = CStr(myRow("BoardDescription"))
    mBoardID = CLng(myRow("BoardID"))

    myThreads = New ThreadList(mBoardID)
End Sub
```

The Board constructor takes the Board name as a parameter, and looks in the database for that Board name. If it cannot find the Board, it throws an Exception; otherwise, it passes the first *DataRow* in the *DataSet* to the inflate method. The inflate method functions exactly as it did in the *User* class: it fills the private fields with values. The only difference here is that the *myThreads* variable is initialized and the *BoardID* is passed to it. Again, the *ThreadList* will be discussed a bit later, but trust that the *ThreadList* takes the *BoardID* passed in and creates a collection of the Threads in this Board. Next, let's take a look at the *Update* method in Figure 8.21. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.21 The *Update* Method (Board.vb)

```
Public Sub Update(ByVal requestor As User)
    If requestor.IsAdmin Then
        'update the database with this board's details
        Dim sql As String
        sql = "UPDATE [Board] SET [BoardName] = '" & mName & _
            "', BoardDescription = '" & mDescription & _
            " WHERE [BoardID] = " & mBoardID.ToString()
        DataControl.ExecuteNonQuery(sql)
    End If
End Sub
```

The *Update* method in the *Board* class does exactly what the *User* class's *Update* method did. The only real difference here is that it checks to make sure the user requesting the update is really an admin. If the user is not an admin, then it throws an exception. Next, take a look at Figure 8.22 for the *CreateThread* method. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.22 The *CreateThread* Method (Board.vb)

```
Public Sub CreateThread(ByVal subject As String, _
    ByVal creator As User)

    Dim sql As String
    sql = "INSERT INTO [Threads] ([ThreadSubject], " & _
        "[CreatorID], [BoardID]) VALUES ('" & subject & _
        "'," & creator.ID.ToString() & "," & _
        mBoardID.ToString() & ")"
    DataControl.ExecuteNonQuery(sql)

    'reinitialize the thread list
    myThreads.InitializeThreads()
End Sub
```

The *CreateThread* method builds the SQL statement necessary to insert a new Thread into the database, and then reinitializes the private *ThreadList* variable by calling its *InitializeThreads* method. You MIGHT be wondering why the *Board*

class has the *Create* method for its child objects, whereas both the *User* class and *Board* class have their *Create* method located in their class definitions. This is because both the *User* class and *Board* class do not have any parent-child relationships with any other classes. When you have a parent object and multiple child objects, the typical place to put the creation of the child objects is in the parent object. This is a matter of semantics—if you prefer to have your child objects create themselves, feel free to do it that way.

Let's explore how to delete objects. The *Board* class contains the *Delete*, *DeleteThread*, and *DeletePost* methods. The *Board* class can obviously delete itself, but why would it also contain the capability to delete both Threads and Posts? It has these two methods because the *Board* class is where the *ModeratorID* lives, and Moderators can delete both Threads and Posts, so it just seems natural to put these two delete methods in the *Board* class. Look at Figure 8.23 for the code. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.23 The *Delete* Method (Board.vb)

```
Public Sub Delete(ByVal requestor As User)
    'only admins can delete boards
    If requestor.IsAdmin Then
        Dim sql As String
        sql = "DELETE FROM Boards WHERE BoardID = " & _
            mBoardID.ToString()
        DataControl.ExecuteNonQuery(sql)
    Else
        Throw New ArgumentException("User not permitted to delete")
    End If
End Sub
```

The first step in the *Delete* method is to check to make sure the requesting user has the appropriate access rights to delete this board. If the user is not an admin, then an *ArgumentException* is thrown. If the user does have access rights to delete a Board, then the SQL statement is built to delete the Board from the database. The SQL statement is executed, and the Board is officially deleted. You can see the *DeleteThread* method in Figure 8.24. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.24 The *DeleteThread* Method (Board.vb)

```
Public Sub DeleteThread(ByVal thread As Thread, ByVal requestor As User)
    If requestor.IsAdmin Then
        Dim sql As String
        sql = "DELETE FROM Threads WHERE ThreadID = " & _
            thread.ID.ToString()
        DataControl.ExecuteNonQuery(sql)
        'reinitialize the threads
        myThreads.InitializeThreads()
    Else
        Throw New ArgumentException("User not permitted to delete")
    End If
End Sub
```

The first step in the *DeleteThread* method is to make sure the requesting user has the appropriate access to delete this thread. If the user is neither an admin nor a moderator of this Board, then an *ArgumentException* is thrown. If the user does have access to delete a Thread, then the SQL statement is built to delete the Thread from the database. The SQL statement is executed, and the *ThreadList* is reinitialized by calling its *InitializeThreads* method.

The next method we need is the *DeletePost* method. Take a look at Figure 8.25 for its implementation. The complete source code for Board.vb can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.25 The *DeletePost* Method (Board.vb)

```
Public Sub DeletePost(ByVal thread As Thread, ByVal post As Post, _
    ByVal requestor As User)

    If requestor.IsAdmin Then
        Dim sql As String
        sql = "DELETE FROM Posts WHERE PostID = " & _
            post.ID.ToString()
        DataControl.ExecuteNonQuery(sql)
        'reinitialize the posts in the thread
        thread.ChildPosts.InitializePosts()
```

Continued

Figure 8.25 Continued

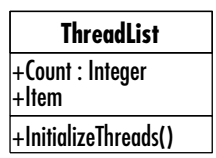
```
    Else
        Throw New ArgumentException("User not permitted to delete")
    End If
End Sub
```

Just as in the *DeleteThread* method, the first step in the *DeletePost* method is to make sure the requesting user has the appropriate access rights to delete this post. If the user is neither an admin nor a moderator of this Board, then an *ArgumentException* is thrown. If the user does have access to delete a post, then the SQL statement is built to delete the post from the database. The SQL statement is executed, and the Threads *ChildPosts* property is reinitialized by calling its *InitializePosts* method.

Designing the *ThreadList* Class

We promised you that we would discuss the *ThreadList*, and here it is. As was mentioned earlier, the *ThreadList* class is a class that wraps an ArrayList. By *wraps*, we mean it contains a private *ArrayList* thereby holding its list of Threads, and exposes certain custom functionalities not necessarily pre-built into the *ArrayList* class. Let's take a look at a UML diagram for the *ThreadList* class in Figure 8.26.

Figure 8.26 The *ThreadList* Class



As you can see from this diagram, there isn't much to the *ThreadList*. It contains a count of the number of Threads in the list, contains an *Item* property to allow you to access the Threads in the list, and gives you the ability to manually force the reinitialization of the list through the *InitializeThreads* method. Again, let's start at the basics and build up from there in Figure 8.27. The complete source code for ThreadList.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.27 The Basics (ThreadList.vb)

```
Public Class ThreadList
    Private list As ArrayList
    Private mBoardID As Long

    Public Sub New(ByVal BoardID As Long)
        mBoardID = BoardID
        Me.InitializeThreads()
    End Sub

    Public ReadOnly Property Count() As Integer
        Get
            Return list.Count
        End Get
    End Property
End Class
```

The *ThreadList* class contains only two private fields: *list* and *mBoardID*. The list variable is used to hold all your Threads, and *mBoardID* is used to look up the Threads in a given Board. The constructor accepts a *BoardID*, and calls the *InitializeThreads* method, as shown in Figure 8.28. The complete source code for ThreadList.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.28 The *InitializeThreads* Method (ThreadList.vb)

```
Public Sub InitializeThreads()
    Dim myData As DataSet
    Dim sql As String
    sql = "SELECT [Threads].*, [Users].* FROM [Threads] " & _
        "INNER JOIN [Users] " & _
        "ON [Users].[UserID] = [Threads].[CreatorID] " & _
        "WHERE " & _
        "[BoardID] = " & mBoardID.ToString() & _
        " ORDER BY [Threads].[ThreadID] DESC"
    myData = DataControl.GetDataSet(sql)
```

Continued

Figure 8.28 Continued

```
    list = New ArrayList()

    Dim myRow As DataRow
    For Each myRow In myData.Tables(0).Rows
        list.Add(myRow)
    Next
End Sub
```

The *InitializeThreads* method is rather straightforward, but there is one major concept that needs to be mentioned. First, a SQL statement is built to select the Threads located in the appropriate Board (this is where the mBoardID variable comes into play). The SQL statement also joins on the Users table, to allow for the *Thread* object to know about the user who created the Thread. Next, the list is initialized, and each *DataRow* in the resultant *DataSet* is added to the list. This is where the important concept is. The private list currently contains a set of *DataRow* objects. Obviously, you do not want to expose a bunch of *DataRow* objects as your list of Threads, so this is where the *Item* property comes into effect. See Figure 8.29 regarding the *Item* property. The complete source code for ThreadList.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.29 The *Item* Property (ThreadList.vb)

```
Public Function Item(ByVal index As Integer) As Thread
    Dim myObject As Object = list.Item(index)
    If myObject.GetType() Is GetType(Thread) Then
        'it is already a thread, so nothing further is needed
    Else
        Dim myThread As Thread
        myThread = New Thread(CType(list.Item(index), DataRow))
        'replace the item in the list with
        'an actual thread object
        list.Item(index) = myThread
    End If

    Return CType(list.Item(index), Thread)
End Function
```

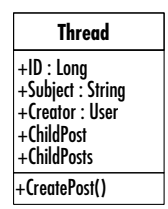
The *Item* property is a little more complex than the average property. Let's review it, line by line. Line 1 creates a variable called *myObject* of type *Object* and sets it equal to the object that is at the specified index of the *ArrayList*. Line 2 compares the type of the object to the type of the *Thread* class. If they are the same, it does nothing; if not, it enters the *Else* part of the *If* statement (lines 5 through 9). Next, a *Thread* variable called *myThread* is declared and set to a new Thread on line 6, passing in the object that is in the specified index in the *ArrayList*. That object is cast to a *DataRow* using CType. Line 9 sets the object at the specified index in the *ArrayList* to the *myThread* variable. Finally, on line 12 it returns the Thread that is in the specified index of the *ArrayList* (and again, is cast to be a *Thread* object).

You might be wondering exactly what all of this accomplishes. Well, if you remember from the *InitializeThreads* method, the *ArrayList* is filled with *DataRow* objects. We do not want to directly expose anyone using our objects to *DataRow* objects, so we need to instead give them *Thread* objects. So, behind the scenes, every time a new index is requested from the *ArrayList*, we quietly "switch" the variable in that index from a *DataRow* to the appropriate *Thread* object. You might also ask why this class doesn't just put the Threads into the *ArrayList* from the start instead of doing it this way. The answer is simple: there is no need for the overhead of having multiple *Thread* objects (each with other objects inside them) in the list when you can save memory and time instantiating objects by just keeping the data for each *Thread* object until it is actually requested. When developing large-scale applications with many parent-child hierarchical relationships, a technique like this will save you and your application a lot of time.

Designing the *Thread* Class

The *Thread* class is the "middle child" in our hierarchy of objects. Luckily for us, much of its functionality and concepts are borrowed directly from the *Board* class, so this should be pretty quick. Let's take a look at another UML diagram in Figure 8.30.

Figure 8.30 The *Thread* Class



Like every class we've examined so far, the *Thread* class shares the same private fields as the Thread table in the database. Like the *Board* class, the *Thread* class contains two properties to access its children: *ChildPost* and *ChildPosts*. *ChildPost* retrieves an individual *Post* object from its list, and *ChildPosts* returns the entire *PostList*. *PostList* will be discussed a bit later. Thread also contains the method to create child Posts. Let's start with the basics in Figure 8.31. The complete source code for Thread.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.31 The Basics (Thread.vb)

```
Public Class Thread
    Private mThreadID As Long
    Private mSubject As String
    Private mCreator As User
    Private myPosts As PostList

    Public Sub New(ByVal myRow As DataRow)
        inflate(myRow)
    End Sub

    Private Sub inflate(ByVal myRow As DataRow)
        mSubject = CStr(myRow("ThreadSubject"))
        mThreadID = CLng(myRow("ThreadID"))
        mCreator = New User(myRow)
        myPosts = New PostList(mThreadID)
    End Sub

    Public ReadOnly Property ChildPost(ByVal postId As Long) _
        As Post
        Get
            'lookup the correct Post
            Dim i As Integer
            For i = 0 To Me.ChildPosts.Count - 1
                Dim myPost As Post = Me.ChildPosts.Item(i)
                If myPost.ID = postId Then
                    Return myPost
```

Continued

Figure 8.31 Continued

```
                End If
            Next i
            'if we've gotten to this point, there is no Post
            'with that ID in this board. throw an exception
            Throw New ArgumentException("Post does not exist")
        End Get
    End Property

    Public ReadOnly Property ChildPosts() As PostList
        Get
            Return myPosts
        End Get
    End Property

    Public ReadOnly Property ID() As Long
        Get
            Return mThreadID
        End Get
    End Property

    Public Property Subject() As String
        Get
            Return mSubject
        End Get
        Set(ByVal Value As String)
            mSubject = Value
        End Set
    End Property

    Public ReadOnly Property Creator() As User
        Get
            Return mCreator
        End Get
    End Property
End Class
```

First, you'll notice the private fields that are the same as the fields in the database. You'll also notice that a Thread has a *Creator* field and property that are *User* objects representing the user that created this Thread. Like the *Board* class, this class has a constructor that accepts a *DataRow* as a parameter and then calls inflate to fill the private fields using that *DataRow*. Also, like Board, you have two child object properties, *ChildPost* and *ChildPosts*. *ChildPost* is used to return a single Post, and *ChildPosts* is used to return the entire *PostList*. Let's take a look at the next method in the Thread class, *CreatePost*, in Figure 8.32. The complete source code for Thread.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.32 The *CreatePost* Method (Thread.vb)

```
Public Sub CreatePost(ByVal subject As String, _
    ByVal body As String, _
    ByVal creator As User)

    Dim sql As String
    sql = "INSERT INTO [Posts] ([PostSubject], " & _
        "[PostBody], " & _
        "[CreatorID], [ThreadID]) VALUES ('" & subject & _
        "','" & body & "'," & creator.ID.ToString() & "," & _
        mThreadID.ToString() & ")"
    DataControl.ExecuteNonQuery(sql)

    'reinitialize the thread list
    myPosts.InitializePosts()
End Sub
```

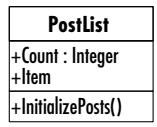
Looking at the *CreatePost* method, you'll notice that it does almost exactly what *CreateThread* did in the Board class. It builds a SQL statement to create a new Post, then executes that statement and reinitializes the private *PostList* object.

Designing the *PostList* Class

Since we're almost finished creating our classes, it's time to look at the *PostList* class. You might be thinking, "I wonder if the *PostList* class is similar to the *ThreadList* class." Such thinking should be rewarded. *PostList* and *ThreadList* are

nearly identical, except in regard to what type of object they collect. Again, let's take a look at the UML diagram for the class first in Figure 8.33; then, in Figure 8.34, we'll review the basics of this class (he complete source code for PostList.vb is available on the companion Solutions Web site for the book).

Figure 8.33 The *PostList* Class



SYNGRESS
syngress.com

Figure 8.34 The Basics (PostList.vb)

```
Public Class PostList
    Private list As ArrayList
    Private mThreadID As Long

    Public Sub New(ByVal ThreadID As Long)
        mThreadID = ThreadID
        Me.InitializePosts()
    End Sub

    Public Sub InitializePosts()
        Dim myData As DataSet
        Dim sql As String
        sql = "SELECT [Users].*, [Posts].* FROM " & _
            "[Posts] INNER JOIN [Users] " & _
            "ON [Users].[UserID] = [Posts].[CreatorID] " & _
            "WHERE " & _
            "[ThreadID] = " & mThreadID.ToString() & _
            " ORDER BY PostDate DESC"
        myData = DataControl.GetDataSet(sql)

        list = New ArrayList()

        Dim myRow As DataRow
```

Continued

Figure 8.34 Continued

```
        For Each myRow In myData.Tables(0).Rows
            list.Add(myRow)
        Next
    End Sub

    Public ReadOnly Property Count() As Integer
        Get
            Return list.Count
        End Get
    End Property
End Class
```

Just like *ThreadList*, *PostList* contains a Count property, a method to initialize posts in a thread, and a constructor that accepts the ID of the parent object. The only real difference here is that this class gets values from the User table instead of the Thread table. Next, let's examine the *Item* function in Figure 8.35. The complete source code for PostList.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.35 The *Item* Function (PostList.vb)

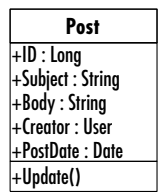
```
Public Function Item(ByVal index As Integer) As Post
    Dim myObject As Object = list.Item(index)
    If myObject.GetType() Is GetType(Post) Then
        'it is already a post, so nothing further is needed
    Else
        Dim myPost As Post
        myPost = New Post(CType(list.Item(index), DataRow))
        'replace the item in the list with
        'an actual post object
        list.Item(index) = myPost
    End If

    Return CType(list.Item(index), Post)
End Function
```

In reviewing this *Item* function, note that it looks remarkably similar to the *Item* function in the *ThreadList* class. In fact, it is exactly the same except that it uses Post instead of Thread. Other than that difference, *PostList* is exactly the same as *ThreadList*.

Designing the *Post* Class

So far, you should have noticed that most of the classes in our code share many of the same ideas: add, update, lists, mimicking the database tables. Well, the *Post* class is no different. In fact, it is rather similar to both the *Board* and *Thread* classes. Let's take a look at the UML diagram for this class in Figure 8.36.

Figure 8.36 The *Post* Class



Just like the other classes, this one is remarkably similar to its brothers—especially the *Thread* class. The only real difference between this class and the *Thread* class is that *Post* has a Body field, pulls its values from the Post table, and doesn't have any child objects. Let's take a look at the whole class in Figure 8.37. The complete source code for Post.vb is available on the companion Solutions Web site for the book.

SYNGRESS
syngress.com

Figure 8.37 Post.vb

```
Public Class Post
    Private mPostID As Long
    Private mPostSubject As String
    Private mPostBody As String
    Private mCreator As User
    Private mPostDate As Date

    Public Sub New(ByVal myRow As DataRow)
        inflate(myRow)
```

Continued

Figure 8.37 Continued

```
End Sub

Public Sub Update(ByVal requestor As User)
    If requestor.ID = mCreator.ID Then
        Dim sql As String
        sql = "UPDATE [Posts] SET [PostSubject] = '" & _
            mPostSubject & "', [PostBody] = '" & mPostBody & _
            "' WHERE [PostID] = " & mPostID.ToString()
        DataControl.ExecuteNonQuery(sql)
    Else
        Throw New ArgumentException _
            ("Only the creator of a post can update it")
    End If
End Sub

Private Sub inflate(ByVal myRow As DataRow)
    mPostID = CLng(myRow("PostID"))
    mPostSubject = CStr(myRow("PostSubject"))
    mPostBody = CStr(myRow("PostBody"))
    mCreator = New User(myRow)
    mPostDate = CDate(myRow("PostDate"))
End Sub

Public ReadOnly Property ID() As Long
    Get
        Return mPostID
    End Get
End Property

Public Property Subject() As String
    Get
        Return mPostSubject
    End Get
```

Continued

Figure 8.37 Continued

```
        Set(ByVal Value As String)
            mPostSubject = Value
        End Set
    End Property

    Public Property Body() As String
        Get
            Return mPostBody
        End Get
        Set(ByVal Value As String)
            mPostBody = Value
        End Set
    End Property

    Public ReadOnly Property Creator() As User
        Get
            Return mCreator
        End Get
    End Property

    Public ReadOnly Property PostDate() As Date
        Get
            Return mPostDate
        End Get
    End Property
End Class
```

As you can see, this class has five private fields with the corresponding five public properties. In addition, it has a constructor that accepts a *DataRow* parameter that passes the *DataRow* to the *inflate* method. Finally, it has an *update* method, with the rule that only the creator of the Post can actually edit the Post. Doesn't seem too hard, does it? Especially after all the other classes we've dealt with, it almost seems passé.

Designing the *MessageBoard* Class

We've finally gotten every class in our message board object library finished; now all we need is a way to get a list of every *Board* object from our database. This is accomplished using the *MessageBoard* class. We won't bother to show you a UML diagram of the *MessageBoard* class, as there is only one method in it: *GetBoards*. Let's take a look at the code in Figure 8.38. The complete source code for MessageBoard.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.38 MessageBoard.vb

```
Public Class MessageBoard
    Public Shared Function GetBoards() As ArrayList
        Dim list As New ArrayList()
        Dim sql As String
        Dim myData As DataSet
        Dim myRow As DataRow

        sql = "SELECT [BoardName] FROM [Board] ORDER BY [BoardName] Asc"
        myData = DataControl.GetDataSet(sql)

        For Each myRow In myData.Tables(0).Rows
            Dim myBoard As Board
            myBoard = New Board(CStr(myRow("BoardName")))
            list.Add(myBoard)
        Next myRow

        Return list
    End Function
End Class
```

This class is fairly easy to understand. What it does is look up each *BoardName* from the database, and create a new *Board* object based on that name. It then adds each Board to its list, and finally returns the list.

That's it. Every single one of our objects to be used in *dotBoard* is completely finished. You might wonder why we did all this work ahead of time,

instead of just jumping into the application itself. That is a very good question, and as such, has a very good answer. We did all this work designing and setting things up so that when we actually build our application, it will go smoothly, quickly, and won't require a lot of coding in the user interface. Any good application splits the UI from the actual implementation of the application, which is exactly what we did. We are about to move on to the user interface of our message board application. You will see that using the work we've already done, the rest of this application is going to be very straightforward and easy.

Designing the User Interface

Finally, we've gotten to our user interface. Our database is constructed; all of our message board classes are created; so the final thing to do is to put a UI on top of

Developing & Deploying…

Copying ASP.NET Applications to Multiple Computers

If you are using the examples on the companion Solutions Web site for the book (www.syngress.com/solutions), please perform the following steps to get your ASP.NET message board up and running on your computer.

1. Copy the files from the Web site to a folder underneath your WWWRoot folder, typically located at C:\Inetpub\WWRoot. Name this folder **dotBoardUI**.
2. Open the **Internet Services Manager** from **Administrative Tools** in the **Control Panel**.
3. Expand the Internet Information Services node, then your computer's node, and finally the Default Web Site node.
4. Find your dotBoardUI folder. Right-click and select **Properties** to bring up the Properties pane.
5. Look at the **Application Settings** panel, and click the **Create** button next to the grayed-out Application Name label and text box.
6. Click **OK**.

it all. Just as when we created the classes our applications are going to use, we need to sit and think for a few minutes to determine exactly what it is our message board will do. The obvious requirements are that a user must be able to register, log in, and modify his or her profile. Anyone must be able to browse the Boards, Threads, and Posts. Registered Users must be able to create Threads and Posts, and Administrators must have the ability to administer users and create and delete Boards.

Sound like a lot? Well, since we have a good majority of this work already built into our numerous classes, most of our work now is to create the UI and tie events to methods our objects will handle. The only other thing our message board should be able to do is be "changed" at will. That is, colors, fonts, and any other type of styling element should be able to be changed without needing to actually modify every single control we place on our form. This will be discussed in a moment, but for now, rest assured, it will be very exciting, and most of the work will be done for us! Let's start by figuring out how to register and log in.

Setting Up General Functions

The first step in designing our application is to create the ASP.NET application. You can either get the solution from the Solutions Web site, or create your own. If you get the files from the companion web site, they are in a folder called "dotBoardUI" in the folder for Chapter 8 in the source code portion for this book on the companion Solutions Web site (www.syngress.com/solutions). The dotBoard.sln file is the main solution file, and everything else in that folder is a part of the project. Either way, your application should be named **dotBoardUI**, to go with your *dotBoardObjects* class library. After you have created your application, add your **dotBoardObjects** project to your solution, and add a reference to the newly added project to your ASP.NET application. Next, rename **Web Form1.aspx** to **default.aspx**. This will make it easier when it comes time to deploy your application, as default.aspx is typically one of the default documents IIS serves when the browser doesn't request a specific file in your application.

Now that you have your project created and the appropriate references made, let's get started on the groundwork for our application. If you think about it, every page you make will likely need access to the currently logged-in user. There are many reasons for this, as you'll see later, so for now just assume that every page will need that information. There are many ways to do this. For

example, you can copy and paste the code necessary to get this information on every page. Anyone familiar with programming techniques should sense a red flag go up at that statement. Copying and pasting the code is a terrible idea, for so many reasons that we don't have space to state them here. Another solution available is to create a public module with the common functions your pages would need. This is a good solution, but let's do it a little differently. We are going to have one Web Form from which all our Web Forms will inherit. Why would we do this? So every Web Form you create will have direct access to the common methods, and every user control you put on these Web Forms will be able to get the information easily.

Add a new class to your project and name it *FormBase.vb*. We're not adding a Web Form in this case, because we don't need any type of UI for our *FormBase*; we just need access to a common set of methods. Take a look at the basic code in Figure 8.39 (which can also be found on the companions Solutions Web site (www.syngress.com/solutions) for this book in the file FormBase.vb).

Figure 8.39 The Basics (FormBase.vb)

```
Public Class FormBase
    Inherits System.Web.UI.Page
End Class
```

Pretty easy, right? What we have here is a class that inherits from *System.Web.UI.Page*. This allows all our Web Forms to inherit directly from this class, instead of inheriting from *System.Web.UI.Page*. The next thing we need is for our *FormBase* to be able to have a reference to the currently logged in user (if there is one). Here is the code to do just that in Figure 8.40. The complete source code for FormBase.vb is available on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.40 Maintaining the Current User (FormBase.vb)

```
Private mCurrentUser As dotBoardObjects.User

Public Property CurrentUser() As dotBoardObjects.User
    Get
        Return mCurrentUser
    End Get
```

Continued

Figure 8.40 Continued

```
    Set(ByVal Value As dotBoardObjects.User)
        mCurrentUser = Value
        'add the user's ID to the session
        Session.Add("userid", Value.ID.ToString())
    End Set
End Property

Public ReadOnly Property IsLoggedIn() As Boolean
    Get
        Return Not mCurrentUser Is Nothing
    End Get
End Property
```

All we have here is a private *dotBoardObjects.User* object, and a public property to retrieve it. The *Set* property sets the private field with the value passed in, and adds the user ID of the passed-in *User* object to the session. We do this so a user does not have to log in multiple times while perusing your message board—you'll see where this comes into play later. The other property we have is one that returns a Boolean value of whether or not there is a currently logged-in user. This property makes it easier for someone to determine if there is a logged-in user. Basically, instead of having to test for **Nothing** over and over, you use this *Boolean* property.

That is all the state maintaining we'll need in our base class. The only other thing our *FormBase* class needs to do is fulfill that last requirement we talked about; that is, the ability to modify every control on every form without needing to actually rename the class names on elements. This is probably one of the most interesting techniques dotBoard will use. Basically, what we will do is create the code necessary to automate the process of restyling every control in every Web Form. This might sound like a daunting task, but actually once you take a look at it, it is rather simple. The first step we need to take is to open up our web.config file and add the following lines of XML directly beneath the *<configuration>* tag as shown in Figure 8.41 (which can also be found on the companion Solutions Web site for the book in the file named web.config).

Figure 8.41 The web.config File

```
<appSettings>
    <add key="ConnectionString"
value="Provider=Microsoft.Jet.OLEDB.4.0;
        DataSource="C:\Location\To\Your\database\dotBoard.mdb;
        User ID=Admin;Password=;" />
    <add key="XmlConfigFile"
        value="C:\Inetpub\WWWRoot\dotBoardUI\styles.xml" />
</appSettings>
```

Okay, now what exactly does that mean? Your *<appSettings>* are custom settings you create and have access to in your application. We are creating two custom settings, which are added using the *<add>* tag. The *key* attribute is the name of the settings, and the *value* attribute is obviously the value. Here we are adding two keys, *ConnectionString* and *XmlConfigFile*. *ConnectionString* is what you use to connect to your database. Remember the *DataControl* class and how it accessed *System.Configuration.ConfigurationSettings*? The *ConnectionString* key is exactly what that class will use. The other key is *XmlConfigFile*, which is used to hold the location to your XML file that will hold the style information we discussed earlier. Please change the values of each to represent where you actually have the files on your computer located.

We now have the *ConnectionString* and *XmlConfigFile* keys added to our AppSettings. Let's start discussing how we will accomplish the "sweeping" change of styles, without needing to manually apply any styles on your controls. First take a look at the following Cascading Style Sheet (CSS) file you should add to your project, as shown in Figure 8.42. This file can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.42 Styles.css

```
body
{
     font-family:Tahoma, Arial, Sans-Serif;
     font-size:10pt;
     color:#000000;
}
.errors
```

Continued

Figure 8.42 Continued

```
{
      font-family:Tahoma, Arial, Sans-Serif;
      font-size:10pt;
      color:#993300;
}
.link
{
      text-decoration:underline;
      font-family:Tahoma, Arial, Sans-Serif;
      color:#FF9933;
}
.header
{
      color:#003399;
      font-size:16pt;
      font-weight:bold;
      font-family:Arial, Sans-Serif;
}
.panel
{
      border: 1px solid #000000;
      padding: 10px;
}
.inputBox
{
      border: 1px solid #000000;
      background-color:#e5e5e5;
}
.label
{
      font-family:Tahoma;
      font-size:8pt;
      color:#000000;
}
```

Continued

Figure 8.42 Continued

```
.button
{
      border: 1px solid #000000;
      background-color: #FF9933;
      color: #000000;
      font-family: Arial, Sans-Serif;
      font-size: 10pt;
}
```

You can see here that we have a number of styles we will want to apply to many different elements throughout our application. Manually setting these styles is hardly desirable, and maintaining these settings if any of your class names change would be a nightmare. So, what can be done to prevent us from having to maintain this? Enter the styles.xml file in Figure 8.43. This file can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.43 Styles.xml

```
<?xml version="1.0" encoding="utf8"?>
<styles>
    <control type="System.Web.UI.WebControls.Label">label</control>
    <control type="System.Web.UI.WebControls.TextBox">inputBox</control>
    <control type="System.Web.UI.WebControls.Button">button</control>
    <control type="System.Web.UI.WebControls.Panel">panel</control>
    <control type="System.Web.UI.WebControls.LinkButton">link</control>
    <control type="System.Web.UI.WebControls.ValidationSummary">
        errors</control>
</styles>
```

You should now notice that the values of these XML tags correspond to an appropriate class name in the preceding stylesheet declaration. Now all we need to do is find a way to associate these XML tags with the appropriate controls on every page. We can accomplish this through two methods, as shown in Figure 8.44.

This file can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.44 Two Methods to Dynamically Apply Styles to Controls (Board.vb)

```
Public Sub ApplyStyles(ByRef objControls As ControlCollection)
    If objXml Is Nothing Then
        Dim xmlLoc As String
        xmlLoc = ConfigurationSettings.AppSettings()("XmlConfigFile")
        objXml = New XmlDocument()
        Try
            objXml.Load(xmlLoc)
        Catch E As Exception
            Throw New Exception("XML Style Config file not found")
        End Try
    End If

    Dim objControl As Control
    For Each objControl In objControls
        Dim style As String
        style = GetStyleName(objControl.GetType.ToString())

        If style <> "" Then
            Dim objWebControl As WebControl
            objWebControl = CType(objControl, WebControl)
             'we only want to apply these styles if we
            'haven't already explicitly set them
            If objWebControl.CssClass.Trim() = "" Then
                objWebControl.CssClass = style
            End If
        End If
        If objControl.HasControls() Then
            ApplyStyles(objControl.Controls)
        End If
    Next objControl
End Sub
```

Continued

Figure 8.44 Continued

```
Public Function GetStyleName(ByVal controlType As String) As String
    Dim objNode As XmlNode
    objNode = objXml.SelectSingleNode("styles/control[@type='" & _
        controlType & "']")
    If objNode Is Nothing Then
        'do nothing
        Return ""
    Else
        'get the css class specified by this node
        Return objNode.InnerText
    End If
End Function
```

That's a lot to digest all at once, so let's break it down. The first thing you'll see is that *ApplyStyles* accepts a *ControlCollection* as a parameter. This collection can be obtained from *Page.Controls* or *Control.Controls*. Next, the subroutine checks to see if the XML document has been loaded yet. If it hasn't, it retrieves the location of the styles.xml file from the *AppSettings* and loads it. If there was an error in the loading of the document, it throws an exception. If there are no problems with the XML document, it loops through every control in the *ControlCollection* that was passed in. For every control, it sets a variable "style" to the value of what the *GetStyleName* function returns. *GetStyleName* takes your control's fully qualified type name (represented in the code by *objControl.GetType().ToString()*), and looks for that in the XML document. It does this by calling the *SelectSingleNode* function of the *XMLDocument* object. It builds an XPath query string and looks for the appropriate node with the type attribute that is the same as the type string passed into the *GetStyleName* function. If it finds that node, it returns the *InnerText* of the appropriate node; otherwise, it returns an empty string.

Control is returned to the *ApplyStyles* method, and the style that was returned is tested to make sure it is not an empty string; there is no point in setting the value if it is empty. Next, the Control is cast to be a variable of type *WebControl*. Since the only Control that can have its style attribute programmatically manipulated is the *WebControl*, and since every control in *System.Web.UI.WebControls* inherits directly from *WebControl*, it is safe to perform this cast. Just make sure you do not add anything other than *WebControls* to your styles.xml file, and this

will work without error. Next, the *CssClass* property of your *WebControl* is tested to make sure it is currently an empty string. It does this because if you specifically set a style on one of your Controls, you most likely do not want that style overridden by this method. If it is empty, it sets the *CssClass* property to the style String that was returned by the *GetStyleName* function. Finally, if the Control has child controls, it recursively calls *ApplyStyles*, but instead with the *Control.ChildControls ControlCollection* as the parameter.

With these two functions, every type of Control you add to your styles.xml file will automatically get CSS styles applied to them, without any maintenance on your part other than a small XML file. Wondering how this will actually get used? All you need to do is in your classes that inherit from FormBase, call the *ApplyStyles* method passing the *ChildControls* of the page you are currently on. Feel free to try this. Modify the stylesheet and styles.xml file all you want. Just rest assured that every control type you add to your XML file will automatically have the CSS classes applied to them that you want.

Building the Log-In Interface

Since we don't have any users created in the database yet, let's take a look at how to register with dotBoard. How to create the User Controls and Forms won't be discussed, but the source code is available on the companion Solutions Web site for the book (www.syngress.com/solutions), as well as multiple screen shots for each Web Form and User Control. Take a look at the register.aspx page in Figure 8.45.

Let's examine the controls on this page. First, there are a number of labels and text boxes used to capture the user's information. There is also a button that will submit the form when pressed. The red-colored controls are validation controls. Validation controls allow you to place "rules" on input without needing to actually code it yourself. The display property of these controls is set to *None*, so they will never show up, but that is where the *ValidationSummary* comes in. The control in the top right of this page is a *ValidationSummary* control, which will aggregate all the errors into one area, so you do not need to place your validation controls in a custom place. The other thing on this form is a *CustomValidator* control. A *CustomValidator* is typically used to handle client-side JavaScript, but it is also quite useful to handle exceptions thrown and display them to the user. Let's take a look at the code behind this form in Figure 8.46. Register.aspx.vb can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.45 The Register.aspx Page

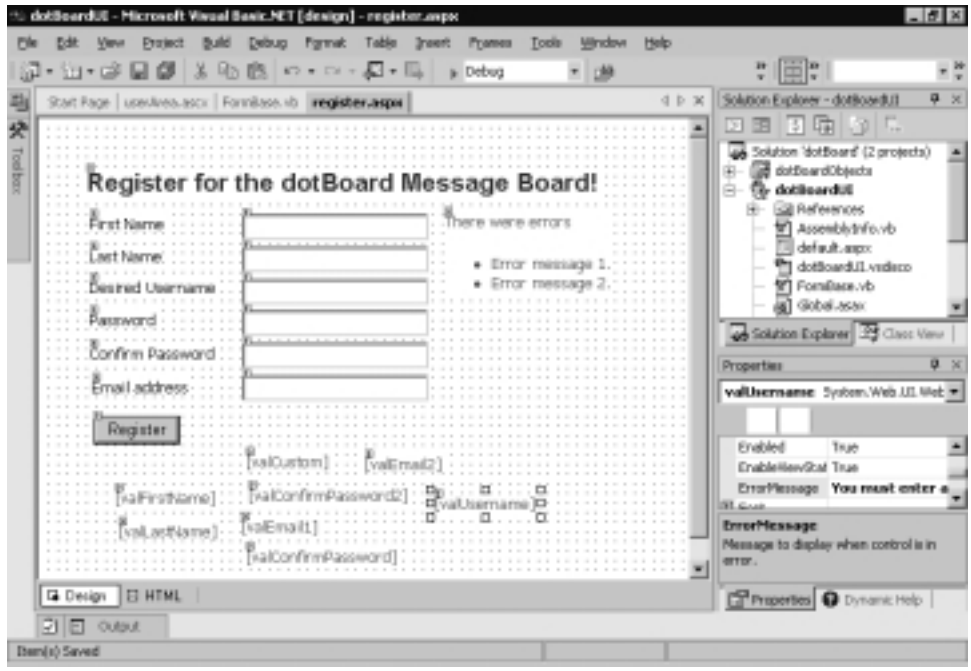


Figure 8.46 The Code-Behind File (Register.aspx.vb)

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    'Put user code to initialize the page here
    Me.ApplyStyles(Me.Controls)
End Sub

Private Sub btnRegister_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnRegister.Click
    'attempt to register the user
    If Me.Page.IsValid Then
        Try
            Dim myUser As dotBoardObjects.User
            myUser = dotBoardObjects.User.CreateUser( _
                txtUsername.Text, txtPassword.Text, _
                txtFirstName.Text, txtLastName.Text, _
```

Continued

Figure 8.46 Continued

```
            txtEmailAddress.Text)
            'if we've made it this far, the create worked
            Dim objPage As FormBase
            objPage = CType(Me.Page, FormBase)
            objPage.CurrentUser = myUser
            'redirect to the default page
            Response.Redirect("default.aspx")
        Catch Ex As Exception
            valCustom.ErrorMessage = Ex.Message
            valCustom.IsValid = False
        End Try
    End If
End Sub
```

First, we have the **Page_Load** subroutine, which handles the *Page.Load* event. All this event does is call the *ApplyStyles* method of the *FormBase* class. Next, we have the **btnRegister_Click** subroutine that handles the Register button's *click* event. The first thing that subroutine does is make sure the page is currently in a valid state. This validity is determined whether or not all of the validation controls you added to your form return a valid result. Only once every validation control becomes valid does *Page.IsValid* ever return true. Next, a *User* object is declared and the *CreateUser* method is called. If the *CreateUser* method throws an exception, then the custom validator on our form is set to invalid and its *ErrorMessage* property is set to the *Message* property of the Exception thrown. If the *CreateUser* succeeded, then a reference to the parent Page, casted to the *FormBase* type, is created and the *CurrentUser* property is set to the User that was just created. Once all this is done, the user is redirected to default.aspx.

As we discussed when we went over *FormBase*, every page will need to know about the currently logged-in user. Likely, every page will also need a login form so the user can log in from anywhere. The best way to do this is to create a Web User Control. Take a look at our *userArea.ascx* control in Figure 8.47.

Boy, that's ugly, isn't it? Don't worry, that's why we created the style code in FormBase. Anyway, what we have here are two panels. The top panel contains the controls necessary to log a user in, while the bottom panel contains the welcome message and any specific actions the user can take. Let's take a look at

the code-behind for this page in Figure 8.48. The complete source code for Figure 8.48 can also be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.47 UserArea.ascx

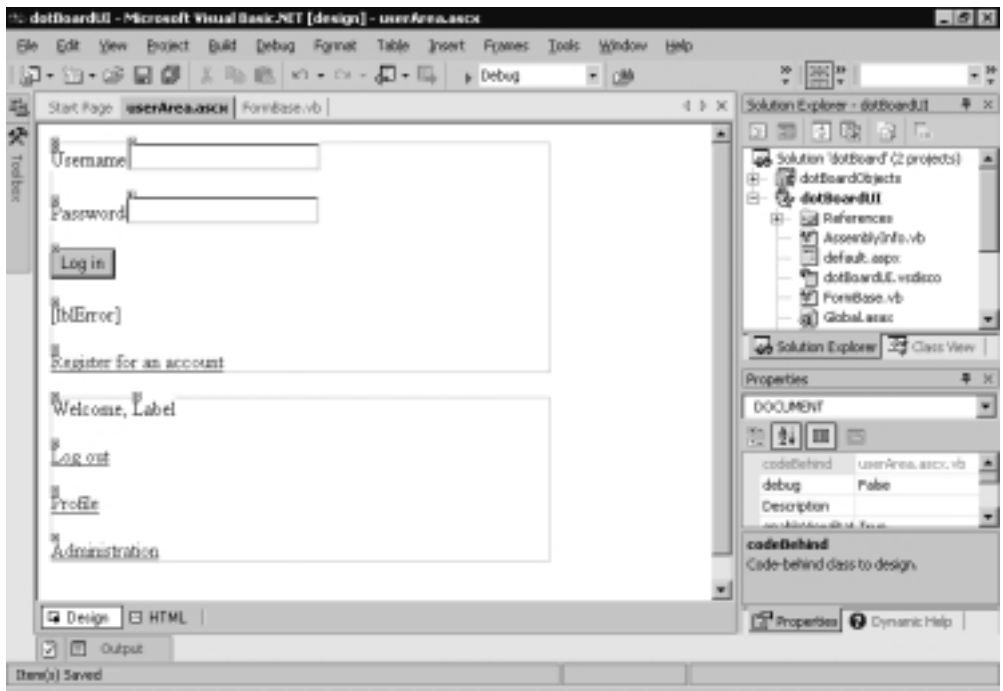


Figure 8.48 The Code-Behind (UserArea.ascx.vb)

```
Private Sub Page_Init(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Init
    'CODEGEN: This method call is required by the Web Form Designer
    'Do not modify it using the code editor.
    InitializeComponent()

    pnlNotLoggedIn.Visible = True
    pnlLoggedIn.Visible = False
    lnkAdmin.Visible = False

    'attempt to log the user in
```

Continued

Figure 8.48 Continued

```
    If Not Session.Contents().Item("userid") Is Nothing Then
        Dim userId As Long
        userId = CLng(Session.Contents.Item("userid"))
         Dim myUser As User
        Try
            myUser = New User(userId)
            Dim objPage As FormBase
            objPage = CType(Me.Page, FormBase)
            objPage.CurrentUser = myUser
            pnlNotLoggedIn.Visible = False
            pnlLoggedIn.Visible = True

            lblWelcome.Text = myUser.FirstName & " " & myUser.LastName

            If myUser.IsAdmin Then
                lnkAdmin.Visible = True
            End If
        Catch Ex As Exception
            lblError.Text = Ex.Message
        End Try
    End If
End Sub

Private Sub btnLogIn_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnLogIn.Click
    'attempt to log in the user
    If txtUsername.Text.Trim() <> "" And _
        txtPassword.Text.Trim() <> "" Then
        Try
            Dim myUser As User = User.Validate(txtUsername.Text, _
                txtPassword.Text)
            Dim objPage As FormBase
            objPage = CType(Me.Page, FormBase)
            objPage.CurrentUser = myUser
            'if it got this far it succeeded
```

Continued

Figure 8.48 Continued

```
            'redirect, to allow the whole page to refresh
            Response.Redirect(Request.RawUrl)
        Catch Ex As Exception
            lblError.Text = Ex.Message
        End Try
    End If
End Sub

Private Sub LinkButton1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles LinkButton1.Click
    'redirect to the register page
    Response.Redirect("register.aspx")
End Sub

Private Sub lnkLogOut_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lnkLogOut.Click
    Session.Remove("userid")
    Response.Redirect("default.aspx")
End Sub

Private Sub lnkProfile_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lnkProfile.Click
    Response.Redirect("profile.aspx")
End Sub

Private Sub lnkAdmin_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lnkAdmin.Click
    Response.Redirect("admin.aspx")
End Sub
```

Okay, there's a lot here, so let's break it down. The *Page_Init* subroutine handles the *Page.Init* event. When this subroutine is called, it attempts to log in the user based on the Session *userId* value. If that value exists, it uses it and initializes the *CurrentUser* object; otherwise, it exits. Finally, the subroutine hides or shows the correct panel and admin link depending on whether the user was successfully

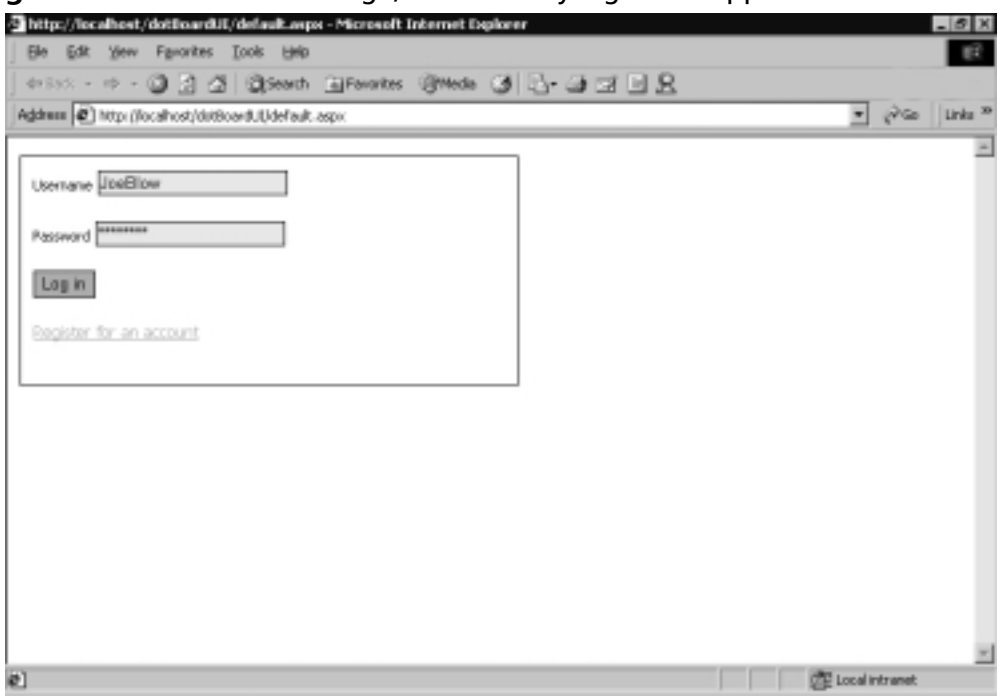
logged in or not, and if the use is an admin or not, and then changes the text of the welcome label to the logged-in user's first and last name.

BtnLogin_Click handles the event when the user clicks the Login button. The first thing it does is check to make sure values have been entered in the username and password fields. If so, it attempts to validate the user with the username and password the user entered. If an exception is thrown, the error label text is set to the message of the exception thrown. If not, it sets the *CurrentUser* property of the *FormBase* to the currently logged-in user, and then redirects the user back to the page he or she is currently on. It does this to make sure all controls on the page have gotten a chance to know that the user has logged in.

Finally, we have four link buttons, the first one redirects the user to the register page we've already seen, while the other clears the user ID out of Session and redirects them back to default.aspx. The third redirects the user to profile.aspx, the user profile page. The fourth one redirects the user to admin.aspx, the admin page.

Finally, open up your default.aspx page, and drag your *userArea.ascx* user control onto the page. You now have a fully functioning login/register area to your message board, where anyone can register and log in and receive customized links depending on what type of user they are. See Figure 8.49 to see what the page looks like.

Figure 8.49 The Default Page, with the Styling Code Applied



Designing the Browsing Interface

The next step in building dotBoard is to determine how to browse through the Boards, Threads, and Posts. When a user first enters the site and views the default page, she should be shown a list of Boards and descriptions she can choose to view. This code is located in default.aspx and default.aspx.vb on the companion Solutions Web site (www.syngress.com/solutions).

Board Browsing

Browsing through our boards isn't very difficult; all we need to do is use a *Repeater* control, and create a custom *DataSet* out of our list of Board objects. Unfortunately, the only control we can drag and drop onto a Web Form is a *Repeater* control, and you can't drag controls into the *Repeater*, so we are going to have to look at the actual quasi-HTML that ASP.NET uses and write the repeated content by hand, as shown in Figure 8.50. The complete source code for Figure 8.50 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.50 The *Repeater* Control (Default.aspx)

```
<asp:Panel runat="server">
    <asp:Repeater id="Repeater1" runat="server">
        <HeaderTemplate>
            <div class="header">Available boards</div>
        </HeaderTemplate>
        <SeparatorTemplate>
            <br><br>
        </SeparatorTemplate>
        <ItemTemplate>
            <a href='board.aspx?boardid=
                <%#DataBinder.Eval(Container, "DataItem.BoardName")%>'>
                <%#DataBinder.Eval(Container, "DataItem.BoardName")%>
            </a>
            <br>
            <%#DataBinder.Eval(Container, "DataItem.BoardDescription")%>
        </ItemTemplate>
    </asp:Repeater>
</asp:Panel>
```

The repeater code creates a header template, separator template, and the actual item template. The only thing we haven't discussed thus far is what data source the *Repeater* should use. Since the *Repeater* control requires a real data source (i.e., *DataSet* or something similar), what needs to be done is our list of Boards needs to be "translated" into a *DataSet*. Take a look at the updated code-behind for the default page in Figure 8.51. The complete source code for Figure 8.51 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.51 The Updated Code-Behind (Default.aspx.vb)

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    'Put user code to initialize the page here
    Me.ApplyStyles(Me.Controls)
    Me.DisplayBoards()
End Sub

Private Sub DisplayBoards()
    Dim myBoards As DataSet = New DataSet()
    Dim list As ArrayList
    list = dotBoardObjects.MessageBoard.GetBoards()

    myBoards.Tables.Add("boards")
    Dim myTable As DataTable = myBoards.Tables(0)
    myTable.Columns.Add("BoardName", GetType(String))
    myTable.Columns.Add("BoardDescription", GetType(String))

    Dim i As Integer
    For i = 0 To list.Count - 1
        Dim myBoard As dotBoardObjects.Board
        myBoard = CType(list(i), dotBoardObjects.Board)
        Dim fields(1) As Object
        fields(0) = myBoard.Name
        fields(1) = myBoard.Description

        myTable.Rows.Add(fields)
        myTable.AcceptChanges()
    Next i
```

Continued

Figure 8.51 Continued

```
    myBoards.AcceptChanges()

    Repeater1.DataMember = "boards"
    Repeater1.DataSource = myBoards
    Repeater1.DataBind()
End Sub
```

Notice the addition to the *Page_Load* method in this file. This subroutine now calls the *DisplayBoards* subroutine. *DisplayBoards* restructures the list of Boards into an appropriate form for a *Repeater* control to use. First, it creates a *DataSet* and gets the list of Boards from the *MessageBoard* class. Next, it creates a new table in the *DataSet* and adds three columns to it. Next, it loops through the list of Boards and builds an object array of the fields to add to the *DataSet*. It then adds a new row by passing in the object array to the *Add* method of the *Rows* collection. Finally, it accepts the changes, and forces the *Repeater* control to *DataBind* to the *DataSet*. Look at Figure 8.52 to see what this page looks like.

Figure 8.52 The Default Page with Boards Displayed

Thread Browsing

Once the user has clicked one of the board links from default.aspx, he is taken to board.aspx. This page will be responsible for determining which board was selected and for displaying the appropriate Threads. Displaying the Threads in a Board will function nearly identically to how displaying Boards functioned. Let's take a look at the important quasi-HTML that this page uses in Figure 8.53. The complete source code for Figure 8.53 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.53 The ASPX Code for Board.aspx

```
<table cellpadding="0" cellspacing="0" border="0">
    <asp:Repeater runat="server" id="Repeater1">
        <SeparatorTemplate>
            <tr> <td colspan="2">   </td> </tr>
        </SeparatorTemplate>
        <ItemTemplate>
            <tr>
                <td>
                started by
                <%#DataBinder.Eval(Container, "DataItem.creatorName")%>
                </td>
                <td>
                <%#DataBinder.Eval(Container, "DataItem.postCount")%>
                total posts
                </td>
            </tr>
            <tr>
                <td colspan="2">
                <a href='thread.aspx?
                <%#DataBinder.Eval(Container, "DataItem.threadLink")%>
                '>
               <%#DataBinder.Eval(Container, "DataItem.threadSubject")%>
                </a>
                </td>
```

Continued

Figure 8.53 Continued

```
            </tr>
        </ItemTemplate>
    </asp:Repeater>
</table>
```

The repeater code creates a separator template and the actual item template. It *DataBinds* the appropriate fields in the data source to items in the template. Let's take a look at how we get the data into the data source in Figure 8.54. The complete source code for Figure 8.54 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.54 Board.aspx.vb

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim mBoard As dotBoardObjects.board
    Dim boardId As String
    boardId = Request.QueryString.Item("boardid")

    Dim myLabel As Label
    myLabel = CType(Me.FindControl("lblHeader"), Label)
    myLabel.Text = boardId

    mBoard = New dotBoardObjects.board(boardId)

    Dim myThreads As DataSet
    myThreads = New DataSet()
    myThreads.Tables.Add("threads")

    Dim myTable As DataTable
    myTable = myThreads.Tables(0)

    myTable.Columns.Add("threadLink", GetType(String))
    myTable.Columns.Add("threadSubject", GetType(String))
```

Continued

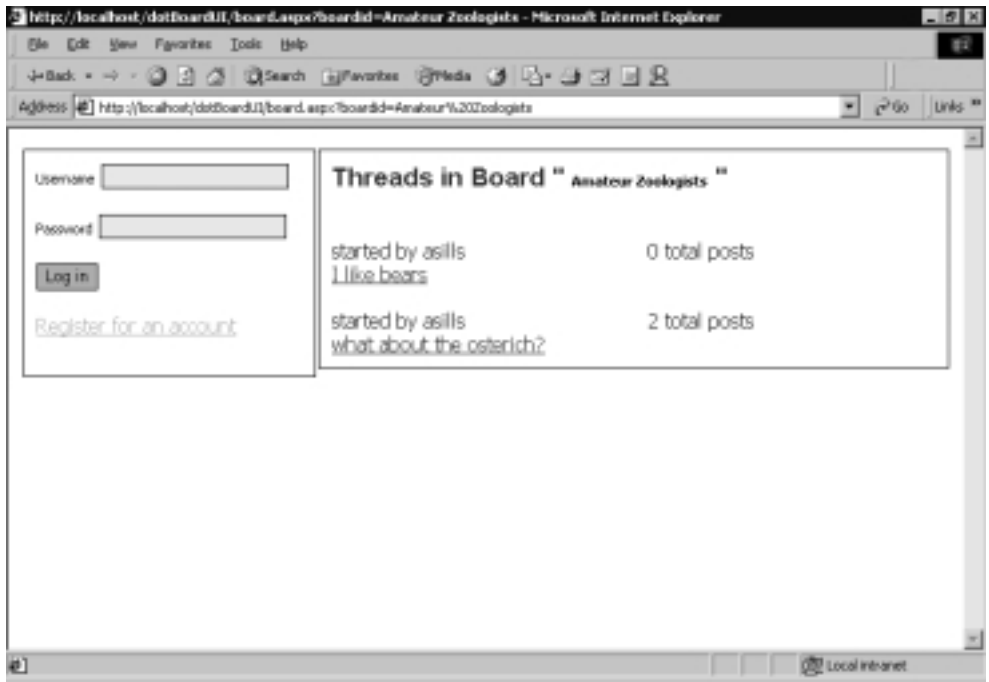
Figure 8.54 Continued

```
    myTable.Columns.Add("postCount", GetType(Integer))
    myTable.Columns.Add("creatorName", GetType(String))

    Dim i As Integer
    For i = 0 To mBoard.ChildThreads.Count - 1
        Dim myThread As dotBoardObjects.Thread
        myThread = mBoard.ChildThreads.Item(i)

        Dim fields(3) As Object
        fields(0) = "BoardId=" & boardId & _
            "&ThreadId=" & myThread.ID.ToString()
        fields(1) = myThread.Subject
        fields(2) = myThread.ChildPosts.Count
        fields(3) = myThread.Creator.Username

        myTable.Rows.Add(fields)
        myTable.AcceptChanges()
    Next i

    myThreads.AcceptChanges()

    Repeater1.DataMember = "threads"
    Repeater1.DataSource = myThreads
    Repeater1.DataBind()

    Me.ApplyStyles(Me.Controls)
End Sub
```

Just like default.aspx, the data binding is relatively straightforward. First, we need to get a reference to the current Board. We do this by requesting the Board name from the query string and initializing the Board using it. Next, we set a label's text property to the name of the Board, so the user knows what Board he's in. Then we create a *DataSet*, add a table to it, and add all the required columns. Afterward, we iterate through the Board's child threads and create an object array

to hold the necessary fields to add to the *DataSet*. Finally, we add all the rows to the *DataSet* and force the *Repeater* control to *DataBind*. Take a look at Figure 8.55 to see what the completed page looks like.

Figure 8.55 The Board Page with Threads Displayed



Message Browsing

The last piece to browsing the message board is to see individual Posts themselves. Just like Boards and Threads, displaying this data is accomplished by using a *Repeater* control and a *DataSet*. Let's take a look at the important quasi-HTML and the code-behind in Figures 8.56 and 8.57. The complete source code for Figure 8.56 and 8.57 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.56 Thread.aspx

```
<asp:Repeater runat="server" id="Repeater1">
   <ItemTemplate>
       <tr>
           <td>posted by
```

Continued

Figure 8.56 Continued

```
            <%#DataBinder.Eval(Container, "DataItem.postCreatorName")%>
            <%#DataBinder.Eval(Container, "DataItem.postCreatorEmail")%>
            </td>
            <td>
               posted at
               <%#DataBinder.Eval(Container, "DataItem.postDate")%>
            </td>
        </tr>
        <tr>
            <td colspan="2">
            <b>
            <%#DataBinder.Eval(Container, "DataItem.postSubject")%>

            </b>
            </td>
        </tr>
        <tr>
           <td colspan="2">
             <%#DataBinder.Eval(Container, "DataItem.postBody")%>
             </td>
         </tr>
    </ItemTemplate>
</asp:Repeater>
```

Figure 8.57 Thread.aspx.vb

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    Dim boardId As String
    Dim threadId As Long

    boardId = Request.QueryString.Item("boardId")
    threadId = CLng(Request.QueryString.Item("threadId"))
```

Continued

Figure 8.57 Continued

```
Dim myBoard As dotBoardObjects.board
myBoard = New dotBoardObjects.board(boardId)

Dim myThread As dotBoardObjects.thread
myThread = myBoard.ChildThread(threadId)

lblHeaderBoard.Text = myBoard.Name
lblHeaderThread.Text = myThread.Subject

Dim myPosts As DataSet
myPosts = New DataSet()
myPosts.Tables.Add("posts")

Dim myTable As DataTable
myTable = myPosts.Tables(0)

myTable.Columns.Add("postId", GetType(Long))
myTable.Columns.Add("postSubject", GetType(String))
myTable.Columns.Add("postBody", GetType(String))
myTable.Columns.Add("postDate", GetType(Date))
myTable.Columns.Add("postCreatorName", GetType(String))
myTable.Columns.Add("postCreatorEmail", GetType(String))

Dim i As Integer
For i = 0 To myThread.ChildPosts.Count - 1
Dim myPost As dotBoardObjects.Post
    myPost = myThread.ChildPosts.Item(i)

    Dim fields(5) As Object
    fields(0) = myPost.ID
    fields(1) = myPost.Subject
    fields(2) = myPost.Body
    fields(3) = myPost.PostDate
```

Continued

Figure 8.57 Continued

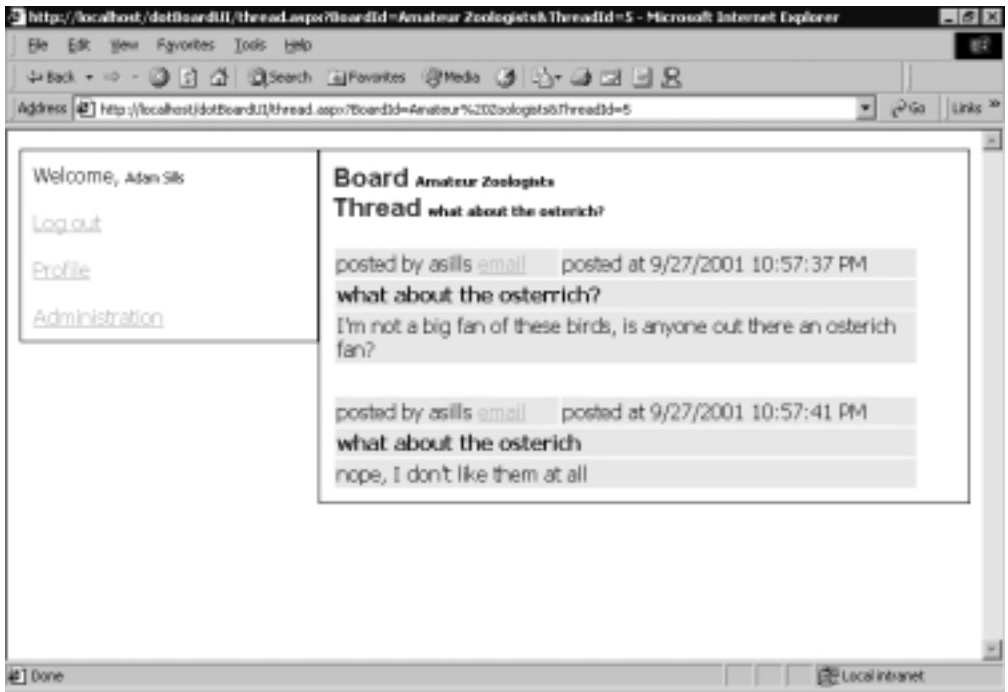
```
        fields(4) = myPost.Creator.Username
        If Me.IsLoggedIn = True Then
            fields(5) = "<a href='mailto:" & myPost.Creator.Email & _
                "'>email</a>"
        Else
            fields(5) = ""
        End If

        myTable.Rows.Add(fields)
        myTable.AcceptChanges()
    Next i

    myPosts.AcceptChanges()

    Repeater1.DataMember = "posts"
    Repeater1.DataSource = myPosts
    Repeater1.DataBind()

    Me.ApplyStyles(Me.Controls)
End Sub
```

Again, this code is nearly identical to that of the last two pages we've dealt with. The only real difference is that one of the fields is actually building a short HTML string. This is because the repeater can't handle *if* statements. Therefore, in order to hide or show users' e-mail addresses depending on whether the viewer is logged in or not, we need to build a string instead of directly inserting the value. If the user is logged in, then the anchor tag for the poster's e-mail address is built; otherwise, an empty string is used.

Creating the User Functions

Registered Users (Members) get a special set of functions they can access, such as creating threads and posts, editing their profile, and editing the messages they've posted. A Guest (that is, an unregistered user) is limited to a very small set of functionalities—specifically, viewing the threads and messages (Figure 8.58).

Figure 8.58 The Thread Page



Editing the Member Profile

The next step in building our application's user interface is to allow a registered user to modify his or her member profile. This includes first name, last name, password, and e-mail address. Let's take a look at the profile.aspx page in Figure 8.59.

The profile page contains text boxes for every field in the *User* object, except for the user ID and username. These two fields are read only, and should never be changed. Like the Register page, this page contains a number of validation controls with their display value set to none, and a *ValidationSummary* control added to the page to aggregate all the errors a user might receive while inputting information. When this page first loads, it should default all fields (except for passwords) with their existing values, so a user does not have to type everything over, just change the fields he or she wants to change. Upon clicking the **Update Profile** button, the user's details should be updated and the user given a message explaining that his or her profile was updated. Let's take a look at the implementation of these features in Figure 8.60. The complete source code for Figure 8.60 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.59 The Profile Page

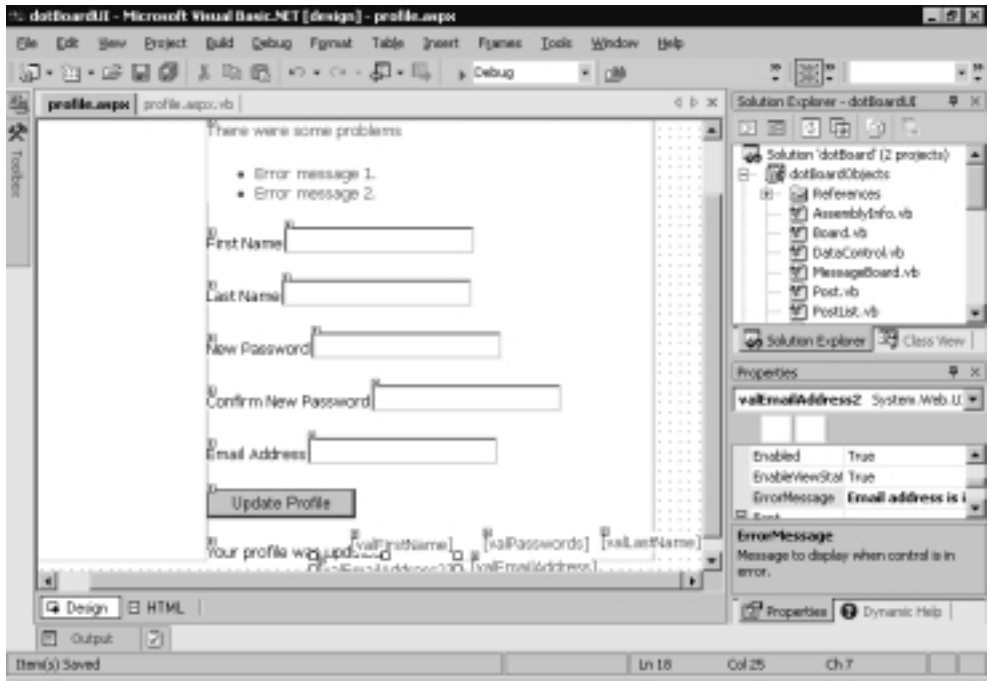


Figure 8.60 The Code-Behind (Profile.aspx.vb)

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    If Me.IsLoggedIn = False Then
        'only logged in users can access this site
        Response.Redirect("default.aspx")
    End If

    If Page.IsPostBack = False Then
        txtFirstName.Text = Me.CurrentUser.FirstName
        txtLastname.Text = Me.CurrentUser.LastName
        txtEmailAddress.Text = Me.CurrentUser.Email
    End If

    Me.ApplyStyles(Me.Controls)
End Sub
```

Continued

Figure 8.60 Continued

```
Private Sub btnUpdate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnUpdate.Click
    If Page.IsValid Then
        If txtNewPassword.Text.Trim() <> "" Then
            Me.CurrentUser.Password = txtNewPassword.Text
        End If
        Me.CurrentUser.FirstName = txtFirstName.Text
        Me.CurrentUser.LastName = txtLastname.Text
        Me.CurrentUser.Email = txtEmailAddress.Text
        Me.CurrentUser.Update()
        lblMessage.Visible = True
    End If
End Sub
```

Updating the user profile is rather easy. First, the *Page_Load* method checks to make sure there is a valid, logged-in user. If not, it redirects the user back to default.aspx. If the user is logged in and the page has not posted back to itself yet, it sets the values of the text boxes to the existing values of the current *User* object. Afterward, it applies the styles to the page and exits.

When the **Update button** is clicked, the *btnUpdate_Click* method is called. The subroutine first checks to make sure all the validation controls have returned valid results. If not, it exits the subroutine. If they have returned valid results, it first checks to see if the user entered a new password, and if so, sets the current *User* object's password to what the user entered. Next, each of the *User* objects' fields are set to what the user entered, then the *User* object is updated to the database. Finally, the message label indicating that the profile was updated successfully is displayed.

Creating Threads and Posts

The last thing to do for Registered Users is to generate a page for them to create new threads and posts. In order to get to this page, let's take a look at board.aspx and thread.aspx again. We need to add a *LinkButton* to each. When clicked, that link button needs to redirect the user to createpost.aspx. See Figures 8.61 and 8.62. The complete source code for Figure 8.61 and Figure 8.62 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.61 *LinkButton1_Click* Event (Board.aspx)

```
Private Sub LinkButton1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles LinkButton1.Click
    Dim boardId As String
    boardId = Request.QueryString.Item("boardid")
    Response.Redirect("createPost.aspx?boardName=" & boardId)
End Sub
```

Figure 8.62 *LinkButton1_Click* Event (Thread.aspx)

```
Private Sub LinkButton1_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles LinkButton1.Click
    Dim boardId As String
    Dim threadId As Long

    boardId = Request.QueryString.Item("boardId")
    threadId = CLng(Request.QueryString.Item("threadId"))

    Response.Redirect("createPost.aspx?boardName=" & boardId & _
        "&threadId=" & threadId.ToString())
End Sub
```

The function of these buttons is almost the same. The first one redirects the user to createpost.aspx?boardName=[*The selected Board*], and the second redirects the user to createpost.aspx?boardName=[*The selected Board*]&threadId=[*The selected Thread*]. The same page handles the creation of new Threads and Posts, so if you are creating a new Post, you just pass in the *ThreadID* along with the board name. If you are creating a brand new Thread, you just pass in the Board name. Let's take a look at createpost.aspx to see what controls are on that page in Figure 8.63.

The Create Post page contains the necessary controls to accept user input and create a new Thread and/or Post. The other controls on the page are a *ValidationSummary*, two *RequiredFieldValidators*, and a *Panel* that contains the current Thread information. Obviously, if the user is creating a new Thread and Post, the Thread panel will not be visible; whereas, if the user is creating a new Post

inside a Thread, the Thread panel will be visible and display the appropriate Thread subject. Let's take a look at the code necessary to initialize this form in Figure 8.64. The complete source code for Figure 8.64 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.63 The Create Post Page

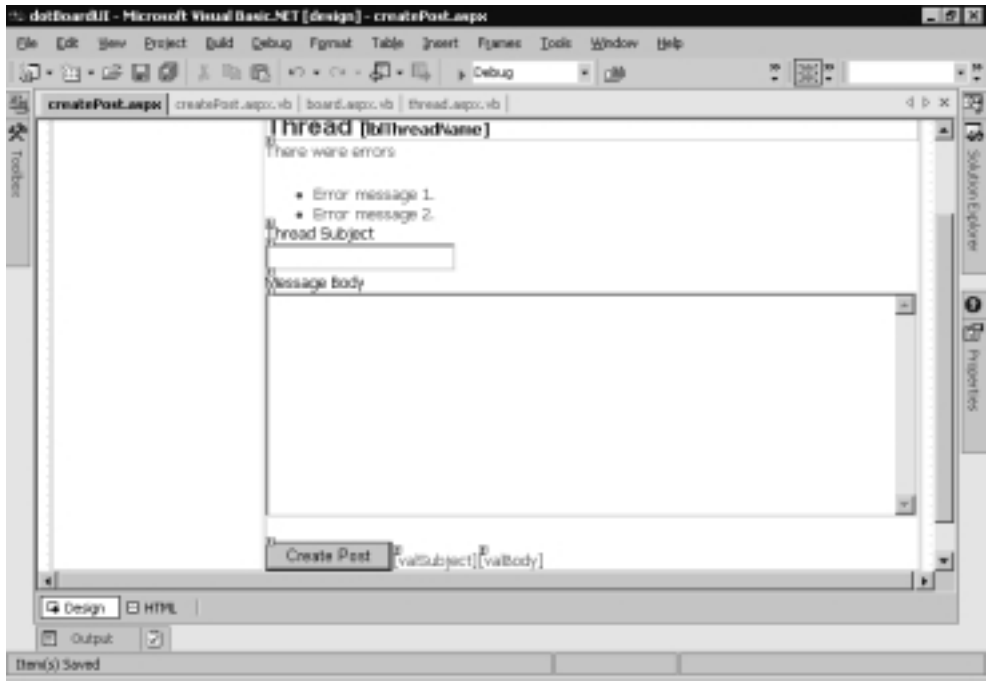


Figure 8.64 The Code-Behind Initialization (Createpost.aspx.vb)

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    'only logged in users are allowed in this page
    If Me.IsLoggedIn = False Then
        Response.Redirect("default.aspx")
    End If

    mBoardName = Request.Item("boardName")
    If Request.Item("threadId") Is Nothing Then
        mThreadID = 0
```

Continued

Figure 8.64 Continued

```
    Else
        mThreadID = CLng(Request.Item("threadId"))
    End If

    mBoard = New dotBoardObjects.board(mBoardName)
    lblBoardName.Text = mBoard.Name

    If mThreadID = 0 Then
        pnlShowThread.Visible = False
    Else
        pnlShowThread.Visible = True
        mThread = mBoard.ChildThread(mThreadID)
    End If

    If Not Me.IsPostBack Then
        'put the default values in the thread and board text boxes
        If mThreadID <> 0 Then
            txtThreadSubject.Text = mThread.Subject
            lblThreadName.Text = mThread.Subject
        End If
    End If

    Me.ApplyStyles(Me.Controls)
End Sub
```

First, what we do is verify that there is a logged-in user. If there isn't, we redirect the user back to the default page. If the user is valid, we get a reference to the current board and if the *ThreadID* was passed in, we get a reference to the appropriate Thread as well. Finally, if the page hasn't posted back to itself and we have a current Thread, we default the text box and label values with the Thread's subject. All that's left is to take a look at the code that actually creates Posts and Threads, as shown in Figure 8.65. The complete source code for Figure 8.65 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.65 *btnCreatePost_Click* Code (Createboard.aspx.vb)

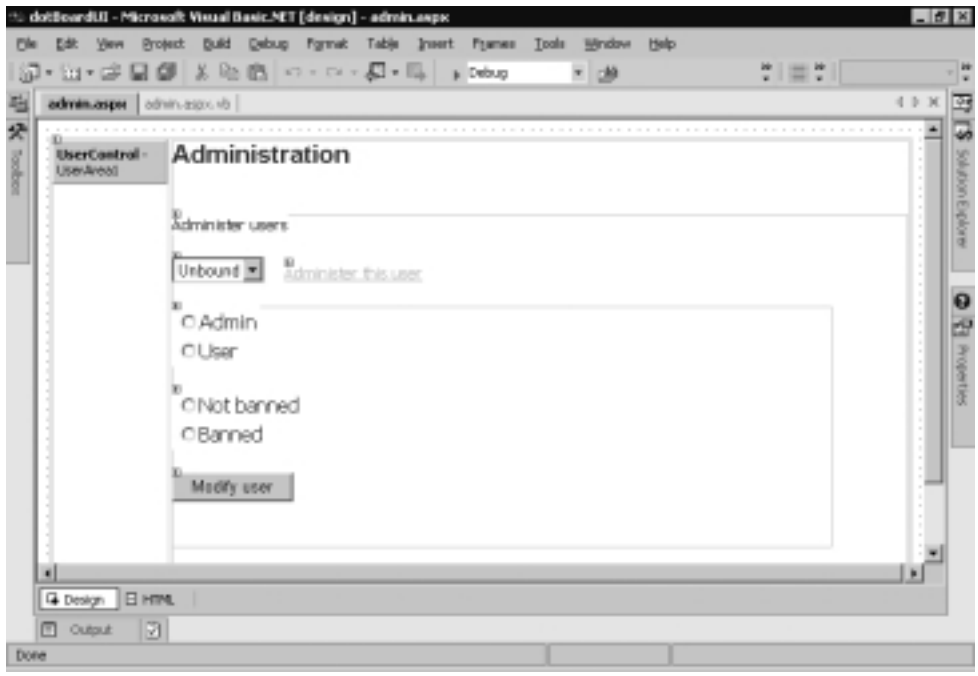
```
Private Sub btnCreatePost_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnCreatePost.Click
    If Me.IsValid = True Then
        If mThreadID <> 0 Then
            'we're adding a post to a thread. do nothing here
        Else
            'we're creating a new thread and adding a post
            mBoard.CreateThread(txtThreadSubject.Text, Me.CurrentUser)
            'let's find that thread. it will be the first one
            'in the list
            mThread = mBoard.ChildThreads.Item(0)
        End If

        mThread.CreatePost(txtThreadSubject.Text, _
            TextBox1.Text, Me.CurrentUser)
        'redirect the user to the current thread
        Response.Redirect("thread.aspx?boardId=" & mBoardName & _
            "&threadId=" & mThread.ID.ToString())
    End If
End Sub
```

What happens in this bit of code is that we first check to make sure the page is valid. If not, we do nothing; otherwise, we attempt to create the Thread and/or Post. If the *ThreadID* is currently "0" (that is, no *ThreadID* was given to the page), then we create a new Thread and set the private *mThread* variable to the new Thread (remember that when adding a new Thread, since Threads are ordered by their *ThreadID* field, new Threads appear at the top of the *ThreadList*). Lastly, we create a new Post from the current *Thread* object and redirect the user to the thread.aspx page to view the new and/or updated Thread.

Building the Administrative Interface

Administrators need to do a few things that other people can't. First, they need the ability to delete anything—Boards, Threads, and Posts. They also need the ability to edit any Post, and modify any user's admin or banned status. Let's take a look at the useradmin.aspx screen in Figure 8.66.

Figure 8.66 The User Admin Page



This page allows administrators to promote other users to Administrator status, and ban problematic users from logging in to the site. First, we have a *DropDownList* control that we will *DataBind* to a *DataSet*. There is also a *LinkButton* that will show the admin panel at the bottom once we've selected a user to administer. The two radio button lists will be used to display and set the current admin/banned status of the selected user. Finally, when the user clicks the **Modify User** button, the current user will be updated with the new banned and admin values the administrator entered. Let's first take a look at the code necessary to set up the form in Figure 8.67. The complete source code for Figure 8.67 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.67 The *Page_Load* Method (Admin.aspx.vb)

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    'only logged-in admins can enter this page
```

Continued

Figure 8.67 Continued

```
    If Me.IsLoggedIn = False Then
        Response.Redirect("default.aspx")
    ElseIf Me.CurrentUser.IsAdmin = False Then
        Response.Redirect("default.aspx")
    End If
    'get the users bound to the drop down list
    If Not Me.IsPostBack Then
        Dim myUsers As DataSet
        Dim sql As String
        sql = "SELECT UserID, UserName FROM Users"
        myUsers = dotBoardObjects.DataControl.GetDataSet(sql)
        dlUsers.DataTextField = "Username"
        dlUsers.DataValueField = "UserID"
        dlUsers.DataMember = "data"
        dlUsers.DataSource = myUsers
        dlUsers.DataBind()
    End If
    Me.ApplyStyles(Me.Controls)
End Sub
```

The first thing this method does is guarantee that there is a logged-in user, and that the currently logged-in user is an administrator. If either of these is not true, it sends the user back to default.aspx. Next, it makes sure the page has not posted back to itself; since there's no need to *DataBind* a drop-down list every time the page is executed, as ASP.NET will handle that for us. If the page has not posted back to itself, it builds a SQL statement to retrieve the UserIDs and Usernames from the Users table in the database. It then gets a *DataSet* from the *dotBoardObjects.DataControl* class, and dynamically binds the *DropDownList* to the *DataSet*. Finally, it applies the styles to this page and exits.

The next thing we need to do is have the ability to select a user from the drop-down list, and have the page load that user's information. The click event handler for the Choose User link handles this. Let's take a look at the code for it in Figure 8.68. The complete source code for Figure 8.68 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.68 The *lnkChooseUser_Click* Method (Admin.aspx.vb)

```
Private Sub lnkChooseUser_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lnkChooseUser.Click
    Dim userID As Long
    userID = CLng(dlUsers.SelectedItem.Value)
    Dim myUser As dotBoardObjects.User
    myUser = New dotBoardObjects.User(userID)

    If myUser.IsBanned = True Then
        rblBanned.Items(1).Selected = True
    Else
        rblBanned.Items(0).Selected = True
    End If

    If myUser.IsAdmin = True Then
        rblAdmin.Items(0).Selected = True
    Else
        rblAdmin.Items(1).Selected = True
    End If

    rblBanned.Visible = True
    rblAdmin.Visible = True

    Panel1.Visible = True
End Sub
```

This gets the user ID from the DropDownList's *SelectedItem.Value* property, and creates a new *User* object from it. Next, the appropriate radio buttons are selected depending on whether or not the user is banned or is an admin. Finally, the admin panel and the two radio button lists are set to visible, so they will appear when the page refreshes. Next, we need to handle when the administrator clicks the **Modify User** button and update the selected user based on what the administrator entered. See Figure 8.69 for the code involved. The complete source code for Figure 8.69 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.69 The *btnModify_Click* Method (Admin.aspx.vb)

```
Private Sub btnModify_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnModify.Click
    Dim userID As Long
    userID = CLng(dlUsers.SelectedItem.Value)
    Dim myUser As dotBoardObjects.User
    myUser = New dotBoardObjects.User(userID)

    'we now have the user, so let's set his admin/banned properties
    If rblBanned.Items(0).Selected = True Then
        'the user is not banned
        myUser.IsBanned = False
    Else
        myUser.IsBanned = True
    End If

    If rblAdmin.Items(0).Selected = True Then
        'the user is an admin
        myUser.IsAdmin = True
    Else
        myUser.IsAdmin = False
    End If

    myUser.Update()
End Sub
```

Just as before, the first thing we do is get a reference to the selected *User* object. The next step is to determine which radio buttons were selected, and set the *IsAdmin* and *IsBanned* properties accordingly. The last step is to update the selected user by calling its *Update* method. Now you can promote other users to be administrators, or ban them from entering your site again. If a banned user attempts to log on, he will receive an error explaining that his account was banned. You might be wondering why we don't just delete the banned user. We don't do this because the Thread and Post tables are dependent on the User table, and deleting a user from the User table would not be allowed due to the relationships involved.

The other thing that administrators can do is create and delete Boards, delete Threads, and delete Posts. Let's start with creating a Board. The first step involved in this is adding a new *LinkButton* to the user area user control. This button will be named "lnkCreateBoard" and will have its text property set to Create New Board. Once clicked, it should redirect the user to createboard.aspx. Let's take a look at that code in Figure 8.70. The complete source code for Figure 8.70 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

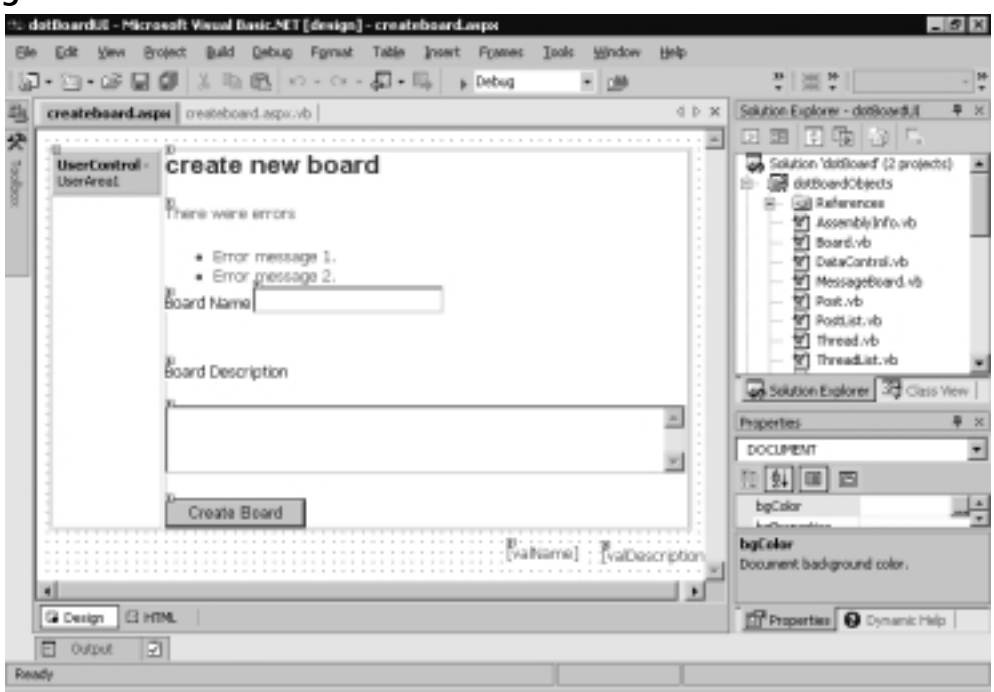
SYNGRESS
syngress.com

Figure 8.70 The *lnkCreateBoard_Click* Code (Userarea.ascx.vb)

```
Private Sub lnkCreateBoard_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles lnkCreateBoard.Click
    Response.Redirect("createboard.aspx")
End Sub
```

Now that we have the administrator going to the Create Board page, let's take a look at that page (Figure 8.71).

Figure 8.71 The Create Board Form



Like all our other pages that accept user input, this page has controls on it for every piece of information we need to perform the task at hand. Also, like the other pages, there is a validation control for every text box to make sure the user enters the required information. Let's take a look at the code-behind for this form in Figure 8.72. The complete source code for Figure 8.72 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.72 The Code-Behind (Createboard.aspx.vb)

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    'only logged-in admins can enter this page
    If Me.IsLoggedIn = False Then
        Response.Redirect("default.aspx")
    ElseIf Me.CurrentUser.IsAdmin = False Then
        Response.Redirect("default.aspx")
    End If
End Sub

Private Sub btnCreate_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnCreate.Click
    If Me.IsValid = True Then
        'create the new board
        dotBoardObjects.Board.CreateBoard(txtBoardName.Text, _
            txtBoardDescription.Text, _
            Me.CurrentUser)
        Response.Redirect("default.aspx")
    End If
End Sub
```

Like every other admin page so far, this page guarantees that the current user is a logged-in administrator, and if not, redirects to the default page. After the user has entered the required information to create a board and clicks the **Create Board** button, the *btnCreate_Click* method is called. First, the method checks to make sure the page is valid, then it creates the Board based on the values the administrator entered. Finally, it redirects the administrator back to the default page so he can see his newly created board.

The last things an administrator should be able to do are delete Boards, Threads, and Posts. This functionality can be placed on the appropriate pages where this information is actually displayed. What we will do is, next to every Board, Thread, and Post we will place an *HtmlAnchor* control next to each item that will point to an .aspx page named delete[*type of object to delete*].aspx. For example, deleting Boards will link to deleteBoard.aspx. Let's go over the three places in our code that need to change because of this new feature in Figures 8.73, 8.74, and 8.75. The complete source code for the next three figures can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.73 The *DisplayBoard* Method Changes (Default.aspx.vb)

```
Dim fields(1) As Object
fields(0) = myBoard.Name
fields(1) = myBoard.Description
If Me.IsLoggedIn = True Then
    If Me.CurrentUser.IsAdmin = True Then
        fields(1) &= "<br><br><a href='deleteBoard.aspx?boardName=" & _
            myBoard.Name & "'>&gt;&gt;delete</a>"
    End If
End If
```

Figure 8.74 The *Page_Load* Method Changes (Board.aspx.vb)

```
Dim fields(3) As Object
fields(0) = "BoardId=" & boardId & _
    "&ThreadId=" & myThread.ID.ToString()
fields(1) = myThread.Subject
If Me.IsLoggedIn = True Then
    If Me.CurrentUser.IsAdmin = True Then
        fields(1) &= "<br><br><a href='deleteThread.aspx?" & _
            "boardName=" & mBoard.Name & _
            "&threadId=" & myThread.ID.ToString() & _
            "'>&gt;&gt;delete</a>"
    End If
End If
```

Figure 8.75 The *Page_Load* Method Changes (Thread.aspx.vb)

```
Dim fields(5) As Object
fields(0) = myPost.ID
fields(1) = myPost.Subject
fields(2) = myPost.Body
If Me.IsLoggedIn = True Then
    If Me.IsLoggedIn = True Then
        fields(2) &= "<br><br><a href='deletePost.aspx?" & _
            "boardName=" & myBoard.Name & _
            "&threadId=" & myThread.ID.ToString() & _
            "&postId=" & myPost.ID.ToString() & _
            "'>&gt;&gt;delete</a>"
    End If
End If
```

You can see that all of these changes are very similar. Each gets slightly more complicated as you get further down the object hierarchy; you need to pass more information to get a reference to the correct objects. Now all we need to do is create the three pages that will handle deleting our objects. All three are very similar, and are shown in Figures 8.76, 8.77, and 8.78. The complete source code for the following three figures can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.76 DeleteBoard.aspx.vb

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    If Me.IsLoggedIn = True Then
        If Me.CurrentUser.IsAdmin = True Then
            Dim boardName As String
            boardName = Request.QueryString.Item("boardName")
             Dim myBoard As dotBoardObjects.board
            myBoard = New dotBoardObjects.board(boardName)
            myBoard.Delete(Me.CurrentUser)
        End If
    End If

    Response.Redirect("default.aspx")
End Sub
```

SYNGRESS syngress.com

Figure 8.77 DeleteThread.aspx.vb

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    If Me.IsLoggedIn = True Then
        If Me.CurrentUser.IsAdmin = True Then
            Dim boardName As String
            Dim threadId As Long
            boardName = Request.QueryString.Item("boardName")
            threadId = CLng(Request.QueryString.Item("threadId"))

            Dim myBoard As dotBoardObjects.board
            myBoard = New dotBoardObjects.board(boardName)
            Dim myThread As dotBoardObjects.thread
            myThread = myBoard.ChildThread(threadId)

            myBoard.DeleteThread(myThread, Me.CurrentUser)
        End If
    End If

    Response.Redirect("default.aspx")
End Sub
```

SYNGRESS syngress.com

Figure 8.78 DeletePost.aspx.vb

```
Private Sub Page_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    If Me.IsLoggedIn = True Then
        If Me.CurrentUser.IsAdmin = True Then
            Dim boardName As String
            Dim threadId As Long
            Dim postId As Long

            boardName = Request.QueryString.Item("boardName")
            threadId = CLng(Request.QueryString.Item("threadId"))
            postId = CLng(Request.QueryString.Item("postId"))
```

Continued

Figure 8.78 Continued

```
            Dim myBoard As dotBoardObjects.board
            myBoard = New dotBoardObjects.board(boardName)
            Dim myThread As dotBoardObjects.thread
            myThread = myBoard.ChildThread(threadId)
            Dim myPost As dotBoardObjects.Post
            myPost = myThread.ChildPost(postId)

            myBoard.DeletePost(myThread, myPost, Me.CurrentUser)
        End If
    End If

    Response.Redirect("default.aspx")
End Sub
```

A lot of code, for sure, but it should all be relatively easy to follow. Each page retrieves the objects necessary to delete whatever it is trying to delete, and then calls the appropriate delete method on the *Board* object. When it finishes, each redirects the user back to the default page. If the person accessing this page is neither logged in nor an admin, it does nothing but the final redirect. You don't want anyone who is not an admin deleting your Boards, so even on pages in which the user never sees the UI, it's still a good idea to perform every security check necessary.

The final administrative interface we need to create is to give the administrators the ability to edit Posts, in the case of offensive or undesired language that doesn't necessarily need to be deleted. First, we'll need to add another button to the view thread page right next to the **Delete** button. See Figure 8.79 for the changes. The complete source code for Figure 8.79 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 8.79 Page_Load Changes (Thread.aspx.vb)

```
If Me.IsLoggedIn = True Then
    If Me.IsLoggedIn = True Then
        fields(2) &= "<br><br><a href='deletePost.aspx?" & _
            "boardName=" & myBoard.Name & _
```

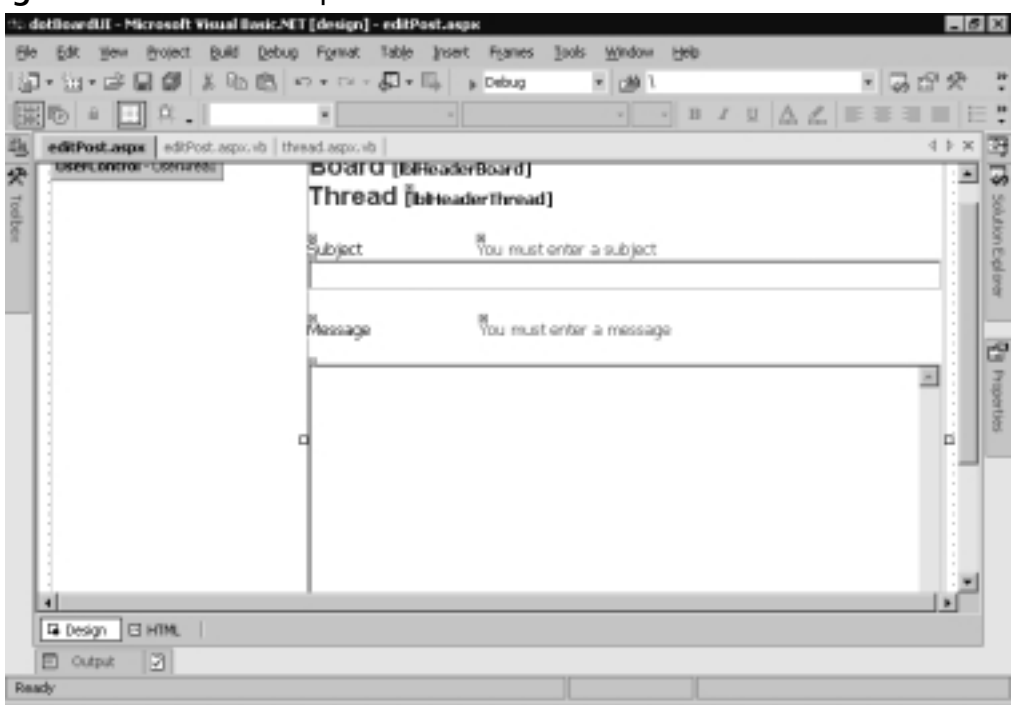
Continued

Figure 8.79 Continued

```
                "&threadId=" & myThread.ID.ToString() & _
                "&postId=" & myPost.ID.ToString() & _
                "'>&gt;&gt;delete</a>"
            fields(2) &= "   " & _
                "<a href='editPost.aspx?" & _
                "boardName=" & myBoard.Name & _
                "&threadId=" & myThread.ID.ToString() & _
                "&postId=" & myPost.ID.ToString() & _
                "'>&gt;&gt;edit</a>"
        End If
    End If
```

All that has changed is that a new HTML anchor tag is added that points to a new page called editPost.aspx. Let's take a look at this page and examine what controls are on it (Figure 8.80).

Figure 8.80 editPost.aspx



You should notice that this page looks very similar to the Create Post page. In fact, it is nearly identical—so identical that we could have reused the same page instead of creating the new one. The only reason we aren't using the create post page is for the sake of simplicity; there's no need to complicate pages we have already finished for new functionality. All we need to do now is take a look at the code-behind page in Figure 8.81. The complete source code for Figure 8.81 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 8.81 The Code-Behind (editPost.aspx)

```
Public Class editPost
    Inherits FormBase

    Private mBoard As dotBoardObjects.Board
    Private mThread As dotBoardObjects.Thread
    Private mBoardName As String
    Private mThreadID As Long
    Private mPostID As Long
    Private mPost As dotBoardObjects.Post

    Private Sub Page_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load
        'only logged in users are allow in this page
        If Me.IsLoggedIn = False Then
            Response.Redirect("default.aspx")
        ElseIf Me.CurrentUser.IsAdmin = False Then
            Response.Redirect("default.aspx")
        End If

        mBoardName = Request.Item("boardName")
        mThreadID = CLng(Request.Item("threadId"))
        mPostID = CLng(Request.Item("postId"))

        mBoard = New dotBoardObjects.board(mBoardName)
        mThread = mBoard.ChildThread(mThreadID)
        mPost = mThread.ChildPost(mPostID)
```

Continued

Figure 8.81 Continued

```
        lblHeaderBoard.Text = mBoard.Name
        lblHeaderThread.Text = mThread.Subject

        If Not Me.IsPostBack Then
            txtSubject.Text = mPost.Subject
            txtMessage.Text = mPost.Body
        End If

        Me.ApplyStyles(Me.Controls)
    End Sub

    Private Sub btnEditPost_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles btnEditPost.Click
        If Me.IsValid Then
            mPost.Subject = txtSubject.Text
            mPost.Body = txtMessage.Text
            mPost.Update(Me.CurrentUser)
            Response.Redirect("thread.aspx?boardID=" & _
                mBoard.Name & "&threadId=" & _
                mThread.ID.ToString())
        End If
    End Sub
End Class
```

You should immediately notice how similar the code-behind of the Edit Post page is to the Create Post page. Again, we could have used the same page, but to keep things simple we're using two separate pages. The *Page_Load* method first checks to make sure there is a logged-in user, and that the user is an administrator. Next, it gets a reference to the appropriate *Board*, *Thread*, and *Post* objects, and fills the label and text box controls on the page with values. The *btnEditPost_Click* method makes sure the page is valid, then sets the values on the *Post* object, commits it to the database, and redirects to the ThreadView page so the user can see the changes.

Summary

Our message board is 100-percent complete and ready for use. We have analyzed our message board and created a solution to fit with all our requirements. Our message board is an object-oriented application that is scalable, maintainable, and well defined. We have created all the necessary classes to maintain our data and the relationships between our data through the use of custom list objects and classes. We also have a built-in security model where every action that requires administrative access is checked before the requestor is allowed to perform the operation.

Our user interface is somewhat extensible in that it dynamically applies styles to multiple types of *WebControls* that we defined using CSS and an XML document. Each Web Form we created inherits the *FormBase* class, which allows all our Web Forms to have access to a few common methods and properties, in addition to the *System.Web.UI.Page* methods and properties. Our user interface contains all the necessary interfaces to browse through Boards, Threads, and Messages, as well as interfaces to administer users, and those that contain interfaces to create and delete Boards, Threads, and Messages.

All in all, we have a functioning message board that could be placed anywhere and run on top of SQL Server or MS Access. It was accomplished in an object-oriented manner and hopefully, by now, you understand the use for designing OO applications. We have also separated the UI and UI logic from the actual "business rules" applied to our objects. If we wanted, we could take our *dotBoardObject* class library and put a Windows Form front end on it, a Web Service front end on it, or even attach a Console Application front end—all because we kept our UI completely separate from our implementation.

Solutions Fast Track

Setting Up the Database

- ☑ Analyze your data and create the tables necessary to represent the solution to our problem. Make sure you have broken down each piece of data into the smallest possible representation of that data. For example, you wouldn't want to have a field in your database for the user's full name; instead, you would want first and last name fields.
- ☑ Analyze your data and create the relationships necessary between the different sets of data.

Designing Your Application

- ☑ Analyze your data and find a way to fit it into an object-oriented environment. Many times, you can use the analysis you performed while building your database in this step.
- ☑ Map the fields in the database to appropriate fields in each object.
- ☑ Analyze our solution and determine the types of methods each of our objects will contain. You need to provide interfaces to modify, add, and delete every relationship and field in each of your objects.

Designing the User Interface

- ☑ Analyze what type of actions our users will need to perform, and create the necessary Web Forms.
- ☑ Analyze what type of actions our administrators will need to perform, and create the necessary Web Forms.

Setting Up General Functions

- ☑ Create the *FormBase* class that contains all the necessary properties and methods our Web Forms will need to hold. Determine what functionality you need shared throughout every Web Form, and build it into this class.

Building the Log-In Interface

- ☑ Create the user area user control. Place this control on every Web Form so each form can have a reference to the currently logged-in user.
- ☑ Create the Registration page, which allows users to register for your message board.

Designing the Browsing Interface

- ☑ Create the Board browsing. Create the Web Form and use the *Repeater* control and *DataBind* it to a *DataSet*.
- ☑ Create the Thread browsing. Create the Web Form and use the *Repeater* control and *DataBind* it to a *DataSet*.
- ☑ Create the Post browsing. Create the Web Form and use the *Repeater* control and *DataBind* it to a *DataSet*.

Creating the User Functions

- ☑ Generate the Thread creation. Create the Web Form and use *Validation* controls and text boxes to get the necessary information.
- ☑ Generate the Post creation. Create the Web Form and use *Validation* controls and text boxes to get the necessary information.

Building the Administrative Interface

- ☑ Create the interface to ban and promote users. Make sure only administrators can access this functionality using the properties built into the *FormBase* class.
- ☑ Create the interfaces necessary to delete Board, Thread, and Post pages. Modify the existing View Board, Thread, and Post pages to create the links to the delete pages.
- ☑ Create the interfaces necessary to edit Posts. Modify the existing view Post page to create the links to edit Posts.

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** and click on the **"Ask the Author"** form.

Q: When designing applications, do I need to design them in an object-oriented manner?

A: Absolutely not, although when applications are designed in an OO manner, they are typically more scalable and maintainable, and allow for the use of multiple user interfaces. You are not forced to create applications in an OO manner, but good programming practices typically stress object orientation.

Q: Are there any performance issues when using an OO approach versus a more procedural approach?

A: Yes, typically the OO approach adds a bit of overhead to everything you do. For example, the creation of the custom *DataSet* in order to view Boards,

Threads, and Posts spends extra time that wouldn't have been lost if you had gone directly to the database instead of accessing the data through objects. The price of scalability and maintainability is a possible performance loss. Luckily, with .NET, execution is very fast after the initial compile, so it's also very likely that you would never notice the speed loss.

Q: How important is it to use Validation controls?

A: Very important. In ASP 3.0 and 2.0 (heck, even ASP 1.0), all validation had to be done by hand. Empty fields needed to be validated as well as e-mail addresses and URIs. With Validation controls, ASP.NET does all of this for us, allowing us to focus more on the logic and business rules in our application.

Q: How can I ban a list of IP addresses in the future?

A: First, you would need to create a table in your database to store the list of IP addresses, and provide a way for an administrator to enter an IP address into it. Then, at every page you want to disallow this list of IP addresses from viewing, compare the IP address of the requesting user and compare it to the list of IP addresses you have banned. If it exists in your list of banned addresses, redirect the user to another page or do whatever else you feel is appropriate.

Chapter 9

Building a Remote Database Viewer

Solutions in this chapter:

- Understanding ADO.NET
- Accessing Data from a Database Using ADO.NET
- Converting Binary Data Using Base64
- Designing and Implementing a Remote Database Viewer

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

By now you might have realized the versatility of XML as a markup language, but you still might not be in a position to appreciate its usefulness beyond the implementations that we have carried out so far. If you review the previous chapters, you will find that we restricted our focus to working with plain text wrapped into XML encoding, with the exception of only XML serialization. We have yet to consider the conversion of data other than plain text into XML encoding. This would be necessary if, for instance, it was needed to extract all the data from a SQL Server database to be sent to another application in the form of XML documents, or if a complete database structure were to be synchronized over the Internet by wrapping it in XML documents. Serialization would work, but only partially in these cases, as images, other OLE objects, and even stored procedures need to be handled. Fortunately, some techniques have evolved for such situations too; else, the utility of XML would have been severely limited.

Perhaps you are already familiar with the technique of encoding binary data into a character string; for example, using Base64 encoding. This technique can be used within the .NET Framework. In fact, it offers some respite from the tedium involved in extracting an image from a database, encoding it, and wrapping it into an XML document. The use of namespaces is part of the process, but the introduction of ADO.NET is more significant. Microsoft has integrated XML in the ADO.NET architecture, leveraging its ease of use. Thus, one can read recordsets from SQL Server databases and then save them as XML documents, using only a handful of code.

In this chapter, we address both of these aspects of data encoding in XML documents by creating a tool that allows one to remotely browse extracted data from a SQL Server database and store the data as XML documents. First, we will take a close look at the constitution of ADO.NET and how binary data encoding works.

Understanding ADO.NET

ADO.NET is based on the principle of universal data access (UDA) for accessing data from a database. Although ADO.NET is specifically designed for building .NET applications, it completely rules out database dependency as in the case of ADO (ActiveX Data Objects). ADO.NET is backed up by all such classes that are capable of database handling features, such as sorting the data using various options for viewing the data in different formats, and reading data from and putting it back on the database. ADO.NET uses XML as its default format for data traversing, while with ADO, COM Marshalling is predominantly used for that

purpose. Yet, once you start working with ADO.NET you will find that familiarity with the concepts and techniques of ADO go a long way in building effective solutions for data access ADO.NET. The ADO.NET also sports these features:

- **Interoperability** In the ADO.NET architecture, data travels across the applications in an XML stream format. This confluence of XML and ADO.NET offers the parent application a free environment for transmitting data to the receiving application, without bothering about how data will be read by the receiving application. The only requirement demanded of the data-consuming application is the capability to read the XML.
- *DataSet* The *DataSet*, a new feature within ADO.NET, is the memory resident copy of data. It can represent the data of more than one table, without using the *JOIN* clause in an SQL query. Thus, we can assume *DataSet* as some sort of mini database that holds the data of multiple tables and represents the relationship between them. Whatever modification is required can be done on the copy of the data, and later, this modified copy of the data is submitted to the database for resolving updates. The copy of the data maintained by the *DataSet* can be distributed among various components, thereby the need of individually querying the data source. In the ADO.NET environment, you can call *DataSet* as a client-side cursor location.
- **Performance** The performance of ADO.NET is further upgraded by the extended support of XML. As mentioned earlier, XML is the (default) data format in ADO.NET, which allows for manipulating and recognizing the data irrespective of the data source, as the scope of XML is not restricted to certain data types. Once the data is gained by the recipient application, it can convert the data using its own data types.
- **Scalability** Any Web-based application that uses ADO.NET can handle any number of users without bothering about bottlenecks such as server overloading due to too many connections, slow processing of queries, and so forth, because ADO.NET applications do not maintain constant connection with the database of the server. Neither does an ADO.NET application, at the time of running, deploy any database lock too long; instead, it works on the principle of *disconnected access database*.
- **Maintainability** As the performance load on a deployed application server increases, system resources can become scarce, thereby adversely affecting response time. To solve this problem, software architects can segregate the server's business logic processing and user interface processing

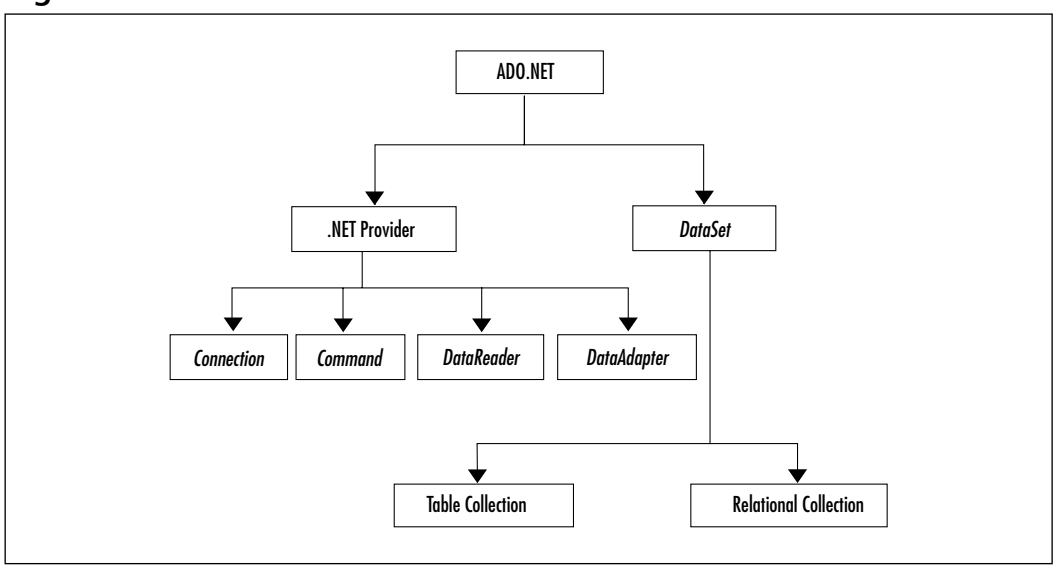
into separate tiers on separate machines. This, in effect, would mean that the application server tier is replaced with two tiers, which would alternate the load on system resources.

The problem is not designing a three-tier application; rather, it is increasing the number of tiers after an application has been deployed. If the original application is implemented in ADO.NET using datasets, this transformation is quite easy. Remember, when you replace a single tier with two tiers, you arrange for those two tiers to trade information. Since, the tiers can transmit data through XML-formatted datasets, the communication between the tiers is simplified.

The ADO.NET Architecture

The working of ADO.NET is based on two components: *DataSet* and .NET Data Provider. Figure 9.1 represents the inner architecture of ADO.NET.

Figure 9.1 The ADO.NET Architecture



Using .NET Data Provider

This core constituent of ADO.NET is specifically designed for data access and manipulation. A .NET Data Provider lays down the connection with the database, executes the commands or processes the queries, and retrieves the results. The .NET Provider comprises four objects, which are illustrated in Figure 9.2 and described in Table 9.1.

Figure 9.2 Core Objects of .NET Provider

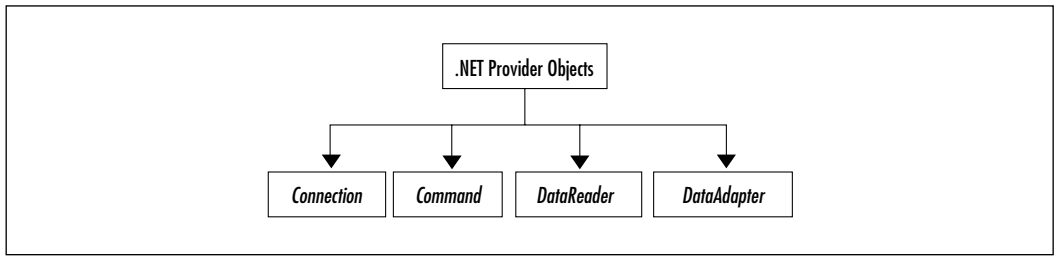


Table 9.1 Objects and Their Descriptions

Objects	Description
Connection	Establishes connection with a specified data source.
Command	Executes command on data source.
DataReader	Reads data from data source in read-only and forward-only streams.
DataAdapter	Fills the *DataSet* and handles update data source.

Connection

A *connection* with a database can be established either by following the conventional method of using the *Open* function, or by implicitly calling the *DataAdapter*. While establishing a connection with the database, the information required to authenticate the user and other information such as the name of the database is passed.

Command

Command contains the information about the SQL query that is submitted to the database. The submitted SQL query could be a query to retrieve the data, perform an Update or Insert query, or it could be a stored procedure.

DataReader

DataReader provides the functionality of reading data from the data source. You can consider it a server-side Cursor-Location. The *DataReader* object also contains the methods and properties necessary to deliver a forward-only data stream of rows from a data source. This means that you cannot update, add, or otherwise modify records streamed from *DataReader*.

DataAdapter

DataAdapter provides a set of methods and properties that are needed for retrieving and saving data between a *DataSet* and its source data store. It performs the work of transferring returned data from a database into a *DataSet*, and manages the reconciliation of data updated with regard to a database.

Data adapters are passed connections and commands whenever their action methods are invoked. One remarkable feature of *DataAdapter* is that it allows these commands to be set explicitly for controlling the statements used at runtime for resolving changes, and permits the use of stored procedures.

DataAdapter is the object that connects to the database to fill in the data. Moreover, depending upon the operations that take place, while the *DataSet* holds the data, *DataSet* connects to the database to update the data.

Thus, *DataAdapter* works in close alliance with *DataSet* and acts as the source of data. *DataAdapter* provides methods and properties to handle the data returned by the database, and passes it to the *DataSet*. The *DataAdapter* uses *Command* objects to execute SQL commands at the data source, and fills the *DataSet* with data. It also handles the task of updating the database.

The .NET data providers can be broadly categorized as follows:

- **The SQL Server .NET data provider** To use this data provider, you must have access to SQL Server. This data provider uses specifically defined protocols to communicate with SQL Server. It performs effectively, as it accesses SQL Server without involving the OLEDB or ODBC layers. In earlier versions of SQL Server, the OLEDB .NET provider was used with SQL Server OLE DB Provider (SQLOLEDB).
- **The OLEDB .NET provider** This data provider uses native OLEDB in conjunction with COM for accessing the data. The OLE DB handles both manual and automatic transactions. To use OLEDB .NET provider, you must include an OLE DB provider as well. Table 9.2 lists various OLE DB providers that are compatible with ADO.NET.

Table 9.2 OLEDB Providers Compatible with ADO.NET

Driver	Provider
SQLOLEDB	Microsoft OLE DB Provider for SQL Server
MSDAORA	Microsoft OLE DB Provider for Oracle
Microsoft.Jet.OLEDB.4.0	OLE DB Provider for Microsoft Jet

Using *DataSets* and *DataTables*

In previous versions of ADO, the *RecordSet* class had the responsibility of retrieving records from the database and performing all operations required for data manipulation. In ADO.NET, the *DataSet* class replaces the *RecordSet* class. Although both *DataSet* and *RecordSet* serve the same purpose, the flexibility and more diversified approach of *DataSet* gives ADO.NET a definite edge over ADO. In ADO, where *RecordSet* represents only one table, *DataSet* is a collection of two or more tables, and represents the relationship between the tables. In technical terms, you can consider *DataSet* as a relational database, where data is kept inside the memory of *DataSet*. Thus, data is kept strictly away from the original data source, thereby preventing any direct intervention.

DataSet maintains a copy of the data of related tables, and all the required operations are performed on this copy. Subsequently, *DataSet* submits this local copy of data to the original data source and resolves updates with the database. In this way, *DataSet* not only maintains the safety and integrity of data, but also promotes data operations with disconnected data sources in the true sense; unlike with ADO, where minimum cursor locations have to be maintained in spite of disconnected data sources. The algorithm of data and disconnected data sources is further consolidated by the merging of ADO.NET with XML. The data travels across various applications in XML format. The most remarkable feature of XML technology is that once data gets into the hands of *DataSet*, the original data source is of little significance. As evident in Figure 9.3, the working of *DataSet* is centered on *Table* and *Relational Collection*.

Figure 9.3 Components of *DataSet*

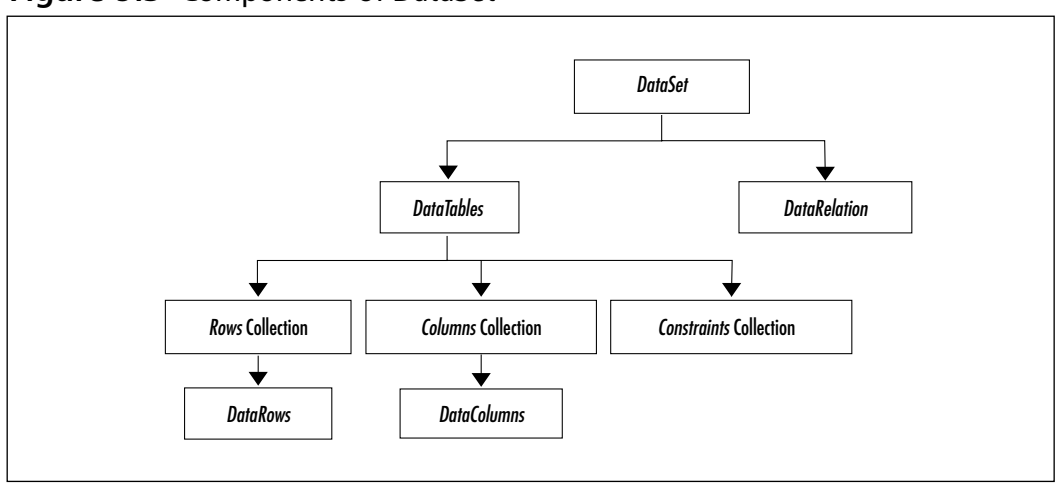


Table Collection represents all the tables *DataSet* is holding. Each table in *Table* Collection is represented by *DataTables*. Just like *DataTables* represents the tables within a database, *DataRows* represents the rows in the table and *DataColumns* represents the columns in a table. Figure 9.3 will help us better understand the components of *DataSet*.

A Quick Comparison of ADO and ADO.NET

In ADO.NET, a single *DataSet* handles data from multiple tables; performing the same task in ADO requires a JOIN in a SQL statement. Another significant difference between ADO and ADO.NET is in cursor types. The static and the read-only cursors of ADO are replaced by the *DataSet* and the *Data Reader*, respectively.

With ADO.NET, data can be shared with other applications using *DataSet* much more easily than with ADO, which uses *RecordSet*. In ADO.NET, XML is used for transmitting data across different applications, whereas in ADO, COM is used, which means that data types supported in ADO have to be COM centric. ADO.NET places no such restriction on data types, so you can send more richly formatted data than with ADO.

In ADO, constant connection with the data source database locks needs to be maintained for long periods of time, which sometimes leads to bottlenecks when too many connections supported by the data source are challenged. This problem has been bypassed in ADO.NET, where there is no need to maintain constant connection with the data source. Once you have the data, your connection becomes offline, and only when you need to submit the data back to the data source does the connection becomes active.

Accessing Data from a Database Using ADO.NET

Now that we have discussed the general scheme of ADO.NET and its components, let's familiarize ourselves with the process of working with data in Client-Server architecture. Working with data implies adding new data, deleting and updating the record, or merely viewing the records by navigating them.

In this section, various examples are given to illustrate the use of ADO.NET for accessing data from the database. The following section discusses the database design, which is accessed by various applications depicted as forms in the examples that follow. Each form has its own working schedule, and to access the data

from the database, each form first establishes the connection and then proceeds with the designated task.

Database Design

Our database will contain five simple rows: *emp_code* (which is our primary key), *emp_firstname*, *emp_lastname*, *designation*, and *salary* (Figure 9.4). Please note that in this example, we are assigning null values to *emp_firstname*, *emp_lastname*, *designation*, and *salary*. This is only for simplicity; in a real-world situation, all of these values cannot be null.

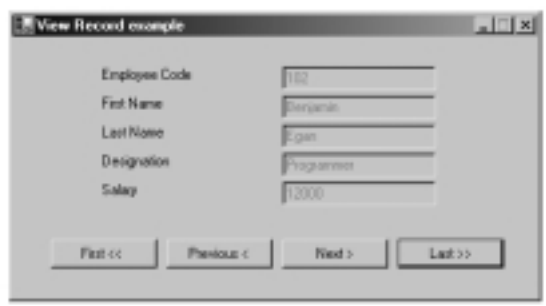
Figure 9.4 Database Design of emp Table

Column Name	Data Type	Length	Allow Nulls
emp_code	numeric	9	
emp_firstname	char	20	✓
emp_lastname	char	20	✓
designation	char	20	✓
salary	numeric	9	✓

Navigating between Records

The form in Figure 9.5 allows you to view various records stored in the table. For navigating between records, various buttons are placed on the form. To view records, you first need to establish connection with the database.

Figure 9.5 Navigation Form



The code block in Figure 9.6 serves to set up the connection with the database. The complete source code for Figure 9.6 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 9.6 Setting Up a Connection with the Database During Form Load

```
private OleDbConnection Conn;
private OleDbDataAdapter dbCmd;
   private DataSet dtSet;
   private DataRowCollection dtRowColl;
   private int CurRec;
   private void frmViewRecord_Load(object sender, System.EventArgs
   e)
   {
      string strSql="";

      Conn = new OleDbConnection("Provider=SQLOLEDB.1;Persist
      SecurityInfo=False;UserID=charul;pwd=charul;Initial
         Catalog=employee;DataSource=developers");

      try
      {
         Conn.Open();
      }
      catch(System.Data.OleDb.OleDbException ex)
      {
         MessageBox.Show(ex.ToString() + "Connection Fail.");
         this.Close();
      }

      dtSet=new DataSet();
      dtSet.Tables.Add("emp");
      strSql="Select * from emp";
      dbCmd= new OleDbDataAdapter(strSql,Conn);

      dbCmd.Fill(dtSet,"emp");
      dtRowColl=dtSet.Tables[0].Rows;
```

Continued

Figure 9.6 Continued

```
        if (dtRowColl.Count==0)
        CurRec=-1;
        else
        {
            CurRec=0;
            DisplayRec();
        }
    }
```

Looking at Figure 9.6, let's clarify how the connection with the database is established. While initiating connection with the database, information such as username, password, data source name, and so forth are passed and an attempt is made to open the connection to the database. If the connection cannot be established, an exception is raised and the user message is displayed informing you of the failure of the attempt. To execute the SQL statement on active connection, the object *dbCmd* of *OleDbDataAdapter* is made.

Once the connection with the database is established, the *dtSet*, an object of *DataSet*, is filled with table emp, which means that whatever data to be handled will be coming from emp table. To retrieve the data, a SQL query is written that will select all the records from the emp table. The *dbCmd* handles the data returned by the database and assigns it to *DataSet* object *dtSet*. From the data returned, *dtSet* picks up the first table and assigns the collection of rows present in the table to *dtRowColl*, an object of the *DataRowCollection* class (the object of *DataRowCollection*, which represents the collection of rows in the table).

The *Count()* method is used to determine whether some rows actually exist in *dtRowColl*. If this method encounters 0, which means that no row exists, nothing is displayed and the variable *CurRec* is set to –1. If otherwise, the *DisplayRec()* function is called, which will display the current record.

It is important to mention here that records in the table start at 0, which means that the first record is considered the second record. The variable *CurRec* represents the ordinal positions of records present in the table, which means that for *CurRec*, the zero record would be the first record. On the other hand, *Count()* function considers that records start from 1. Hence, we can say that when the variable *CurRec* and the *Count()* method encounter the records of the table, a difference of 1 occurs with regard to the position of records. We look at how to

display the current record, the first record, the last record, the previous record, and the next record in Figures 9.7 through 9.11, respectively. The complete source code for Figures 9.7 through 9.11 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS syngress.com

Figure 9.7 Function to Display the Current Record

```
private void DisplayRec()
{
   DataRow dr= dtRowColl[CurRec];
   txtEmpCode.Text=dr[0].ToString().Trim();
   txtFirstName.Text=dr[1].ToString().Trim();
   txtLastName.Text=dr[2].ToString().Trim();
   txtDesignation.Text=dr[3].ToString().Trim();
   txtSalary.Text=dr[4].ToString().Trim();
}
```

SYNGRESS syngress.com

Figure 9.8 Coding to Display the First Record

```
private void cmdFirst_Click(object sender, System.EventArgs e)
{
   if (dtRowColl.Count==0) return;
   CurRec=0;
   DisplayRec();
}
```

The code in Figure 9.8 shows that if the *Count()* function encounters 0, which means that no record is present, nothing is displayed; otherwise, the first record is displayed. For this, the private function *DisplayRec ()* is called.

SYNGRESS syngress.com

Figure 9.9 Coding to Display the Last Record

```
private void cmdLast_Click(object sender, System.EventArgs e)
{
   if (dtRowColl.Count==0) return;
   CurRec=dtRowColl.Count-1;
   DisplayRec();
      }
```

The code in Figure 9.9 shows that if *Count()* encounters 0, nothing would be displayed, otherwise *DisplayRec()* function is called and the last record is displayed to the user.

Figure 9.10 Coding to Display the Previous Record

```
private void cmdPrevious_Click(object sender, System.EventArgs e)
{
   if (dtRowColl.Count==0) return;
   CurRec–;
   if (CurRec<0) CurRec=0;
   DisplayRec();
}
```

Figure 9.10 illustrates how the previous record is viewed. For viewing these records, the value of *CurRec* is successively decreased by 1. If the value of *CurRec* becomes lesser than 0 (*CurRec<0*), it means that the user is attempting to scroll beyond the first record. To handle such an action, *CurRec* is set to 0, so that the first record is displayed and the action is cancelled.

Figure 9.11 Coding to Display the Next Record

```
private void cmdNext_Click(object sender, System.EventArgs e)
{
   if (dtRowColl.Count==0) return;
   CurRec++;
   if (CurRec>=dtRowColl.Count) CurRec=dtRowColl.Count-1;
   DisplayRec();
}
```

Figure 9.11 shows how to move forward and view the next record. The value of *CurRec* is simply increased by 1. If the user tries to view records beyond the last record, *(CurRec>=dtRowColl.Count)* cancels the attempt by displaying the last record.

Add Record Form

The form in Figure 9.12 allows you to add new records in the table. This form has two buttons, **Save** and **Close**, which handle the tasks of adding a new record and closing down the form, respectively. While entering the new record, ensure

that the Employee Code field is not left blank. If it is and you click the **Save** button, a message will make an entry for the Employee Code field. A new record can be added only after entering a value for the Employee Code field.

Figure 9.12 Add Record Form

Figure 9.13 shows how a connection with the database is established and how the buttons perform their tasks. The complete source code for Figure 9.13 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 9.13 Adding a New Record in the Database

```
string strSql="";
   OleDbConnection Conn;
   OleDbDataAdapter dbCmd;
   if (txtEmpCode.Text.Trim()=="")
   {
      MessageBox.Show("Employee code can't be blank.");
      txtEmpCode.Focus();
      return;
      }

      strSql="Insert into emp Values(" + txtEmpCode.Text + "," +
    (txtFirstName.Text.Trim()==""?"null":"'" + txtFirstName.Text.Trim()
     + "'") + "," + (txtLastName.Text.Trim()==""?"null":"'" +
     txtLastName.Text.Trim() + "'") + "," +
     (txtDesignation.Text.Trim()==""?"null":"'" +
     txtDesignation.Text.Trim() + "'") + "," +
     (txtSalary.Text.Trim()==""?"null": txtSalary.Text.Trim()) +")" ;
```

Continued

Figure 9.13 Continued

```
Conn= new OleDbConnection("Provider=SQLOLEDB.1;Persist Security
       Info=False;User ID=charul;pwd=charul;Initial
   Catalog=employee;DataSource=developers");
          dbCmd=new OleDbDataAdapter();
          dbCmd.SelectCommand = new OleDbCommand(strSql,Conn);

          try
          {
             Conn.Open();
          }
          catch(Exception ex)
          {
             MessageBox.Show(ex.ToString());
             return;
          }

             DataSet empDS=null;
             empDS= new DataSet();

             try
             {
                dbCmd.Fill(empDS);
             }
             catch(Exception ex)
             {
                MessageBox.Show("Str : " + ex.Message);
                return;
             }
             Conn.Close();
             MessageBox.Show("Record Added.");

             txtEmpCode.Text="";
             txtFirstName.Text="";
             txtLastName.Text="";
             txtDesignation.Text="";
             txtSalary.Text="";
```

The code snippet in Figure 9.13 handles the task of adding new records in the table residing on the database. For adding new records in the table, the techniques of *DataSet* and *DataAdapter* are used.

At the start of this code snippet, the string type variable *strSql* is declared. This variable will later be used to store the SQL query for inserting new records in the table. Objects *Conn* and *dbCmd*, respectively, of classes *OleDbConnection* and *OleDbDataAdapter* are created. Object *Conn* will be used for setting up a connection with the database, and object *dbCmd* will populate the *DataSet* with records. Once the required objects and the variable have been declared, it is checked to see whether the Employee Code field (*txtEmpCode*) is empty. If it is, a message is shown prompting the user that this field cannot be left blank.

To insert records in the table, an SQL query is written. While writing the Insert Record SQL query, the problem of entering empty values is taken into consideration. In case the user fails to enter information in some field (except that for Employee Code), the default value NULL is entered in the table.

Following this, the connection with the database is opened to submit the new record to the table. The *Conn* object is used to establish a connection with the database. The *Conn* object is populated with information, such as name of the provider, username and ID, and names of data source and Initial Catalog.

In the Select command property of *dbCmd*, the variable *strSql* and the object *Conn*, are passed. According to these arguments, the SQL query will be executed on the active connection. Inside the *try* block, an attempt is made to open the connection with the database. In case, the connection with the database cannot be opened, the statements written inside the catch block will be executed and, as a result, an error message will appear indicating that the connection with the database could not be opened.

Delete/Update Form

This form in Figure 9.14 allows you to update and delete records. To do so, you first need to select an employee code from the combo box. As soon as you select the employee code from the combo box, other information pertaining to the employee code will be displayed in the controls placed on the form. Once the complete information has been displayed, you can perform necessary modifications on the record or delete the record by clicking the corresponding buttons. Before doing so, you need to establish connection with the database, for which the coding listed in Figure 9.15 can be used. The complete source code for Figure 9.15 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 9.14 Delete and Update Form

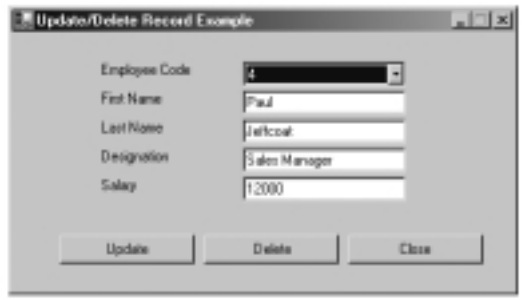


Figure 9.15 Deleting and Updating Records

```
Conn = new OleDbConnection("Provider=SQLOLEDB.1;Persist Security
   Info =False;UserID=charul;pwd=charul;Initial Catalog=employee;
   Data Source=developers");

      OleDbCommand empCmd = Conn.CreateCommand();
      empCmd.CommandText = "Select emp_code from emp";
      OleDbDataReader empReader=null;
      try
      {
         empReader=empCmd.ExecuteReader();
      }
      catch(System.Data.OleDb.OleDbException ex)
      {
         MessageBox.Show(ex.Message);
         this.Close();
         return;
      }

      while(empReader.Read())
      {
      cboEmpCode.Items.Add(empReader.GetValue(0).ToString());
      }
      empReader.Close();
```

The object *Con* is used to make the connection with the database. It is populated with information such as the type of provider, username or ID, password and name of data source and database. The *OleDbCommand* object *empCmd* is used to execute the SQL query on the database. With the help of the SQL query, all employee codes are picked up from the *emp_code* field of *emp* table. This query is submitted to *CommandText* property of *empCmd*

Once the connection string and the SQL query have been written, an attempt is made to open the connection with the database using the *Open* method of the *Con* object. The *try* and *catch* blocks are used to establish connection. The connection with the database is opened inside the *try* block. If the connection fails, the statements written inside the catch block will execute and the attempt for connection will be shut down.

Once the connection with the database is established, the role of *empReader* begins. The *empReader* is filled with the *ExecuteReader()* method of *empCmd*. The *empReader* reads all records (*emp_code*) and stores them in a combo box. Once all records have reached their destination, *empReader* is closed. Figure 9.16 shows the code to display records. The complete source code for Figure 9.16 can be found on the companion Solutions Web site for the book (www.syngress .com/solutions).

SYNGRESS
syngress.com

Figure 9.16 Code to Display Records

```
   OleDbCommand empCmd = Conn.CreateCommand();
   empCmd.CommandText = "Select * from emp where emp_code=" +
cboEmpCode.SelectedItem.ToString().Trim();


   OleDbDataReader empReader=null;


try
   {
      empReader=empCmd.ExecuteReader();
   }
   catch(System.Data.OleDb.OleDbException ex)
   {
      MessageBox.Show(ex.Message);
      this.Close();
      return;
```

Continued

Figure 9.16 Continued

```
}

ClearForm();
if (empReader.Read())
{
   txtFirstName.Text=empReader.GetValue(1).ToString().Trim();
   txtLastName.Text=empReader.GetValue(2).ToString().Trim();
   txtDesignation.Text=empReader.GetValue(3).ToString().Trim();
   txtSalary.Text=empReader.GetValue(4).ToString().Trim();
}
   empReader.Close();
```

The code snippet in Figure 9.16 handles the task of displaying records related to the employee code selected from the combo box. As soon as you select the desired employee code, all records pertaining to the selected employee code will be displayed in various text boxes placed on the form. To do this, an SQL query is written in the *CommandText* property of *empCmd*. This SQL query will fetch all the records related to the employee code selected from the combo box. The results returned by the SQL query are stored in *empReader* using the *ExecuteReader()* method of *empCmd*. After storing the records in *empReader*, the records are read from it and displayed on the form.

If the records could not be stored in *empReader*, an appropriate error message is displayed, the Data Reader is closed, and all controls placed on the form are cleared. Figure 9.17 shows how a record is updated. The complete source code for Figure 9.17 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 9.17 Code to Update a Record

```
string strSql="";
   OleDbCommand empCmd = Conn.CreateCommand();

   if (cboEmpCode.SelectedIndex==-1)
   {
      MessageBox.Show("No employee code selected.");
```

Continued

Figure 9.17 Continued

```
        cboEmpCode.Focus();
        return;
    }
        strSql="update emp set  emp_firstname='" + txtFirstName.Text +
"',emp_lastname='" + txtLastName.Text + "',designation='" +
txtDesignation.Text + "',salary=" + txtSalary.Text + " where emp_code="
+ cboEmpCode.SelectedItem.ToString().Trim();

        empCmd.CommandText= strSql;
            int iRows;
            try
            {
                iRows=empCmd.ExecuteNonQuery();
            }
            catch(Exception ex)
            {
                MessageBox.Show(ex.Message.ToString());
                return;
            }

            if (iRows>0)
            {
                MessageBox.Show("Record updated");
            }
```

The code snippet in Figure 9.17 executes when the Update button is clicked. The Update button is used to update the records in case any changes are made to them. For making an update, you must first select an employee code. If the Update button is clicked without selecting an employee code, the error message "No employee code selected." is displayed, and focus is set on the combo box.

For updating the records, a SQL statement that updates all related records of the employee code selected is written. This SQL query is assigned to the previously declared *strSql* variable and is passed to the *CommandText* property of *empCmd*.

An integer type variable is declared to determine whether the record is updated. For this purpose, the update SQL query is executed using the

ExecuteNonQuery() method of *empCmd*. The *ExecuteNonQuery()* method returns the number of rows affected in the table. The number of affected rows is stored in *iRows*. If update SQL query could not be executed (number of rows affected is zero), an error message box is shown. If the number of rows stored in *iRows* is greater than 1, which indicates successful update, the message "Record updated" is displayed. Now, let's discuss deletion of records. Figure 9.18 contains the code for deleting a record. The complete source code for Figure 9.18 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 9.18 Code to Delete a Record

```
OleDbCommand empCmd = Conn.CreateCommand();

   if (cboEmpCode.SelectedIndex==-1)
   {
      MessageBox.Show("No employee code selected.");
      cboEmpCode.Focus();
      return;
   }

   empCmd.CommandText = "delete emp where emp_code=" +
cboEmpCode.SelectedItem.ToString().Trim();
   int iRows=empCmd.ExecuteNonQuery();

if (iRows>0)
   {
      cboEmpCode.Items.Remove(cboEmpCode.SelectedItem);
      ClearForm();
      MessageBox.Show("Record deleted");
   }
```

The code snippet in Figure 9.18 comes into action when the Delete button is clicked to delete an employee record. To delete a record, you first must select an employee code from the combo box. If you don't, the message, "No employee code selected." is shown, and focus is set on the combo box.

For deleting records, an SQL query is written that will delete all the records of the employee code selected by the user from the combo box. To

make sure the records have been deleted, the delete SQL query is executed using the *ExecuteNonQuery()* method of *empCmd*. As mentioned earlier, this method will return the number of rows affected in the table; the number of rows affected being stored in variable *iRows*. If the number of rows affected in the table is greater than zero, it means that all the records of the selected employee code have been deleted from the form and that that employee code has been removed from the combo box. We need to reset the controls on the form as well, as shown in Figure 9.19. The complete source code for Figure 9.19 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 9.19 Function to Clear the Controls Placed on the Form

```
txtFirstName.Text="";
   txtLastName.Text="";
   txtDesignation.Text="";
   txtSalary.Text="";
```

This coding belongs to a private function, which clears (i.e. hides, it does not delete) all the records from various controls placed on the form. We also need to validate the user input for *txtSalary*.

```
if (e.KeyChar==(char)8)
   {
      return;
   }
   if (e.KeyChar<'0' || e.KeyChar>'9')
   {
      e.Handled=true;
   }
```

The validation goes within the *KeyPress* event of *txtSalary*. This coding restricts the user from entering non-numeric data in the text box. The text box accepts only Integer type values and allows the Backspace key to operate.

Converting Binary Data Using Base64

In the previous section, you learned how the client application accesses the database residing on the server for processing data, and also about the role of

ADO.NET as a bridge between the client application and the server. The data retrieved from the server by the client application was in the simplest form, being just plain text accompanied by some integer values. In addition, while submitting data back to the server, the client application was not allowed to send arbitrary data such as binary data, nor was there any provision to handle such data. For example, in the previous section, you were sending very common information such as name, address, telephone number, salary, and so forth. However, your application is not limited to sending such fundamental data. There is the provision for handling binary data such as images. This section explains how the application handles binary data.

Base64, a binary conversion algorithm, converts binary data into plain ASCII text, which is subsequently stored as a nonsequential string of text. The data sent using Base64 could be any arbitrary data, which is encoded into plain ASCII text before transmitting over the network. The recipient application decodes this data, then rebuilds it, and the content the data represents is displayed.

How Base64 Works

The encoding process represents 24-bit groups of input as output strings of four encoded characters. Proceeding from left to right, a 24-bit input group is formed by concatenating 3*8-bit input groups. These 24 bits are then treated as four concatenated 6-bit groups, each of which is translated into a single digit in the Base64 alphabet. When encoding a bit stream via Base64, the bit stream must presume to be ordered with the most significant bit first. That is, the first bit in the stream will be the highest-order bit in the first 8-bit byte, the eighth bit will be the lowest-order bit in the first 8-bit byte, and so on. Table 9.3 shows the Base64 matrix table.

Table 9.3 The Base64 Matrix

Value	Char	Value	Char	Value	Char	Value	Char
0	A	16	Q	32	G	48	W
1	B	17	R	33	H	49	X
2	C	18	S	34	I	50	Y
3	D	19	T	35	J	51	Z
4	E	20	U	36	K	52	0
5	F	21	V	37	L	53	1

Continued

Table 9.3 Continued

Value	Char	Value	Char	Value	Char	Value	Char
6	G	22	W	38	M	54	2
7	H	23	X	39	N	55	3
8	I	24	Y	40	O	56	4
9	J	25	Z	41	P	57	5
10	K	26	A	42	Q	58	6
11	L	27	B	43	R	59	7
12	M	28	C	44	S	60	8
13	N	29	D	45	T	61	9
14	O	30	E	46	U	62	+
15	P	31	F	47	V	63	/

Let's consider an example to understand the Base64 algorithm. Suppose Application A sends an XML file containing an image to Application B over the network. While sending the image file to Application B, the entire data of the file (image) is encoded in plain ASCII text using Base64. The encoded image file would appear as shown here:

```
<Image>
 <bmp>Qk24QwAAAAAAALYDAAAoAAAAgAAAAIAAAAABAAgAAAAAAAAAAADDDgAAww4AAOAAAA
DgAAAA////AGNjawBKSnMAQkqcAEJKpQBKUnsASlqtAEJSpQBCUq0AOUqtAFJjrQBSY7UAS
lqlAGNrjABSY6UASlqcAEJatQBaY4QAWmulAFJjnABKWpQAQlKMAEpjtQBCWq0Aa3OMAGNr
hABSWnMAY3OlAFprnABSY5QASlqMAFJrtQBKY60AQlqlAGNznABac7UAUmutAEpjpQBaa5Q
AWnOtAFJrpQBCWpQASlJjAGt7nABjc5QASlp7A.................................
.........................................................................
.........................................................................
.........................................................................
.........................................................................
6OZo6OjmaOZo5mjmafn5+Nn42Nn42NZceNno2Ojp+Nn2afjWXHZY2fjo6OoGaOoI6gjqCOg
o4mkIODaL6+19fXm1y6n46OZo5mjmaOjo6Ojo7In42eZcdmjo6gjmafx2XHno2fjp6NjY6O
n43Hn2aOjp+fjZ6NnmGNjY2fjZ+Nn5/Ijo5myI6OoI6gjqCOjo6gn6BmoGbIjp+fZY2fjp+
Nx46CyJ+Nno2Nn46OjqCOZo5mjmaOji0/g72qvr6r3LeblsiNn46gjqCOoGagZqBmoGagZp
+NjZ6NZo6gjqBmn41lx2Wfjo1lx2XIjmaNjY2fZo6Ojo6fn5+Nn42fjZ+Nno1lx42Nno2Nj
Y2NjY2fZo6Ojo6Ojo5moI6fx42Cjp6NZo6Ogo6fZcdljZ+OZo6gjqCOjqCOP1Amj9qqvLzL
lmhcjWXHjY2OZo6Ojo6Ojo6Ojo6Ojo6fn42NjY1mjo6OoI2fx2WNn8hmjWWNn8hmn42NjY2
fZo6OjmaOZo5mjmbIZshmyGafn5+fn5+fjceNno2Nn46foI5mjo6ejY6OjZ6Nn46Ojo6OjZ
5hno2OjmaOjoKOjn9RhWfLvavVvb2W2KuOjcdlx2WNoGagjo6gZqBmoGagZqBmoJ+NZceNj
Z+OjoKOn5/HZcefjo2NZY2Nn42Nno2NjZ+Ojo6Ojo6Ojo6Ojo6Ojo6Ojo6OjmaOn5+f
```

```
jY2NjqCOgp+NZY2On43HjWaOoGbOjY2ejY2fjo6gjo6gSlAmw729vb3a1bebZMifZp7HZce
Njo5mn46Ojo6Ojo6Ojo6Ojo6fn56Nno2fjo6OjmafjY2ejY2Nnsdlx56Nnsdlx2XHZcdlx2
XHZcdlx2XHZcdlx2XHZcdlx41mjo6fno2NZo6OjmbIjZ6Njmafno2Ojo6On56NnseNoEldj
ogtAAA=</bmp>
</Image>
```


When Application B receives this encoded data, it reconverts it into its native format (bmp or jpeg format) for display.

Converting Binary Data into Base64 Format

Our first example of working with Base64 illustrates how binary data is converted into plain ASCII text using Base64 and stored in a XML file. The second example shows how the Base64 formatted data is read and how its contents are displayed. In the implementation process of Base64, the binary data that we are converting is an image. This image is converted into Base64 format and stored under bmp node of the XML file.

Database Design

The structure of the table named as myImages is provided in Figure 9.20. This table is used for storing the name of the image and the length of the image.

Figure 9.20 Database Design of the myImages Table

Column Name	Data Type	Length	Allow Nulls
Image_name	varchar	20	✓
Img	image	16	✓

The form shown in Figure 9.21 acts like an interface for the user to save the image in an XML file in Base64 format. The **Save** button placed on the form performs this task.

Figure 9.21 Write XML Form

SYNGRESS syngress.com

Figure 9.22 Setting Up a Connection with the Database During Form Load

```
private void frmWriteXml_Load(object sender, System.EventArgs e)
{
      Conn=new OleDbConnection("Provider=SQLOLEDB.1;Persist Security
Info=False;User ID=charul;pwd=charul;Initial Catalog=employee;Data
Source=developers");
      try
      {
         Conn.Open();
      }
      catch(System.Data.OleDb.OleDbException ex)
      {
      MessageBox.Show(ex.ToString() + "Connection Fail.");
         this.Close();
      }
   }
```

The code in Figure 9.22 (which can be found on the companion Solutions Web site for the book (www.syngress.com/solutions) initiates the connection process with the database. While connecting to the database, information such as the names of the provider, user, database, data source and user password is passed and stored in *Con* (an Object of *OleDbConnection* class). As usual, an attempt is made to open the connection with the database. In case the attempt fails, an exception is raised, the user is informed about the cause of the failure, and the connection process is closed.

Once connection with the database is established, the form is ready to write an XML file in binary format using the Base64 algorithm. Figure 9.23 shows our Base64 file ready to go as an XML file. The complete source code for Figure 9.23 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS syngress.com

Figure 9.23 Coding to Save the Base64 Encoded Image in an XML File

```
   private void cmdSave_Click(object sender, System.EventArgs e)
   {
      if (txtImageName.Text.Trim()==="")
```

Continued

Figure 9.23 Continued

```
        {
            MessageBox.Show("Image name can' be empty");
            txtImageName.Focus();
            return;
        }
        if (txtXmlFile.Text.Trim()==="")
        {
            MessageBox.Show("Xml filename can' be empty");
            txtXmlFile.Focus();
            return;
        }
            OleDbDataAdapter dbCmd;
            DataSet dtSet;
            DataRowCollection dtRowColl;

            dtSet=new DataSet();
            dtSet.Tables.Add("myImages");
            string strSql;
            strSql="Select img from myImages where image_name='" +
txtImageName.Text.Trim() + "'";
            dbCmd= new OleDbDataAdapter(strSql,Conn);
            dbCmd.Fill(dtSet,"myImages");
            dtRowColl=dtSet.Tables[0].Rows;

            if (dtRowColl.Count==0)
            {
                MessageBox.Show("Image not found!");
                return;
        }
            DataRow dRow=dtRowColl[0];
            byte[] byteImage=(byte[])dRow[0];

            XmlTextWriter xmlWrite= new XmlTextWriter("c:\\temp\\" +
txtXmlFile.Text,null);
```

Continued

Figure 9.23 Continued

```
      xmlWrite.Formatting = Formatting.Indented;
         xmlWrite.Indentation= 5;
         xmlWrite.Namespaces=false;
         xmlWrite.WriteStartDocument();
         xmlWrite.WriteStartElement("Image","");

         try
         {
            xmlWrite.WriteStartElement("","bmp","");
xmlWrite.WriteBase64(byteImage,0,byteImage.Length);
            xmlWrite.WriteEndElement();
         }
         catch(Exception ex)
         {
            MessageBox.Show (ex.ToString(),"Error!!");
            return;
         }

         xmlWrite.WriteEndElement();
         xmlWrite.WriteEndDocument();
         xmlWrite.Flush();
         xmlWrite.Close();
         MessageBox.Show("Xml Copied...");

      }
```

The code in Figure 9.23 works when the user clicks the **Save** button to save the XML file in Base64 format. How the coding provides such a mechanism is explained in the lines that follow. It is important to recall that if the user leaves any field blank and clicks the Save button to trigger the process, an error message is displayed.

An object *dbCmd* of *OleDbDataAdapter* class is created, which will take the responsibility of handling the data returned by the database and forwarding it to *DataSet*. To receive the data created by *dbCmd*, the *dtSet* object of the

DataSet class is created. Apart from these objects, another object *dtRowColl* of *DataRowCollection* is created to represent the collection of rows present in the memory resident table in *dtSet*. Once all the required objects have been created in *dtSet*, the table residing on the database named *myImages* is added up and a SQL query is written to retrieve the image from *myImages* table. To run this SQL query, *dbCmd* is used. It will execute the SQL query on active connection. Once the SQL query is executed, the *dtSet* is filled with the data of *myImages* table, and from the collection of tables in *dtSet*, the first table is picked up and the collection of rows of this table is stored in *dtRowColl*.

If it is found that no rows are present in the collection of rows (*dtRowColl.Count==0*), which indicates that no record is present, an error message is shown to the user stating that no image is found.

On the other hand, if rows are present in the table, the first row from the collection of rows is picked up and stored in object *dRow* of *DataRow* class, which represents individual rows in the collection of rows.

Although there could be multiple tables and rows in data being handled by the dataset, in this example there is only one table, which comprises only one field. That is why we are choosing the first table and the first column.

Once the row has been received, its contents (image) are stored in *byteImage* array of *byte* type, and an *xmlWrite* object of *XmlTextWriter* opens the *Write* stream to write the length of the file on the specified location. Before writing the document, its formatting, indentation, and so forth are set. The process of writing the document avails the *WriteStartDocument()* method.

While writing the document, the first element of the document is set as *Image* and the subelement is set as *bmp*. After setting the elements for writing the XML file, the length of the file is written in binary format using the *WriteBase64()* method of *xmlWrite*. This function will write the file in binary format on the specified location mentioned when opening up the *xmlWrite* in write stream.

After writing down the file, the subelement is closed. In case the file could not be written using *WriteBase64()*, an exception is raised.

If the file is successfully copied, a message to this end appears. Finally, the element *Image* is closed and all the resources captured by the process of writing the binary data in the XML file are released using the *Flush()* method. Then, the *xmlWrite* is closed to finish the process. Figure 9.24 shows the simple code to close the form. The complete source code for Figure 9.24 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 9.24 Code to Close the Form

```
private void cmdClose_Click(object sender, System.EventArgs e)
   {
      Conn.Close();
      this.Close();
   }
```

The previous code shows the *Close* function placed on the form. As soon as the user clicks on it, the connection with the database closes down followed by the closure of the form.

Reading Base64 Encoded Data from an XML File

This example illustrates how the Base64 encoded data, which is stored as an XML file, is decoded and rebuilt to display the image. Figure 9.25 is a screenshot of the form.

Figure 9.25 View an XML Form

The form in Figure 9.25 takes the name of the image, which is stored in the XML file. After entering the name of the XML file, when the user clicks the **View** button, another form appears with the image set in the background (Figure 9.26). The code in Figure 9.27 will help you better understand how it works. Figure 9.26 shows our sample output, and Figure 9.27 lists the code that we are going to use for viewing the image within the XML file. The complete source code for Figure 9.27 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 9.26 Form to View the Image

Figure 9.27 Code to View the Image Stored in the XML File

```
private void cmdView_Click(object sender, System.EventArgs e)
{
    frmViewBitmap frmView=new frmViewBitmap();
    frmView.SetXmlFileName("c:\\temp\\" + txtXmlFile.Text);
    frmView.ShowDialog();
    frmView=null;
}
```

The code snippet in Figure 9.27 works for the View button, which displays the XML file entered by the user. To view the XML file entered by the user, a new instance, *frmView*, of form *frmViewBitmap* is made. In the *SetXmlFileName* property of *frmView*, the source destination of the XML files is mentioned, along with the name of the XML file entered by the user. Finally, the form is displayed using the *ShowDialog* method of *frmView*. Figure 9.28 lists the code to close the form, Figure 9.29 lists the code to retrieve the XML file, and Figure 9.30 lists the code for reading the image using Base64. The complete source code for Figures 9.28 through 9.30 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

Figure 9.28 Code to Close the Form

```
 private void cmdClose_Click(object sender, System.EventArgs e)
{
     this.Close();
}
```

Figure 9.29 Function to Retrieve the XML Filename

```
public void SetXmlFileName(string strXmlFile)
{
     strXmlFileName=strXmlFile;
}
```

This public function holds the name of the XML file from which the content of the image will be read.

Figure 9.30 Function to Read the Image from the XML File Using Base64

```
private void ViewBitmap_Load(object sender, System.EventArgs e)
{
    XmlTextReader readXml= null;

    try
    {
        readXml= new XmlTextReader(strXmlFileName);
        int len = 64*1024; //reads upto 64 kb of data
        byte[] byteBmp = new byte[len];

        while(readXml.Read())
        {
            if(readXml.NodeType == XmlNodeType.Element)
            {
                if(readXml.LocalName.Equals("bmp"))
                {
                    readXml.ReadBase64(byteBmp,0,len);
                }
            }
        }

    readXml.Close();
    readXml= null;

    FileStream streamFile=File.OpenWrite("c:\\temp\\temp.bmp");
    streamFile.Write(byteBmp,0,byteBmp.Length);
    streamFile.Close();
    streamFile=null;
    Image img;
    img=System.Drawing.Bitmap.FromFile("c:\\temp\\temp.bmp");

    this.BackgroundImage =img;
    this.Width=img.Width;
```

Continued

Figure 9.30 Continued

```
        this.Height=img.Height;
        img=null;

    }
        catch(Exception ex)
        {
            MessageBox.Show(ex.ToString());
        }

        if(readXml!= null)
        {
            readXml.Close();
        }

    }

private void frmViewBitmap_Deactivate(object sender, System.EventArgs e)
    {
        this.BackgroundImage=null;
    }
```

The previous code displays a private function that handles the task of reading the contents of an XML file and converting its bmp node (which carries the image) into binary format using the Base64 algorithm. Once the node is converted into binary format, it is displayed to the user as a background image of the form. To explain the encoding process of Base64, we will use the images stored in the XML file under *bmp* node. The coding of the function is explained next.

This function can be split into three phases. First, the XML file is *read* and the contents under the *bmp* node are retrieved and converted into binary format using Base64 functionality. Second, this binary formatted data is *written* on some file and saved on a particular destination. Finally, this binary data is *displayed* to the user in the form of an image set in the background of the form.

First, an object *readXML* of *XMLTextReader* is made to read the file passed to it as an argument. This *readXML* object can read data up to 64KB binary data. If

the data read by it is less than 64KB, this function adjusts the data accordingly. If the data exceeds 64KB, this function truncates the excess data.

Once the process of reading the XML file begins, it is verified that the element type in the XML file is *node*, and the type of node is *bmp*. For determining whether the node type is *bmp*, the *Equals()* method of *readXML* is used and node type bmp is passed to it. Once *Node* and *Element* authentication are done, the file is read using the *ReadBase64()* method. While reading the file, three arguments (*byteBmp*, *0* ,*len*) are passed. According to these arguments, the content (image information) stored under the *bmp node* is read and stored in the variable *len* of integer type. After reading it and converting it into binary format, it is assigned to array *byteBmp* of byte type. Once the length of the image file is gained, *readXML* is closed and set to NULL to remove any junk memory left undetected. Up to this point, the file has only been read and has yet to be displayed.

In order to display the file, it first needs to be stored somewhere. For this, an object *streamFile* of *FileStream* class is made, which will open the *Write* stream to write down the file on a specified location with some desired name ("c:\\temp\\temp.bmp"). While writing the file on a specified location, the arguments (*byteBmp*, *0*, *byteBmp.Length*) are passed. According to these arguments, the complete length stored in *byteBmp* array will be written from the starting point of the file. Once the process of writing the file is finished, the stream is closed.

After the file has been read and stored, the task that remains is to display the image. For this, an object *img* of *Image* type is made. In this *img* object, the mentioned image filename is copied as a bitmapped image and is set as the background image of the form. The height and width of the form are set equal to those of the image so that the image fits correctly in the background of the form.

If at any point in this process a failure occurs, an exception is raised, the user is informed of the reason for the failure, and *readXML* is closed, since it should not be left open after raising the exception.

Designing and Implementing a Remote Database Viewer

Developing a business application for an enterprise can be a difficult task if it calls for consistently accessing data from the database for a particular application. To access the relational database for the client-server application, developers use the ODBC. However, for the Web environment, a developer has to either develop his

own data access methods or use the application programming interfaces (APIs) that are compatible with the new environment. A better alternative is to use ADO with OLEDB, which works with almost all data sources for different environments such as client-server, Web, and so forth.

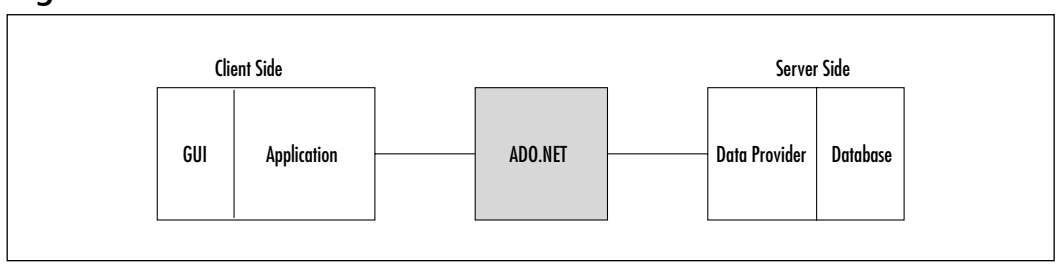
ADO can be used to access data for applications. With this, by using any application language or tool, you can develop front-end database clients or objects at middle-tier level. ADO works as a single interface for accessing data for developing 1-to-n tier client-server, Web-based, or any other database-driven applications.

What Is a Remote Database?

Before we can look at a remote database, we need to understand the traditional client-server architecture, usually called a LAN. A client logs on to the network using a username and password (provided by the server administrator). The server authenticates the username and password of the client and allows him to access the resource of the network. The client can now access the data from the server or send the data to the server using the connection-oriented protocol that supports the LAN. With this, permanent link has been established between the server and the client.

Figure 9.31 schematically shows the client-server application accessing the data store using ADO. The application and the user interface communicate directly with the data store. Various applications use various LAN components to provide the state.

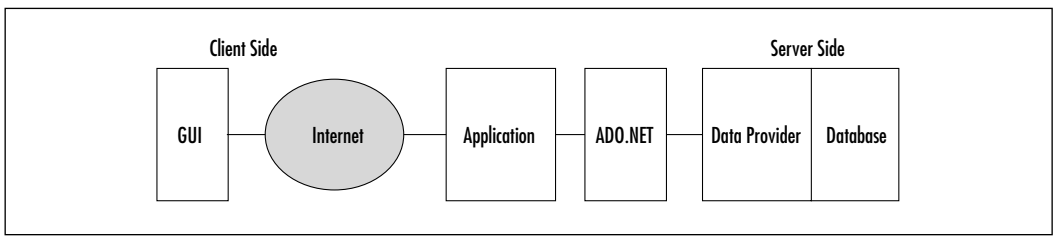
Figure 9.31 Client-Server Architecture



This process fails on the Internet, as the HTTP protocol is used to access the components of the Web, such as any HTML page, image, or database. HTTP is a connectionless protocol that cannot identify the client for the server automatically as in a LAN environment. In the case of the Web, the client sends the request to the Web server using the HTTP protocol. Once the Web server sends

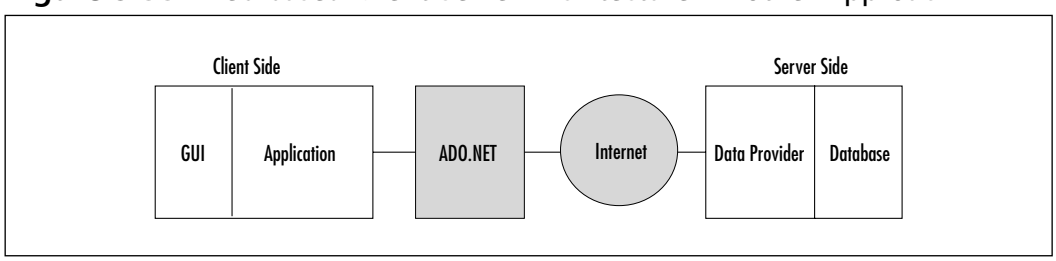
the response in HTML format, the connection at the physical level discontinues. To handle requests related to accessing the database, we require another interface that can understand the SQL query and carry out the processing accordingly. In this approach, the user interface moves over the client and the browser over the Internet. We communicate with the database with a server-based application using the HTTP protocol. The server-based application might be developed using Active Server Pages, CGI, or an ISAPI application written in any language. Figure 9.32 illustrates this concept.

Figure 9.32 Web-based Client-Server Architecture



Now we'll take a quick look at how remote data access can extend the *application* itself rather than just the *interface* across the Internet. In this case, the user interface, application, and ADO move over the Internet at the client machine node. ADO can communicate quite effectively with the database, even over the Internet. This approach is similar to the client-server approach, except it uses the Internet to communicate between the client and the server (Figure 9.33).

Figure 9.33 Web-based Client-Server Architecture: Another Approach



Advantages and Disadvantages of Remote Data Access

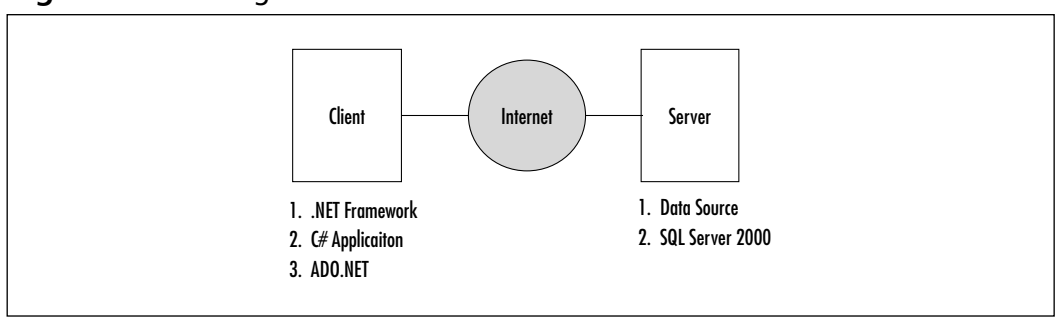
Thus, we can extend the scope of the client-server approach of data management to cover the Internet using remote data access. This approach helps to reduce the

network traffic and saves time as compared to the Web approach, wherein the user interface passes each request to the server for processing. It lessens the demand on the server and reduces processing time.

Consider an example in which a remote user is viewing a list of products from a database, whose order he wants to change. In the traditional method of using ADO with ASP or any other application on the server, the entire recordset must be transferred each time over the Web, whereas with remote data access, the recordset is cached on the client side, where it can be sorted, filtered, and viewed locally. Moreover, no request goes to the server that creates network traffic.

However, the requirement of the application must always be considered. For example, if the customer wants to search a single record from thousands of records, it is better to use the traditional Web approach, which requires sending only one record over the Web. With remote data access, you have to transfer the complete recordset of thousands of records over the Web, which will consume substantial hard disk space and more time. Thus, a judiciously planned mixture of techniques, depending on the nature of the application, is often necessary.
Figure 9.34 explains the components required at both the client and the server sides. The client is installed with the .NET Framework, and the server carries SQL Server 2000. SQL Server 2000 is compatible and can return the result in XML format to the client. The connection between the client and the server has been established using the Internet. You can create the database on SQL Server according to the requirement of the application. The table on SQL Server 2000 can be created using either SQL Enterprise Manager or SQL query analyzer.

Figure 9.34 Design of the Remote Database Viewer



To communicate between the client and server, you can either create the DSN on the server or use the database name. Here we are using the name of the database for communication. The connection between the client and server can

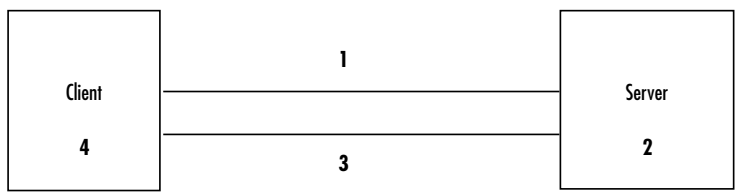
be established by using the connection string object of ADO.NET. The connection string carries the following information:

- IP address of the site
- Database name
- Username
- Password

You can provide either the database name or the name of the DSN created at the server in the connection string.

Figure 9.35 explains how the client communicates with the server to access the database. The user can view the result of the query, as well as edit the database.

Figure 9.35 Implementation of the Remote Database Viewer



1. Client sends the SQL query to the server.
2. SQL Server processes the query and generates the result in XML format.
3. Server sends back the result to the client, either in data set or data reader.
4. User can use this object to view, sort, or filter the records.

The client sends the SQL query for accessing the database to the server using the cmdobject of ADO.NET. For example, if the user wants to access the list of authors in the Pubs database of the SQL server, an SQL query can be passed from the client to "select authors from pub" in the cmdobject of ADO.NET.

This SQL query is passed to SQL Server 2000 for processing, and the SQL server generates the result in XML format. This result is sent back to the client in either a data set or data reader. If the client supports the XML, the result can be viewed without parsing; otherwise, the client requires an XML parser to view the result.

The data set or data reader can be used to sort, filter, or view the result by the user. For example, if the user wants to change the order of the result set, the client is not required to pass the SQL query again to the server. The same can be

sorted by using the object stored at the client side, which saves time and reduces network traffic.

Implementing a Simple Remote Database Viewer

The first step toward implementing a remote database viewer is to initiate the connection with the database residing on the server. As mentioned earlier, while establishing connection with the database, information such as the IP address of the data source, username, and password, and name of the database must be provided. Other important information that needs to be supplied is the name of the provider. The name of the provider tells the type of data source. In simple words, we can say that the provider tells whether the data source is SQL Server, Oracle, or Access. All these information is passed in the object of the *OleDbConnection* class, which handles the task of establishing and opening the connection with the database. The following code snippet will help you to better understand the concept.

```
Conn = new OleDbConnection("Provider=SQLOLEDB.1;Persist Security
Info=False;User ID=charul;pwd=charul;Initial Catalog=employee;Data
Source=192.168.1.100");
      try
   {
      Conn.Open();
   }
   catch(System.Data.OleDb.OleDbException ex)
   {
      MessageBox.Show(ex.ToString() + "Connection Fail.");
      this.Close();
      return;
   }
```

With reference to the preceding code, in the object *Conn* of *OleDbConnection* class, all the required information is passed and an attempt is made to open the connection to the database. If the connection cannot be established, an exception is raised, and the message "Connection Failed" appears.

Once the connection with the database is established, you need to send a request to the data source to view the data of a particular table. For this, you need to write a SQL query carrying the desired request and execute it. The *OleDbDataAdapter* class will help you in carrying the SQL query written by you, executing it and handling the data returned by the data source.

```
OleDbDataAdapter empCmd = new OleDbDataAdapter("Select * from emp",Conn);
```

In the coding for executing the SQL that follows, you will see that all the records of *emp* table are fetched and assigned to *OleDbDataAdapter*. The data returned by the data source is an XML file representing the records of table in textual format as depicted in the following code snippet.

```
<NewDataSet>
  <emp>
    <emp_code>4</emp_code>
    <emp_firstname>Paul</emp_firstname>
    <emp_lastname>Jeffcoat</emp_lastname>
    <designation>Sales Manager</designation>
    <salary>12000</salary>
  </emp>
  <emp>
    <emp_code>501</emp_code>
    <emp_firstname>John</emp_firstname>
    <emp_lastname>Smith</emp_lastname>
    <designation>Sr. Manager</designation>
    <salary>20000</salary>
  </emp>
  <emp>
    <emp_code>102</emp_code>
    <emp_firstname>Benjamin</emp_firstname>
    <emp_lastname>Egan</emp_lastname>
    <designation>Programmer</designation>
    <salary>12000</salary>
  </emp>
</NewDataSet>
```

This XML file is handed over to the *DataSet*. Once the data reaches the *DataSet*, all the rows are picked up and stored in *DataRowCollection*, which represents all the rows in the *DataSet*.

```
    DataSet ds=new DataSet();
    try
    {
```

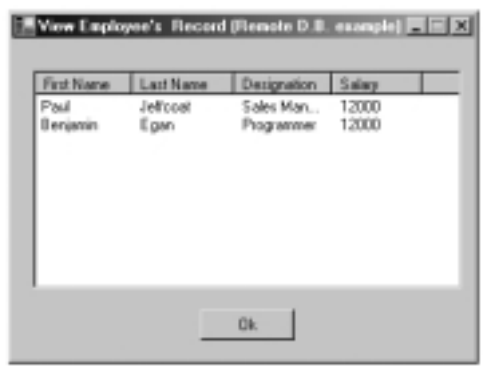
```
        empCmd.Fill(ds,"emp");
    }
    catch(Exception ex)
    {
    MessageBox.Show(ex.ToString());
    }

    DataRowCollection drCol=ds.Tables[0].Rows;
```

Now that we have procured the records (rows) of the table, the only task that remains is to display the records in some GUI-based format so that the records can be viewed and easily understood. The list box can be used to display the collection of rows.

```
    foreach  (DataRow dr in drCol)
    {
        ListViewItem lstItem=new ListViewItem();
        lstItem.Text=dr[1].ToString().Trim();
        lstItem.SubItems.Add(dr[2].ToString().Trim());
        lstItem.SubItems.Add(dr[3].ToString().Trim());
        lstItem.SubItems.Add(dr[4].ToString().Trim());
        lstEmpView.Items.Add(lstItem);
    }
```

After you execute all the code snippets listed for implementing the Remote Database Viewer, the final output would appear as shown in Figure 9.36.

Figure 9.36 Output of the Remote Database Viewer Application



Summary

In this chapter, we saw how XML can be used for communicating between the .NET application and the database server by virtue of the ADO.NET architecture. We discussed the designing and the implementation of a Remote Database Viewer, and explored the process for retrieving binary format data that is stored in a database and writing them in an XML file in Base64 format.

We learned how, through the ADO.NET architecture, the client retrieves a copy of data from the server, and after some modification, sends the data to the server using ADO.NET. Once connection between the client and the server is established, this connection works like a bridge between them to help retrieve and send data between the client and server. Sample programs on the ADO.NET emphasized the connectionless architecture on which ADO.NET is built.

The contents of this chapter should have equipped you with enough knowledge to appreciate fully the use of ADO.NET with XML in the .NET environment, and design and implement our example, the Remote Database Viewer.

Solutions Fast Track

Understanding ADO.NET

- ☑ ADO.NET is .NET's new ADO architecture.
- ☑ ADO.NET relies on XML to provide communication between tiers, without requiring that proprietary information be exchanged.
- ☑ ADO.NET can support different types of data, including XML and database.

Accessing Data from a Database Using ADO.NET

- ☑ ADO.NET helps to simplify the communication process for data retrieval between a client application and a database.
- ☑ *DataSet* is a miniature relational database in which the data is kept in memory.
- ☑ *DataReader* holds a stream of data from the database for a query operation.
- ☑ *DataReader* is like forward-only or read-only *RecordSet*.
- ☑ The main objects in ADO.NET are *Connection*, *DataReader*, and *DataAdapter*.

- ☑ In the ADO.Net architecture, XML is the key to remoting.

Converting Binary Data Using Base64

- ☑ In the Base64 conversion, all the data is converted into the 64-character set range from A to Z, a to z, 0 to 9, /, + .
- ☑ *System.Xml* namespace provides the libraries to convert the data into Base64 format.

Designing and Implementing a Simple Remote Database Viewer

- ☑ Remote data access allows the client to connect to a database over the Internet, thus reducing network usage.
- ☑ Previous implementations of remote data access failed or did not provide adequate protection, when creating the connection over the Internet because of the HTTP protocol.
- ☑ ADO.NET is able to circumvent many of the issues related to HTTP.

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** and click on the **"Ask the Author"** form.

Q: What is a *DataSet*?

A: A *DataSet* is a miniature relational database in which the data is kept in memory for accessing and updating through the collection of objects provided by it. Thus, *DataSet* provides a copy of data for processing on the client machines. In this case, the client would be something such as a client workstation, a Web server, or a remote Internet client.

Q: Which one is faster, the *DataSet* or the *DataReader*?

A: When a large amount of data is being retrieved from the database, the *DataReader* is faster because it provides access to data one row at a time. However, in retrieving small amounts of data, *DataSet* might be faster.

Q: Does, ADO.NET support remote data access?

A: Yes, ADO.Net provides support for remote data access, whether your database is on the local computer, a Web server, or a remote Internet client.

Q: When I work on the copy of data on the client machine, which I receive from the server through the ADO.NET, how is the data updated on the server?

A: To update the data on the server, you would have to explicitly update the data on the server through the connection to the database. You have to call the *CommandUpdate* function of the *DataSet* to update the data on the server database.

Chapter 10

Building a Wholesale Catalog

Solutions in this chapter:

- Basic Design Considerations
- Coding the Project
- XML Packages Design
- Customer Interface Design
- Business and Web Services

☑ Summary

☑ Solutions Fast Track

☑ Frequently Asked Questions

Introduction

You can read many books on XML and .NET, but the only way you can really master them is to do it yourself. Nevertheless, we want to finish this book with real-life applications that incorporate (nearly) everything we covered in this book.

The subject of this chapter is the creation of a business-to-business, or a supply-chain, application that we have named "Wholesale Catalog." We not only want to show you an application using XML, but also an application from which you can derive knowledge and extend to build a full-blown B2B application yourself.

The Wholesale Catalog will deal with a number of issues you face when connecting different systems together, with the exception of using XML as the encoding format of information. In short, the Wholesale Catalog is kept synchronized online with the catalog of different suppliers. Each supplier has its own format of XML documents that needs to be unified for the Wholesale Catalog. What is a wholesaler without customers? Therefore, it will have a client side too that can search and browse through the catalog. Even though we described the catalog concept in a few lines, this chapter will show that it is more complex; however, remember that no e-commerce site works without a well-designed and implemented online catalog. Using XML, the .NET Framework and Web Services, you can do a great job achieving this.

Agora Markets Inc. is a startup e-business. Briefly, it wishes to mimic the old-fashioned Town markets, or Farmers markets in virtual form. Sellers in a traditional market all have separate stalls to display their products, and shoppers can browse through the various stalls, pinch and prod the various products, and compare prices. In the virtual market Agora Markets Inc., suppliers will replace stall-holders. Agora Markets Inc. will maintain a catalog of all the suppliers' products, which customers can browse either by supplier or by product type. Shoppers can select items from this catalog, add items to a standard shopping cart, and pay for their purchases. Agora markets will manage all details such as security and credit card billing. They will notify the suppliers of the orders, and the suppliers will only be responsible for shipping their products. Agora Markets Inc. takes a small percentage of each sale as its commission.

Agora Markets has asked Netforce to design and execute its operation. Central to the entire operation is the maintenance of an up-to-date catalog with all the goods of the suppliers. Because Agora Markets Inc. wishes to compete not only on the excellence of its product, but also on price, it is essential to find an efficient way to update the catalog. Initial discussions make it clear that the best way to do this is to allow the supplier to electronically update the central catalog.

This chapter outlines the design process and code that results in a catalog that can be updated by the individual suppliers.

Before delving into the nitty-gritty of the design process, some philosophical issues need to be addressed:

- How is data best stored? As XML or in a RDBMS?
- What protocols and pathways will be used to pass data to the catalog (e.g., WAN versus Internet, HTTP)?
- What language will be used (e.g., EDI versus XML)?
- What vocabulary will be used?

Basic Design Considerations

This is a book on XML, so it might be natural to expect that everything between its covers would be accomplished using XML—whether this is the best solution or not. However, remembering the well-known adages "When the only tool you have is a hammer, everything tends to look like a nail", or "Just because this is the only tool you know how to use does not necessarily mean it is the best tool for the job," we are not going to follow that course. This chapter is about designing an e-business application using business principles. Thus, we are not going to take a "knee-jerk" reaction to XML; we are going to use whatever tool is best for the project.

Storage: XML versus Traditional Databases

Both XML and a traditional database are great for storing information. Databases are great for storing highly structured information that is going to be kept in a central store. Databases have been optimized over the years for speed of access, frequency of access, and ease of searching. There are numerous RDBMS systems on the market such as DB2, Oracle, and SQL Server, all of which give outstanding performance. However, they tend to be expensive and have a high overhead, both financially and in the amount of maintenance required.

NOTE

Not familiar with RDBMS? Don't be upset about it if you aren't—not even the people who work with RDBMS know they even have one! RDBMS, the acronym for Relational DataBase Management System, is an application that allows you to manage a database and allows for relational database structure.

XML, on the other hand, is well suited to storing irregular information, and information that is going to be kept in a distributed environment. Although XML data stores can be searched almost as easily as a traditional database, the speed of search is considerably slower. The financial overheads of XML are much less than for a traditional database. XML then would appear to be best suited to storing irregular data, to storing data that is going to be maintained in a distributed environment, and where access and searches are going to be relatively infrequent. Such criteria describe many scenarios such as document storage, and medical, legal, and government records.

A health system is an example of a place that can profitably use both methods of storage. Nowadays, most health systems consist of a conglomeration of hospitals and clinics. The nonmedical details of the patient, such as name, age, clinic number, social security number, and so forth, is highly structured data with relatively fixed fields, and is information that is going to be accessed on a regular basis, by all the departments in the health system. These details are best kept in a traditional database. On the other hand, the clinical notes on the patient are highly unstructured data that is usually kept in the various doctors' locations; in other words, in a distributed environment. It is also accessed infrequently, usually when a patient comes to visit. These notes can be profitably kept in an XML data store.

Information Transport Methods

Electronic document interchange (EDI) is traditionally carried out over dedicated value-added networks (VANs) that link partner to partner. This provides tremendous security for the transferred material. However, there are several drawbacks. There must be a dedicated link for each partner, and that is expensive. Using third-party VANs, the charge is usually by the amount of information (bytes) being transferred, so the more information that is pushed through the system, the greater the cost. This factor was one of the design considerations for the terseness of such EDI protocols as the ASC X12 EDI transmission structure (the American Standards group concerned with EDI).

Access to the Internet, on the other hand, is "free." Once a link has been made, there is no extra cost for the amount of information that will be sent. Furthermore, there are no dedicated connections, so anyone who can afford the price of a linkup can potentially become a player—In other words, just about everyone. However, the problem with the Internet, as we all know, is security. The cost of securing our information so it cannot be tampered with can be significant. As with any business decision, we need to weigh the cost of security against the need.

Developing & Deploying…

EDI

EDI allows structured data to be sent electronically. In XML, data is structured using tags. In EDI, the data is structured using line breaks and text delimiters. The way in which the information is ordered is also important. This information is packed and unpacked at each end by special machines called *translators* (compared to XML processors). EDI requires that both trading partners follow exactly the same protocol, and to this end, there are a number of standard ways of encoding data (compared to XML schema). The American Standards Association puts its seal of approval on various standards maintained by various groups and consortia (the ASC x12 standards). Other trade groups also maintain separate protocols (the health industry uses HL7—Health Level 7—protocols).

Those wishing to find out more about EDI are directed to *Electronic Commerce with EDI* by Robert L. Sullivan, which is privately published but available through www.Amazon.com. It is by far and away the best short account of the subject.

Those wishing to get all the advantages of the Internet while maintaining security should consider deploying Advanced Network eXchange (ANX). This is a distributed virtual private networking (VPN) that uses HTTP protocols that provides an industrial-strength foundation for doing e-business. The cost of linking to the ANX net is only marginally more than linking to the Internet, but because all users are registered and must sign an agreement, the majority of casual and malicious hackers are excluded, and because they would be registered, the is a 99.44-percent guarantee that they will be caught, or at least be traced to their workplace.

XML and EDI

When it comes to encoding information for transfer in an e-business application, there are theoretically many ways we could do it, but there are only two ways with any solid credentials, XML and EDI.

EDI has been around for a long time, and has revolutionized many industries and their industrial practices. The major drawback to EDI is the large expense of becoming a player. Any business thinking of deploying EDI is looking at between $50,000 to $250,000 in startup costs, and this is just too much for most small and medium-sized businesses. On the other hand, most businesses have all that is needed to use XML right from the start. Although there are inevitably some entry costs, using XML for B2B information interchange is considerably less expensive. A supplier on an XML-based system need only acquire or build a simple application to become a player. (We will be building such a simple application later in this chapter.) However, XML is much more verbose than EDI. By some estimates, an XML B2B exchange requires 5 to 10 times as many bytes as EDI does. This is an important consideration if one is using a VAN where cost is proportional to the amount of data passed. However, with ANX and the Internet, the cost is fixed, and the same whether 1 or 1000GB are passed.

XML Vocabularies

One problem with XML is that it is just too easy to design a custom language, and so languages and vocabularies tend to multiply. Although organizations such as OASIS (www.oasis-open.org) are attempting to bring standardization to XML languages, XML is still a long way from EDI with its several standard, and standardized, vocabularies all overseen by the American Standards Institute (ASC X12 standards). With an XML B2B transaction, it is often necessary to design one's own language. The problem is that many other players in similar fields will have done the same thing, and they might not be willing to switch. This means that there must be a way to translate from language to another. Luckily, there is: XSLT, and furthermore, ASP.NET has great support for XSLT.

Implementation of the Agora Markets Catalog

Here are the preliminary decisions and rationalization for the implementations in our Agora Markets Catalog.

Data Store

The information in our catalog will be regular and structured. We are also expecting a heavy volume of traffic. We will therefore store all of our information in a relational database. We will use Access to prototype and develop the project, and will migrate to a SQL server at a later date.

Developing & Deploying…

ASP.NET and SQL Server

Both of these products are Microsoft technologies, so it should come as no surprise that ASP.NET will obtain optimal function with SQL Server. Indeed, ASP.NET allots special Connection and Command objects to SQL Server that are much more efficient than their OleDB equivalents are. We will see how to migrate our code to take advantage of these objects at the end of the chapter. Microsoft claims that using the specialized objects can speed processing by up to a factor of 10.

Transport Protocols

Although we anticipate we will be trading with large suppliers, we also expect a large number of small and medium-sized suppliers, and have no wish to cut them out of the market. For this reason, we will not use EDI, but will use an XML-based system instead.

We will not be transferring state secrets, so the ordinary level of security provided by the Internet should suffice. In addition, we would like to encourage "casual" suppliers, and we do not want the (minimal) inconvenience of hooking up to an ANX link to be a bar to entry. For this reason, we will be using XML-based protocols transferred by HTTP over the Internet.

Vocabularies

We could find no suitable XML vocabulary registered with OASIS or elsewhere, so we will design our own XML vocabulary. A registry of schemas and DTDs is maintained by OASIS and XML-ORG at www.xml.org/xml/registry.jsp.

Requirements

Concurrent with making the "philosophical" decisions, the first step in any design project is a discussion with the client to thoroughly understand their business rules, and to draw up a list of their needs and their requirements. The requirements will then be expanded and modified by the programmers to produce a feasible and practical list. After discussion with Agora Markets, and discussion with

and analysis by Netforce programmers, the following lists of requirements for the project are drawn up:

- The catalog data must be held in a readily accessible data store that can be easily updated and searched.
- The catalog must be capable of handling large volumes of requests.
- There must be provision for e-business to e-business updating of the catalog.
- Communication between e-businesses will use XML.
- Information entered into the catalog must be data typed.
- The catalog must have provisions for information about sales, etc.
- The catalog must be easily searchable.
- The catalog must be accessible and searchable as a Web page.
- The catalog must be accessible to business services.
- ASP.NET will be used whenever possible.

Analysis

We will keep this to a minimum here; however, you should remember that in any successful project, the precoding analysis is just as important, if not more important, than the coding itself.

The single most important part of the analysis is to understand the business processes involved. If the programming team does not understand what needs to be accomplished, then all the most sophisticated modeling techniques and programming will produce a useless application. Therefore, before writing a single line of code, a diagram should be made of all the required parts of the application, what each part will accomplish, and how they will fit together. Notes should also be made as to what part of the application will carry out what function, and how information will be passed from one part of the application to another.

Data Store

We have already decided that we will use a traditional database as our data store. Often, the decision as to which database is to be used has been made for us by what is already installed! For our case study, we will use SQL Server; SQL Server

is optimized for use with ASP.NET, is secure, and is easily capable of handling the large volume of requests we anticipate.

Catalog Updating

Agora Markets Inc. will be operating on very low margins, and so will want to keep any human handling of data to a minimum. Most of the larger suppliers will also want to do this, so there must be a way for completely automatic upgrading of the database. However, smaller suppliers might want to manually enter there products into the catalog, and provision must be made for these people by supplying a Web form of some kind that will integrate with the catalog database.

Business-to-Business E-Communications

Two entities that agree to do business with each other electronically are known as *trading partners*. When partners agree to communicate, they must agree on several protocols, one of which is the method of message transmission. In a typical exchange, one trading partner will initiate a sequence of communication:

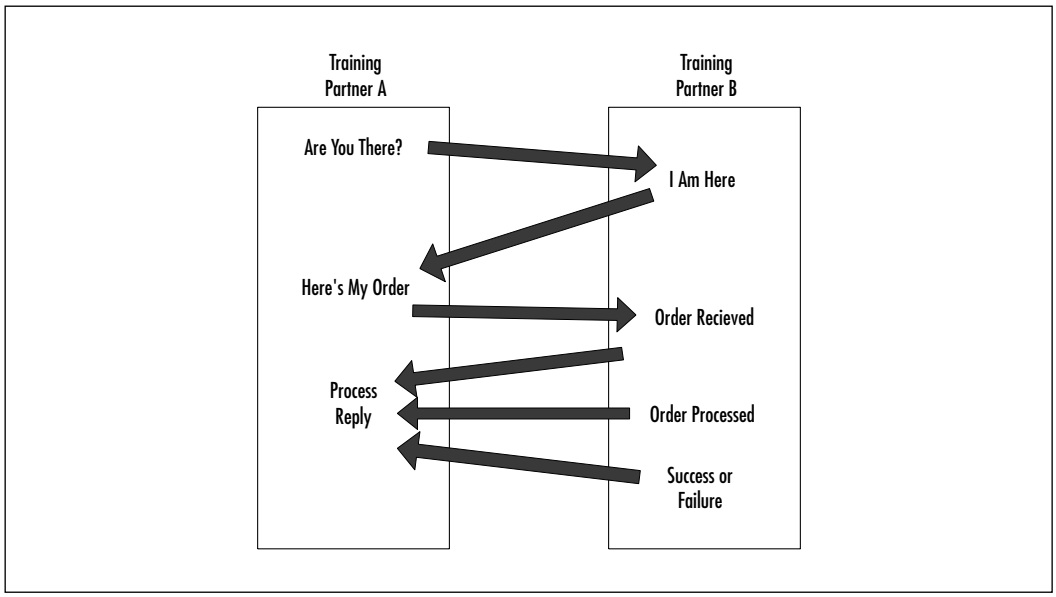
- Partner A: Are you there and ready to receive a message?
- Partner B: I am here and ready to receive a message.
- Partner A: Here is my message…
- Partner B: I have received your message (receipt acknowledgement).
- (Partner B processes the message.)
- Partner B: (If applicable) I have processed your message (process acknowledgment).
- Partner B: This is the message I received, and I have processed it successfully, OR…
- Partner B: This is the message I received, and there was a fatal error (i.e., no processing was performed) OR…
- Partner B: This is the message I received, and there were some errors (success or error acknowledgment).

Depending on the nature of the communication, Partner B might wrap up all the acknowledgments in one packet, or it might send separate acknowledgments. In every case, it is important that it states what message it is replying to. The most common way a reply is sent is to either send a copy of Partner A's message back

with an acknowledgment, or send back a unique ID that identifies the message. When Partner A receives the acknowledgment, it will use its own internal process to react to or record this information in any way that Partner A finds appropriate.

This process is illustrated graphically in Figure 10.1.

Figure 10.1 "Partner" Communication Process



In the case of our wholesale catalog, we are going to use the HTTP protocol on the Internet to carry out the transaction. This will automatically take care of the first part of the transaction. The order in this case will be an order to update the wholesale catalog, Agora Markets (Partner B) will use an ASP.NET page to process the order, and also to generate the acknowledgments, and all the acknowledgments will be sent back simultaneously.

The order will be in the form of an XML file, hopefully in an agreed-upon standard format, and the acknowledgments will also be in the form of a standard XML file.

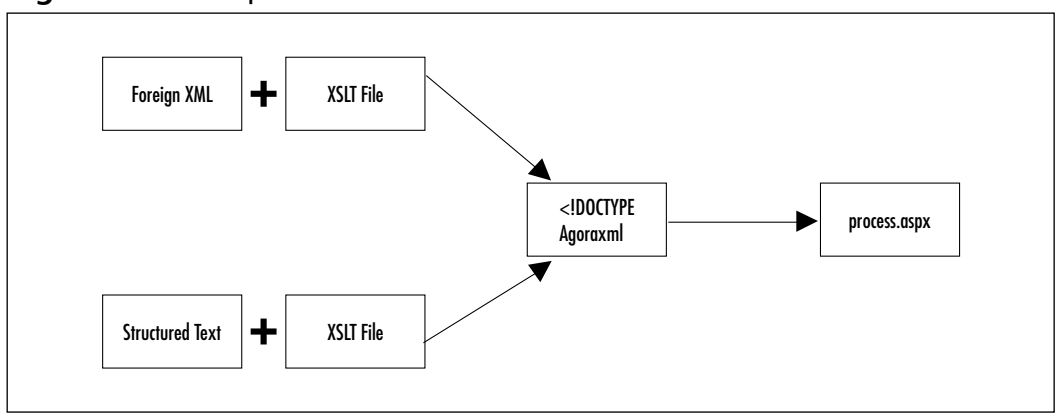
The XML Files

XML is, of course, a meta-language, which is used to build a "language," which is a set of tags defined by both partners and validated via a mutually agreed-upon schema.

In any communication, it is important that the communicants understand one another, and that when they say "Apples," they both mean the same thing.

As an analogy, if two persons of the same culture and language communicate, there is usually not a problem. If people of different languages communicate, then they must either learn each other's language, or employ a translator. Note, however, that some of the more subtle miscommunications occur when two people *think* they understand one another. These types of miscommunication can be eliminated in this application if a common XML format is agreed upon. However, it must be appreciated that potential trading partners might have invested a lot of time and energy in building electronic exchange vocabularies in XML or structured text (for EDI communications) format, so if Agora Markets wants them aboard, then they must be prepared to translate the foreign formats into the one that they employ. Luckily, in most cases, this can be accomplished using XSLT. It should be noted that XSLT can potentially handle formatted text as easily as it can handle XML. Figure 10.2 shows an example of using XSLT to transform XML.

Figure 10.2 Simple Transforms with XSLT



Data Typing Entries

Before any information is entered into our database, we need to data type it to ensure that it is correct. Basically, we have two choices: we can let the database do it, or we can do it prior to submission to the database. In ASP3, the second course of action was preferable for a number of reasons:

- Suitable error messages could be designed.
- Some data types (such as an e-mail address, a zip code, or a URL) are not recognized by databases.
- It is always better to catch an error sooner rather than later.

Often, error trapping of database messages required quite complex and tedious coding. However, now that ASP.NET supports the Try/Catch, error trapping of data mismatches at the DB level becomes much easier for us, and we will make liberal use of this triad in our code.

If we wanted to use custom data types, we could use either "hand-rolled" functions or rely on XML schemas to do the data typing. We will be using standard data types in the application that we will be creating.

Catalog

Let's look at the requirements for our catalog:

- The catalog must have provisions for information about sales.
- The catalog must be easily searchable.
- The catalog must be accessible and searchable as a Web page.
- The catalog must be accessible to business services.

Because we are using a database, we just have to make sure that our interfaces use the power of SQL to efficiently sort the catalog. Luckily, the database programmers have done most of the work of optimization for us already. Since we are also using ASP.NET, providing business services is, as we shall see, extremely simple after we have done the initial catalog design. A full discussion of ASP.NET can be found in *The ASP.NET Web Developer's Guide* from Syngress Publishing (ISBN: 1-928994-51-2).

Coding the Project

The coding process for our project falls neatly into the following five categories:

- Database design
- XML packages design
- Supplier interface and B2B design
- Customer interface design
- Business services

We will look at each of these in more depth in the following sections.

Database Design

A basic understanding of relational database design will tell you that database design is just a case of deciding what information you want to store, designing the records to store this information, and then normalizing the database. If only it were so simple; usually, the design of the database will change radically in the first week or so of the project as programmers realize that their original design is seriously flawed! The trick is to get the database design sorted out before we start writing reams of code to analyze or manipulate the data.

Remember, databases are essentially just repositories of information, and there are two broad uses to which we put that information:

- We store and record transactions of some kind.
- We analyze data that has been stored in the databases somehow.

The overwhelming majority of databases are of the transaction type, and the design of this database aims at eliminating all redundancy of data for both performance and data integrity reasons. Analytical databases are in many respects the polar opposite to this; they are mainly used for analyzing data that has been stored, so there is a lot of reading of data and very little writing. We will take a closer look at these.

OLTP versus OLAP

Online Transaction Processing (OLTP) is concerned with the integrity of the data stored in the database. It is not a good idea to have a name stored in two separate tables in the database. If we do this, we run the risk of changing the name in one location, and not changing it in the other. This will lead to a contradiction in our database. Contradicting databases are said to lack integrity. The way to prevent this from happening is to normalize the database; most databases are normalized to at least the third normal form.

Online Analytical Processing (OLAP), on the other hand, is concerned with analysis of the stored data. With OLAP, it is good practice to have derived data in the database, as this speeds up analysis of the data considerably. Although the catalog part of our DB could be considered an OLAP DB, we will not be carrying out any complex analyses, which allows us to basically have a normalized database.

Developing & Deploying…

Normalization

We have designed a database for you. If you are not familiar with database design and need to be able to design a database, then it is suggested that you consult any of the various books written on this subject.

For those who are rusty about their database design, here is a quick "aide memoir":

There are theoretically six levels of normalization (seven if you include the Boyce-Codd level), but rarely is normalization carried past the third level outside academic circles.

The first normal form is all about eliminating repeating instances of data and ensuring that the data is atomic. In other words, the data is independent and self-contained.

The second and third normal forms depend on The Primary Keys in the records. In a second-level normalized DB, each column in a record must depend on the whole primary key. In the third normal form, no column can have a dependency on anything other than the primary key, and there must be no derived data.

However, consider the following fragments from two different SQL queries:

```
WHERE Qty * UnitPrice > 100

WHERE TotalPrice >100
```

The second query requires no computation and will run much faster than the first, especially in large query sets. If we were dealing with complex analyses, as we often are in OLAP procedures, we could probably eliminate several joins and computations, and so the speed of processing can become very significant in large datasets.

Note that SQL Server 7.0 and later now supports computed columns. These are columns that are derived from other columns, and are built automatically when data is entered into the record. This means that computing is done at the input side, rather than the output side, so again speeding OLAP services. The bottom line is that OLAP services should always be benchmarked for speed of delivery.

XML Packages Design

The XML package that will be used to pass information from the supplier to the catalog needs to be designed with four thoughts in mind:

- It needs to contain all the essential information.
- It should lack ambiguity.
- It should be relatively easy to parse.
- It should as far as possible lack verbosity in order to conserve bandwidth.

As we are expecting to have to convert other document types to our document type, we should not overdo the granularity. With XSLT, it is always relatively easy to go from "more granular" to "less granular."

In our case, we want to be able to fill in all the fields of the table in our database, and we want the form to be able to carry information about sales, special offers, and wholesale deals. This information can be carried either on attributes or as element content. Which should we use?

The argument of "attribute" versus "element" is a longstanding one with really no clear answer, but here are some points to bear in mind:

- Parsing attributes usually take fewer resources than elements do (a tree does not have to be built).
- Quotes, both double and single, can be a problem in attribute values. If you expect your values to contain quotes, then place that value in an element. This is why we will put the product description as element content in our application.
- If you are using a DTD as your schema (as we are), you can make an attribute a *#REQUIRED* attribute.
- When the parser makes lists, attribute lists are unordered. If your code is going to use relationships (e.g., *firstChild*, etc.), then don't use attributes.
- In ASP.NET, if we call for an attribute that is not there, an exception will be thrown. (This is not so in ASP with its "*getAttribute*" DOM method.) There is also no easy way to check to see whether the attribute is present. Therefore, if information is optional and not required, it is best to put it in an element. Although calling for an element that is not there

will still cause an exception, there are easy ways to test for the presence of an element or its content before calling it (*hasChildren*, *getElementsByTagName*).

The design of the XML package file is quite simple. The supplier element is used to pass login information, and the file can contain several batches of product details. Each batch will contain product details requiring the same type of action. They can be updated, inserted (new products), or deleted.

All the information about the products is carried on *#REQUIRED* attributes, except the description, which is put in an obligatory element. Sale and wholesale information is put in separate optional elements.

Figure 10.3 displays a typical XML file that will be passed to our database. This file can be found on the complimentary Solutions Web site (www.syngress.com/solutions) for readers of this book as filename bata2.xml.

SYNGRESS
syngress.com

Figure 10.3 bata2.xml

```
<!DOCTYPE catalog SYSTEM 'catalog.dtd'>
<catalog>
<supplier name="Bata Shoes" uid="bata" pw="bataboy"/>
<batch batch_type="cat_new">
<product prod_code="shoesformalblack1" prod_type="shoes"
prod_name="Formal Black Shoes" prod_class="footware" prod_price="49.95"
saleinfo="true" wholesaleinfo="true"
prod_imgurl="www.hypermedic.com/mygif.gif">
  <description>
   Bata's best  black calf leather shoes for a formal occasion
  </description>
   <imageurl src="http://www.bata.com" height="300" width="800"
desc="Bata's best black formal calf leather"/>
  <saleinfo sale_price="49.95" sale_date1="10/1/2001"
sale_date2="12/25/2001" conditions="While supplies last" promoblurb="A
once in a lifetime opportunity" />
   <wholesaleinfo ws_price="480" ws_batch="12" />
</product>
</batch>
</catalog>
```

Note that when this is sent we will want the *DOCTYPE* declaration removed from the file, because we will have to map it from a virtual path to a relative path, and we can only do this on the server.

Figure 10.4 displays the DTD for the code shown in Figure 10.3. This code can be found on the Solutions Web site (www.syngress.com/solutions) for this book.

Figure 10.4 Catalog.dtd

```
<!ELEMENT catalog (supplier,batch+) >
<!ELEMENT supplier EMPTY >
<!ATTLIST supplier
  supp_name CDATA #IMPLIED
  uid CDATA #REQUIRED
  pw CDATA #REQUIRED

>
<!ELEMENT batch (product+) >
<!ATTLIST batch
  batch_type (cat_new|cat_update|cat_delete) "cat_update"

>

<!ELEMENT product (description,saleinfo?,wholesaleinfo?) >
<!ATTLIST product
  prod_code CDATA #REQUIRED
  prod_type CDATA #REQUIRED
  prod_name CDATA #IMPLIED
  prod_imgurl CDATA #REQUIRED
  prod_class CDATA #REQUIRED
  prod_price CDATA #REQUIRED
  saleinfo CDATA #REQUIRED
  wholesaleinfo CDATA #REQUIRED
  >

<!ELEMENT description (#PCDATA)* >
```

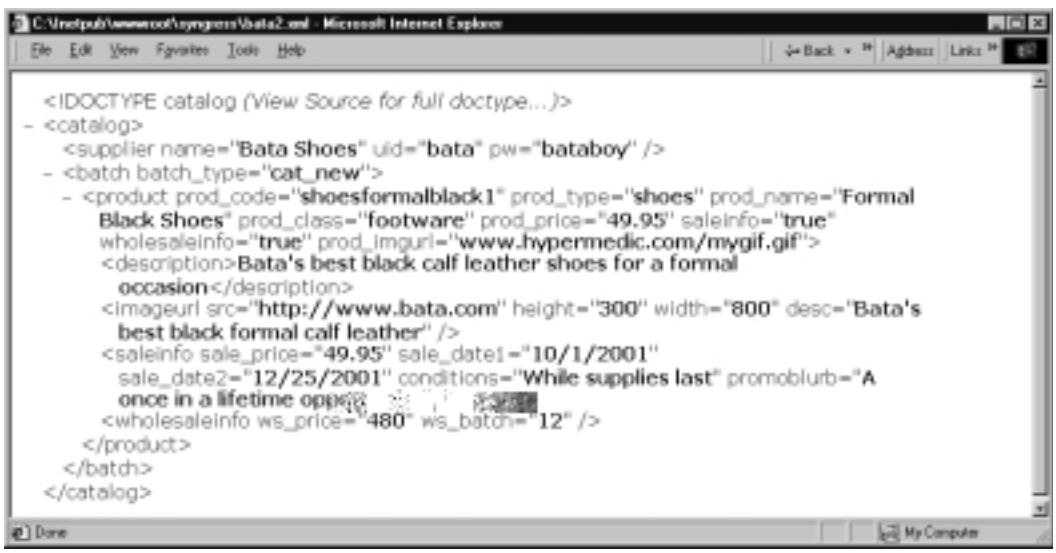
Continued

Figure 10.4 Continued

```
<!ELEMENT saleinfo EMPTY >
<!ATTLIST saleinfo
  sale_price CDATA #IMPLIED
  sale_date1 CDATA #IMPLIED
  sale_date2 CDATA #IMPLIED
  conditions CDATA #IMPLIED
  promoblurb CDATA #IMPLIED
>

<!ELEMENT wholesaleinfo EMPTY >
<!ATTLIST wholesaleinfo
  ws_price CDATA #IMPLIED
  ws_batch CDATA #IMPLIED
>
```

Figure 10.5 shows the same XML file displayed in Figure 10.3 (bata2.xml) as it appears when it is opened in Internet Explorer 5 (IE5).

Figure 10.5 bata2.xml As It Appears When Opened in IE5



NOTE

As we work through the code for our application, you will notice that we will not be working with code for processing the "saleinfo" and "wholesaleinfo" elements, nor will we be adding this information to the catalog. Adding such code would only serve to double or triple the size of the code files without demonstrating any new methods or techniques. Be aware, though, that these categories are an essential part of a "real-world" catalog.

Supplier Interface and B2B Design

Now that we have the XML package, we need to send it to the server, and process it so that the information is placed in the database. Suppliers come in all shapes and sizes—both large corporations with fully integrated e-business services, and smaller suppliers who are just beginning to get into e-business—and we need to cater to both of these classes. Some suppliers will be manually constructing an XML file, while larger and more sophisticated suppliers will be sending entire batches of information to insert new entries in the catalog, modify others, and delete old entries.

As we discussed earlier, though, all the suppliers will want to know whether the material they sent was received, what material we did receive, and were we able to process it. If we were not able to process it, they will want an error message containing at a minimum the reason we could not process it, and the place in the file where the error occurred. As we will see, ASP.NET makes this quite easy for us with the Try/Catch method of error trapping.

Fatal Errors versus Nonfatal Errors

In XML, if a fatal error occurs, the parser must cease processing the file in a normal manner. With HTML, on the other hand, the browser will do the best it can. When we receive an XML package, it might contain two kinds of errors that we can also call "fatal" and "non-fatal." The former would consist of a nonvalid or unformed XML files, or incorrect password information. In this instance, all processing should cease. A non-fatal error might consist of attempting to update a file that is not in the database, or inserting a duplicate copy of a product. In this

case, it would be possible to return a "partial error" and process the other correct items in a batch.

In some instances, in a closed environment, this might be acceptable, but when we are dealing with other unknown systems from unknown institutions, then we cannot know their capabilities. It is thus preferable to either process all the information sent to us, or none of it. The sending supplier can then fix the error in the batch and resend it.

NOTE

We will develop two almost identical files to process the XML package, one that will be viewable on a browser by a human viewer, and the other that will just return a well-formed XML file for machine consumption. The processing code is identical in both files. You will find two simple files on the companion Web site for this book—"testupdate.htm" and "testupdate2.htm"—that you can use to post packets to the "updatecat1.aspx" and "updatecat2.aspx" files.

Coding updatecat1.aspx

Although this file consists of 300 + lines of code, it is really quite straightforward. The code can be divided into the following sections:

- Importing the namespaces.
- Accepting the XML package.
- Validating the XML package.
- Checking the supplier's UID (user identification) and passwordLoop through the batches using a transaction.
- For each batch, loop through the products.
- Read all the values out of the XML file.
- Check the values for correct type.
- Read the values into the database
- Send a success message.
- If an error occurs anywhere, abort the session and send an appropriate error message.

Figure 10.6 Continued

```
    Dim xAttList as XmlNamedNodeMap
    Dim oNode as XmlNode

'data variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataReader as OleDbDataReader
    Dim strSQL as String
    Dim strSQL2 as String
    Dim strConn as String
    Dim oTransaction as OleDBTransaction

  Dim prod_code as string
  Dim prod_type as string
  Dim prod_name as string
  Dim prod_desc as string
  Dim prod_imgurl as string
  Dim prod_class as string
  Dim prod_price as string
  Dim prod_saleinfo as string
  Dim prod_wholesaleinfo as string

  Dim batch_number as integer
  Dim product_number as integer

''''''''''''VALIDATING THE XML PACKAGE''''''''''''''''''

' Add a doctype declaration with link to catalog.dtd
 Dim sDtdPath as string
   'mapPath is now part of request class
    sDTDPath=request.mapPath("catalog.dtd")
    xmlfile="<!DOCTYPE catalog SYSTEM """ & sDTDPath & """>" & nl & xmlfile
```

Continued

Figure 10.6 Continued

```
 Try
   oXML.loadXML(xmlfile)

  Catch oError as exception
    outError.innerHTML="<b>* Error while Loading XML document</b>.<br />" _
           & oError.Message & "<br />" & oError.Source
    exit sub
  End Try

'get uid and pw from'supplier' elements and read into variables
  Dim uid as string
  Dim pw as string

 xAttList=oXML.documentElement.firstChild.Attributes

Try

 uid=xAttList.GetNamedItem("uid").value
 pw=xAttList.GetNamedItem("pw").value


 Catch oError as exception
    outError.innerHTML="<b>* Could not find either 'pw' or 'uid' attribute_
      vallues on the passed XML file</b>.<br />" _
         & oError.Message & "<br />" & oError.Source
    exit sub
  End Try


''''''''''''''''''CHECKING UID AND PW''''''''''''''''''''''''''''''


'check the  the uid and password in the data base
```

Continued

Figure 10.6 Continued

```
    strSQL="SELECT *  FROM supplier WHERE supp_uid='" & uid &"' AND _
      supp_pw='" & pw & "';"

    strConn="Provider=Microsoft.Jet.OLEDB.4.0"
    strConn=strConn & "; Data Source= _
      c:\inetpub\wwwroot\syngress\dbases\catalog.mdb"

 try
    oConn=New OleDBConnection(strConn)
    oConn.open

    oComm=New OleDBCommand(strSQL,oConn)

    oDataReader=oComm.ExecuteReader()

    if oDataReader.Read()then
    else
      outError.innerHTML="<b>* Password Error in passed XML file</b>.<br />"
      oConn.close()
      exit sub
    end if
 'you may also want to check for case here
   if pw <> oDataReader("supp_pw") then
      outError.innerHTML="<b>* Password Error in passed XML file</b>.<br />"
      oConn.close()
      exit sub
    end if

 Catch oError as exception
    outError.innerHTML="<b>* There were errors accessing the _
      database</b>.<br />" _
         & oError.Message & "<br />" & oError.Source
```

Continued

Figure 10.6 Continued

```
    oConn.close()
    exit sub
  End Try
  oDataReader.close()
  oConn.close()

''''''''''''''''LOOP THROUGH BATCHES''''''''''''''''''''''''''



 'run through the passed batches
 Dim i as integer
 Dim j as integer
 Dim batch_type as string
  xBatchList=oXML.getElementsByTagName("batch")

  for i=0 to xBatchList.count-1
   batch_number=batch_number+1
   batch_type=xBatchList.item(i).Attributes.item(0).value
   'xProdList=xBatchList.item(i).childNodes
   batchEl=xBatchList.item(i)
   xProdList=batchEl.getElementsByTagName("product")
try

   oConn.Open()
   oTransaction=oConn.BeginTransaction()
   oComm.Connection=oConn
   oComm.CommandType=CommandType.Text
   oComm.Transaction=oTransaction
   dim oComm2 as New OleDBCommand



''''''''''''''''LOOP THROUGN PRODUCTS''''''''''''''''''''''''''
```

Continued

Figure 10.6 Continued

```
     for j=0 to xProdList.count-1
          product_number=product_number+1
          xAttList=xProdList.item(j).Attributes
'Nested Try

       try

''''''''''''''''''READ XML''''''''''''''''''''''''''''


          prod_code=xAttList.GetNamedItem("prod_code").value
          prod_code=uid & ":" & prod_code


''''''''''''''''''CHECK PROD_CODE''''''''''''''''''''''''''''


   'check to see whether this product is in the DB
             strSQL2="SELECT prod_code from products WHERE prod_code='" _
               & prod_code & "';"
             oComm2.Connection=oConn
             oComm2=New OleDBCommand(strSQL2,oConn)
             oComm2.Transaction=oTransaction
             oDataReader=oComm2.ExecuteReader()

          if batch_type="cat_new" then
             if oDataReader.Read()  then
                 outError.innerHTML="batch number-" & batch_number & " _
                   Product Number- " & product_number &"<b>*Insertion:" _
                   & prod_code & ": There is already a product with _
                   this code in the database</b>.<br />"

                 oTransaction.Rollback()
                 oDataReader.close()
                 oConn.close()
```

Continued

Figure 10.6 Continued

```
                exit sub
            else
                oDataReader.close()
            end if

        elseif  batch_type="cat_delete" then
            if oDataReader.Read() =false then
                outError.innerHTML="batch number-" & batch_number & " _
                  Product Number- " & product_number &"<b>*Deletion:" _
                  & prod_code & ": There is no product with this code _
                  in the database</b>.<br />"

                oTransaction.Rollback()
                oDataReader.close()
                oConn.close()
                exit sub
            else
                oDataReader.close()
            end if

        elseif  batch_type="cat_update" then
            if oDataReader.Read() =false then
                outError.innerHTML="batch number-" & batch_number & " _
                  Product Number- " & product_number &"<b>*Update:" & _
                  prod_code & ": There is no product with this code _
                  in the database</b>.<br />"

                oTransaction.Rollback()
                oDataReader.close()
                oConn.close()
                exit sub
            else
                oDataReader.close()
            end if
```

Continued

Figure 10.6 Continued

```
            else
               outError.innerHTML="batch number-" & batch_number & " _
                 Product Number- " & product_number &"<b>*The Batch _
                 type" & batch_type & ": was not recognized</b>.<br />"

                 oTransaction.Rollback()
                 oDataReader.close()
                 oConn.close()
                 exit sub

            end if


''''''''''''''''''CONTINUE READ XML''''''''''''''''''''''''''''''


          prod_type=xAttList.GetNamedItem("prod_type").value
          prod_name=xAttList.GetNamedItem("prod_name").value
          prod_imgurl=xAttList.GetNamedItem("prod_imgurl").value
          prod_class=xAttList.GetNamedItem("prod_class").value
          prod_price=xAttList.GetNamedItem("prod_price").value
          prod_saleinfo=xAttList.GetNamedItem("saleinfo").value
          prod_wholesaleinfo=xAttList.GetNamedItem("wholesaleinfo").value

          prod_desc=xProdList.item(j).firstChild.firstChild.value
 'write different sqlStrings dependung on value ofbatch_type

  ''''''''''''''''''CHECK TYPES''''''''''''''''''''''''''''''

  'We are relying on the DB to check the type.
'The Try catch will allow us to capture the OleDB errors


''''''''''''''''''BUILD SQL''''''''''''''''''''''''''''''
```

Continued

Figure 10.6 Continued

```
if batch_type="cat_new" then
    strSQL="INSERT INTO products  _
      (prod_code,prod_name,prod_desc,prod_price,prod_class,prod_type,_
      prod_introdate,prod_saleinfo,prod_wholesaleinfo,prod_imgurl,_
      supp_uid) VALUES ("
    strSQL+= "'" & prod_code & "',"
    strSQL+= "'" & replace(prod_name,"'","''") & "',"
    strSQL+= "'" & replace(prod_desc,"'","''") & "',"
    strSQL+= "" & prod_price & ","
    strSQL+= "'" & prod_class & "',"
    strSQL+= "'" & prod_type & "',"
    strSQL+= "'" & now & "',"
    strSQL+= "" & prod_saleinfo & ","
    strSQL+= "" & prod_wholesaleinfo & ","
    strSQL+= "'" & prod_imgurl & "',"
    strSQL+= "'" & uid & "')"


  elseif batch_type="cat_update" then

    strSQL="UPDATE products SET "
     strSQL+=" prod_code='" & prod_code & "',"
     strSQL+=" prod_type='" & prod_type & "',"
     strSQL+=" prod_name='" & replace(prod_name,"'","''") & "',"
     strSQL+=" prod_class='" & prod_class & "',"
     strSQL+=" prod_price='" & prod_price & "',"
     strSQL+=" prod_desc='" & replace(prod_desc,"'","''") & "',"
     strSQL+=" prod_saleinfo=" & prod_saleinfo & ","
     strSQL+=" prod_lastupdate='" & now & "',"
     strSQL+=" prod_wholesaleinfo=" & prod_saleinfo & ","
     strSQL+=" prod_imgurl='" & prod_imgurl & "',"
     strSQL+=" supp_uid='" & uid & "' "
     strSQL+=" WHERE prod_code='" & prod_code & "';"
```

Continued

Figure 10.6 Continued

```
      elseif batch_type="cat_delete" then
         strSQL="DELETE * FROM products WHERE prod_code='" & prod_code &
"';"
      else
        outError.innerHTML="batch number-" & batch_number & " Product _
          Number- " & product_number &"<b>* The batch type was not _
          recognised.</b>.<br />" _

        oConn.close()
        exit sub
      end if



''''''''''''''''READ VALS TO DB'''''''''''''''''''''''''


     oComm.CommandText=strSQL
     oComm.ExecuteNonQuery()



       Catch oError as exception
         outError.innerHTML="batch number-" & batch_number & " Product _
          Number- " & product_number &"<b>* Required Attribute Values _
          were missing</b>.<br />" _
         & oError.Message & "<br />" & oError.Source
         oConn.close()
         exit sub
      End Try
     Next  'end product loop

   oTransaction.Commit()
   oConn.close()

  Catch oError as Exception
```

Continued

Figure 10.6 Continued

```
    outError.innerHTML="batch number-" & batch_number & " Product Number-_
       " & product_number &"<b>* Error while accessing document</b>.<br _
       />"' _
          '& oError.Message & "<br />" & oError.Source

   oTransaction.Rollback()
   oDataReader.close()
   oConn.close()
   Exit Sub
   End Try

  Next   'end product loop


''''''''''''''''SUCCESS MESSAGE''''''''''''''''''''''''''




 success.innerHTML="<b>*" & batch_type & ":The update was successful</b>.
The following file was uploaded.<br />"
 xmlfile=replace(xmlfile,"<","&lt;")

 outxmlfile.innerHTML=xmlfile
 End Sub
</script>

''''''''''''''''PRINT PAGE''''''''''''''''''''''''''


  <netforce:header5 runat="server" />
<div id="outError" runat="server"></div>
<div id="success" runat="server"></div>
<pre id="outxmlfile" runat="server"></pre>
```

Analysis of Code Listing updatecat1.aspx

This is the heart of our product, and many new concepts are introduced, so we will spend a little time analyzing this code. The method we will use will be to look at any new concepts or programming tasks, comparing them if appropriate with similar ASP tasks, and then we will look at any salient code points.

NOTE

The ASP version of this code can be found on the Solutions Web site for this book as "processxml1.asp." However, although the output is similar, you can see that the .NET code is much cleaner! When we benchmarked the code, the .NET code ran about three times as fast as the ASP code!

Import the Namespaces

We will be working with a database here, so we will need to import the classes that handle that.

```
<%@ Import Namespace="System.Data"%>
<%@ Import Namespace="System.Data.OleDb"%>
<%@ Import Namespace="System.XML"%>
```

Since we are prototyping in Access, we have imported the OleDb classes. When we migrate to SQL Server, we will have to change these namespaces. (See *Installation, Migrating to SQL Server* later in this chapter.)

Accepting the XML Package

This will be familiar to ASP coders.

```
xmlfile=request.params("xmlfile")
```

We could have used *request.queryString()* or *request.form()* methods to detail the HTTP method, but the params.method makes things much easier. Like many Internet developers, we tend to develop and debug with the HTTP get method (we can see what is being passed to the server), and then we convert to the post method.

Validating the XML Package

Here we have prefixed the file with a reference to our schema (we are using DTDs, but if you would prefer to use XML schemas, .NET has full support for

them). Note that the *mapPath()* method is now part of the Request object. When dealing with a B2B application, we sometimes do not have the ability to choose what we will be using as our validation language; in this case, it's DTD.

```
    sDTDPath=request.mapPath("catalog.dtd")
    xmlfile="<!DOCTYPE catalog SYSTEM """ & sDTDPath & """>" & nl_ &_
xmlfile
```

We now load the object (by default, load will validate the file).

```
Try
   XML.loadXML(xmlfile)

Catch Error as exception
 outError.innerHTML="<b>* Error while Loading XML document</b>.<br />"_
          & Error.Message & "<br />" & oError.Source
 exit sub
 End Try
```

Unlike ASP, we no longer have to specifically trap any errors, as our Try/Catch statement will catch them for us. Note that if there is an error, we pass it to our error-catching element and exit the sub. This is the method that we will use from this point forward.

Checking the Supplier's User Identification and Password

In ADO.NET, opening a database and reading from it is very similar to ASP, it's just that the syntax might look a little strange.

1. We create a Connection object and open it.
2. We create a Command object, and execute it with a SQL statement and a Connection object.
3. We create a DataReader for the Command object and execute it.
4. We read out the data from the reader.

This will be familiar to ASP programmers, except that ASP allowed us to bypass the Command stage (although a Command object was always implicitly created in the background), and the data was contained in a RecordSet.

The data variables must be declared and typed before implementation. (It is possible to load them when they are declared, but we have not done this here for reasons of clarity.)

```
'data variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataReader as OleDbDataReader
```

The following code is quite straightforward; however, the Read() method requires a little explanation.

```
oConn=New OleDBConnection(strConn)
oConn.open

oComm=New OleDBCommand(strSQL,oConn)

oDataReader=oComm.ExecuteReader()

if oDataReader.Read()then
```

In ASP, we would have to specifically loop through a recordset, testing each time to see whether we had reached the End of the File. With the *Read()* method of *DataReader*, these details are all taken care of. *Read()* returns a Boolean—true if there is material in the *DataReader*; otherwise, false.

In our case here, we are trying see whether there is a matching *UserID* and Password in the database. If there is, true will be returned. If false is returned, we send an error message and exit the subroutine.

Developing & Deploying…

DataReader versus DataSet

DataReader for the most part works pretty much like the old recordset with which ASP programmers are familiar. *DataSet* will create a virtual database (preserved in XML) that we can work with even while disconnected to the database. It requires a complete new subset of objects and methods to work with it. (We will use a small part of it to create an XML file for our Web service later on.)

In ASP it was common to open the connection and keep it open throughout the application. In .NET, we are told that it is preferable to close the connection each time we are finished using it. The *DataReader* must also be closed, or we will get an error next time we try to use it.

```
oDataReader.close()
oConn.close()
```

Loop through the Batches

The XML file that was passed to can contain several batch elements, each with several product elements. We need to loop through these and find out what type of batch update we are dealing with, and then loop through the product elements, writing their values to our database.

```
xBatchList=oXML.getElementsByTagName("batch")


for i=0 to xBatchList.count-1
 batch_type=xBatchList.item(i).Attributes.item(0).value
 'xProdList=xBatchList.item(i).childNodes
 batchEl=xBatchList.item(i)
 xProdList=batchEl.getElementsByTagName("product")
```

The simplest way to make a list of the batch elements is to use the *getElementsByTagName()* method of the *XmlDocument* Class. Note also that the *XmlNodeList* class uses the *count* property. Using *length* as we did in ASP will throw an exception.

For every batch, we need to make a list of the product elements. Each item in a node list is, of course, a node, and *getElementsByTagName()* is not a recognized method of the *XmlNode* class. Therefore, in order to use it, we have to specifically "cast" each node to be a member of the *XmlElement* class. An alternative method would be to use the *childNodes* property, but in that case, all the "saleinfo" and "wholesaleinfo" elements would be included.

Also at this stage, we find out what type of batch we are processing.

```
batch_type=xBatchList.item(i).Attributes.item(0).value
```

A "*cat_new*" value will mean we are processing a batch that is delivering new entries, "*cat_update*" means that an existing entry has to be revised, and a "*cat_delete*" value means of course that an existing entry should be deleted. We

will have to generate different error strings and SQL strings depending on the batch type.

Beginning a Transaction

As we discussed previously, we either want the whole of the batches passed in the XML file to be entered in the database, or we want the whole thing to fail and we want to return an error. ASP.NET has integral support for transactions that makes this easy. We create a Transaction object and pass it to the Connection object using the *BeginTransaction()* method. We must also pass it to every Command object that we are going to use.

```
oConn.Open()
   oTransaction=oConn.BeginTransaction()
   oComm.Connection=oConn
   oComm.CommandType=CommandType.Text
   oComm.Transaction=oTransaction
   dim oComm2 as New OleDBCommand
```

For Each Batch Loop through the Products

Now, start looping through the product elements in each batch and make a list of attributes.

```
for j=0 to xProdList.count-1
      xAttList=xProdList.item(j).Attributes
```

Check the Status of the Prod_Code

We already have the type of batch read into a variable; now we want to find out what the *prod_code* value is. Remember that this is a unique value that identifies a product in our database. It consists of a compound value, the first part being the unique UID of the supplier; and the second part is a unique description of the product provided by the supplier.

```
prod_code=xAttList.GetNamedItem("prod_code").value
prod_code=uid & ":" & prod_code
```

Once we have the prod_code, we need to see if it is already present in the database. If it *is*, *and* the batch type is "*cat_new*", then we must generate an error. If it *is not* present, and the batch type is "*cat_update*" or "*cat_delete*", then, again, we must generate an error and terminate the transaction.

```
'check to see whether this product is in the DB
              strSQL2="SELECT prod_code from products WHERE prod_code='"_
                & prod_code & "';"
              oComm2.Connection=oConn
              oComm2=New OleDBCommand(strSQL2,oConn)
              oComm2.Transaction=oTransaction
              oDataReader=oComm2.ExecuteReader()
```

Note how we must use a second Command object to do this, because the first Command object is being used in the outer loop. We must also assign the Transaction object to this Command object.

See how we generate the error message. Earlier on we had declared two integer variables to represent the batch and the product loops, and we advanced them with each loop. We use these to provide a location in our error message.

```
          if batch_type="cat_new" then
             if oDataReader.Read()  then
                 outError.innerHTML="batch number-" & batch_number & _
                   " Product Number- " & product_number &_
                   "<b>*Insertion:" & prod_code & ": There is already _
                   a product with this code in the database</b>.<br />"

                 oTransaction.Rollback()
                 oDataReader.close()
                 oConn.close()
                 exit sub
```

If there is an error, then we also must roll back the transaction, close both our DataReader and Connection objects, and exit the subroutine.

Read All the Values Out of the XML File

If there is not an error, then we read the rest of our product values and assign them to variables.

```
          prod_type=xAttList.GetNamedItem("prod_type").value
          prod_name=xAttList.GetNamedItem("prod_name").value
          prod_imgurl=xAttList.GetNamedItem("prod_imgurl").value
          prod_class=xAttList.GetNamedItem("prod_class").value
```

```
prod_price=xAttList.GetNamedItem("prod_price").value
prod_saleinfo=xAttList.GetNamedItem("saleinfo").value
prod_wholesaleinfo=xAttList.GetNamedItem("wholesaleinfo").value

prod_desc=xProdList.item(j).firstChild.firstChild.value
```

Check the Values for Correct Type

We are cheating a little here. In ASP, we should check all the values using code, to make sure they are the correct type. However, in ASP.NET it is easy to let the database check the type, and use Try/Catch to return any errors. If we wanted to check an e-mail or a URL format, or indeed if we wanted to write code to check other values, this is where we would insert it.

Build SQL Strings

We now build different SQL strings depending on the batch type. The delete string is the least verbose!

```
elseif batch_type="cat_delete" then
   strSQL="DELETE * FROM products WHERE prod_code='" & prod_code _
     & "';"
```

WARNING

Make sure that you have qualified all columns you use in your SQL string, or you could end up deleting your entire database!

Read the Values into the Database

Once we have the correct SQL string, it is an easy matter to read it into the DB!

```
oComm.CommandText=strSQL
oComm.ExecuteNonQuery()
```

Send a Success Message

If we complete both loops we know there have been no errors, and we can send a success message!

```
success.innerHTML="<b>*" & batch_type & ":The update was _
  successful</b>. The following file was uploaded.<br />"
 xmlfile=replace(xmlfile,"<","&lt;")
```

Print Page

updatecat1.aspx will return a Web page suitable for human consumption. updatecat2.aspx will return an XML file suitable for machine processing. Of course, the suppliers can do what ever they want with the file. We have done our job, by telling them that we have either processed the file or not, as the case may be, and by supplying them with useful information that they can act on.

Figure 10.7 shows the result of a successful upload if updatecat1.aspx (Figure 10.6) is called.

Figure 10.7 Successful Upload

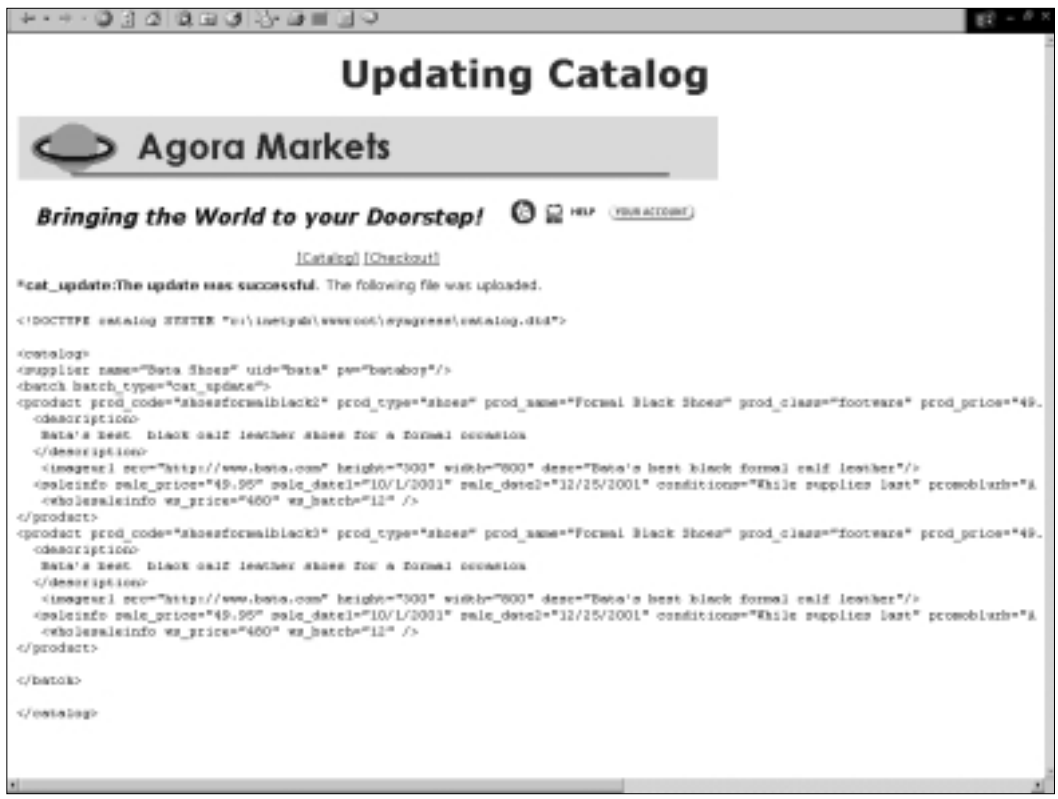
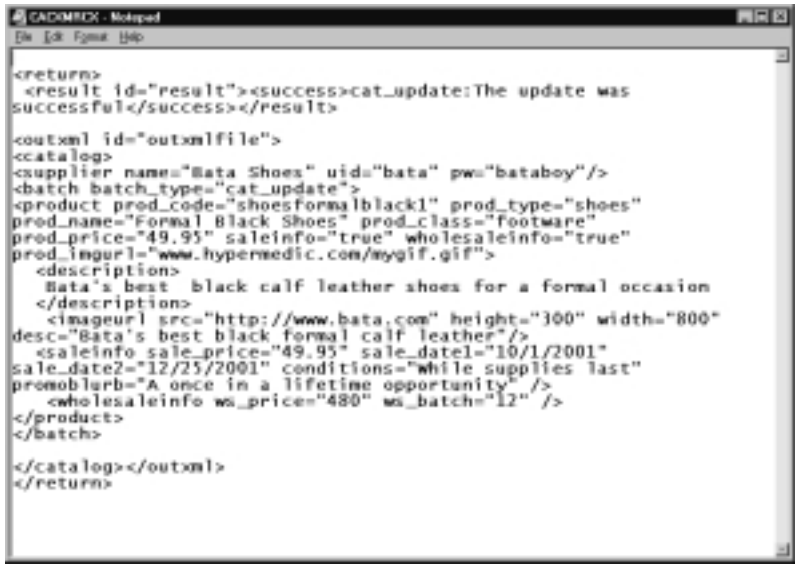


Figure 10.8 File Sent to Supplier



```
<return>
 <result id="result"><success>cat_update:The update was
successful</success></result>

<outxml id="outxmlfile">
<catalog>
<supplier name="Bata Shoes" uid="bata" pw="bataboy"/>
<batch batch_type="cat_update">
<product prod_code="shoesformalblack1" prod_type="shoes"
prod_name="Formal Black Shoes" prod_class="footware"
prod_price="49.95" saleinfo="true" wholesaleinfo="true"
prod_imgurl="www.hypermedic.com/mygif.gif">
  <description>
   Bata's best  black calf leather shoes for a formal occasion
  </description>
   <imageurl src="http://www.bata.com" height="300" width="800"
desc="Bata's best black formal calf leather"/>
  <saleinfo sale_price="49.95" sale_date1="10/1/2001"
sale_date2="12/25/2001" conditions="while supplies last"
promoblurb="A once in a lifetime opportunity" />
   <wholesaleinfo ws_price="480" ws_batch="12" />
</product>
</batch>

</catalog></outxml>
</return>
```

Figure 10.8 shows the XML file that is send back to the supplier if updatecat2.aspx is called.

Customer Interface Design

Now that we have our catalog in a database, and have provided a means for suppliers to add items to the catalog, we need to display the products to our customers and other businesses. We will do the former by designing a traditional Web page interface, where our customers can select items from the catalog, put them in a shopping cart, and finally check them out of the shopping cart.

This interface will consist of three sections:

- A page that accesses the catalogs content
- Page(s) that show the contents of the shopping cart
- Checkout page(s)

GUI: The Catalog Page

This page allows browsers of the Web to access our catalog. As with any modern business Web page, the page should use readily understandable navigation and layout paradigms, and its use should be intuitive. We will want:

- A header containing announcements and a navigation strip.
- An area in which we can search the contents of the catalog (in this case, we have opted to do this via a left-hand column).
- An area in which the results of our search are displayed.
- An ability to load displayed articles into a shopping cart.

Figure 10.9 shows such a page as it might be displayed in a browser. Note that **Clothing** has been selected, which has produced a drop-down list of subheadings, and **Apparel/male** has been selected from the subheadings. The page also offers options to search by a selection string, and by vendor.

Figure 10.9 Our Catalog GUI

This page is certainly a serviceable page, but in a real world catalog, we would probably introduce several enhancements; namely:

- In this page, all the items in a selected group are displayed at one time.
- An image accompanies every item.

- In a **real-world** page we would probably introduce a means to scroll through the selected group with a **backward** and **forward** bar.
- We would probably introduce some mechanism whereby the catalog image would only be displayed on command.

However, this page should act as a good starting point for your own catalog.

The code that produced this page is surprisingly terse, thanks to the use of user controls, and can be seen in Figure 10.10. The complete source code for Figure 10.10 can be found on the companion Solutions Web site for the book (www.syngress.com/solutions).

SYNGRESS
syngress.com

Figure 10.10 Code Listing for Catpage1.aspx

```
<%@ Register tagprefix="netforce" tagname="vendors" _
  src="usercontrols/vendors.ascx" %>
<%@ Register tagprefix="netforce" tagname="header2" _
  src="usercontrols/header2.ascx" %>
<%@ Register tagprefix="netforce" tagname="header1" _
  src="usercontrols/header1.ascx" %>
<%@ Register tagprefix="netforce" tagname="classes" _
  src="usercontrols/classes.ascx" %>
<%@ Register tagprefix="netforce" tagname="footer1" _
  src="usercontrols/footer1.ascx" %>
<%@ Register tagprefix="netforce" tagname="body1" _
  src="usercontrols/body1.ascx" %>
<html>
 <head>
  <title>Agora Markets Catalog</title>
 <link rel="stylesheet" type="text/css" href="genstyle.css">
 </head>
 <body>
   <netforce:header1 runat="server" />
   <table  border='1'>
  <tr>
   <!—Navigation Column—>
   <td rowspan='2' valign='top'>
   <h3>Search</h3>
```

Continued

Figure 10.10 Continued

```
    <form action='catpage1.aspx' method='post'>
    <input type='text' name='searchval' />
   <br />
   <input type='submit' Value='Search' />

   </form>

   <h3>Vendors</h3>
   <form action='catpage1.aspx' method='post'>
   <netforce:vendors runat="server" />
   <br />
   <input type='submit' Value='Fetch' />
   </form>

   <h3>Categories</h3>
   <netforce:classes runat="server" />
   </td>
   <td><netforce:header2 runat="server" />   </td>
   </tr>

   <tr><td valign='top'><netforce:body1 runat="server" /></td></tr>
  </table>
  <netforce:footer1 runat="server" />

 </body>
</html>
```

Analysis of Code

This book is not a primer on XHTML, so for the most part we will only analyze the areas that are pertinent to ASP.NET. However, note the use of *UserControls* to compartmentalize the code. *UserControls* not only make it easier to reuse code, but they make it easier to code, in that we can work on just one area of our code

at a time. This makes coding in ASP.NET akin to coding in an object-orientated language.

The code that produces the interface is standard XHTML, and the drop-down menus are produced using Style properties and Java script. All the code is reproduced on the companion Solutions Web Site (www.syngress.com/solutions) and is well commented, therefore it will not be explained here, as it is assumed the reader has more than a passing acquaintance with Web page design.

The part of the code that does the business is contained in body1.ascx and the listing for this *UserControl* is shown in Figure 10.11 (both these files can be found on the Solutions Web site for the book). Essentially, it takes the values that are passed to it, looks in the database for matches, and then uses a loop to write out a table of all the items that match the search items.

SYNGRESS
syngress.com

Figure 10.11 Listing for body1.ascx

```
<%@ Import Namespace="System.Data"%>
<%@ Import Namespace="System.Data.OleDb"%>

<script language="VB" runat="server">

 Sub Page_Load(Source As Object,E as EventArgs)

'declare variables
    Dim catClass as string
    Dim catType as string
    Dim vendors as string
    Dim strResult as string
    Dim searchval as string
    Dim ordXML as string
    Dim counter as integer
'database variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataReader as OleDbDataReader
    Dim strSQL as String
    Dim strConn as String
'misc variables
```

Continued

Figure 10.11 Continued

```
    Dim imgpath as string

'makesure the correct namespace prefix is used from the array of user _
  controls.
  vendors=request.params("ctrl1:vendors")
  searchval=request.params("searchval")
  catClass=request.params("class")
  catType=request.params("type")

    strConn="Provider=Microsoft.Jet.OLEDB.4.0"
    strConn=strConn & "; Data Source=
c:\inetpub\wwwroot\syngress\dbases\catalog.mdb"

    oConn=New OleDBConnection(strConn)
    oConn.open

 if catClass="" AND vendors="" AND searchval="" then
  ' response.write ("<span style='font-weight:bold;font-size:14pt;_
    '>Thankyou for shopping at Agora markets. In order to see our _
    products, please select either a category or a vendor</span>")
   strSQL="SELECT * FROM products WHERE prod_type='XXXXXXX';"

 else if searchval <> "" then
          strSQL="SELECT products.*,supplier.supp_name FROM products,_
            supplier WHERE products.supp_uid=supplier.supp_uid AND _
            products.prod_name LIKE '%" & searchval & "%' ORDER BY _
            prod_name;"
          response.write(strSQL)
          'exit sub

 else if vendors <> "" then
          strSQL="SELECT products.*,supplier.supp_name FROM
products,supplier WHERE products.supp_uid=supplier.supp_uid AND
supplier.supp_uid='" & vendors & "' ORDER BY prod_name;"
```

Continued

Figure 10.11 Continued

```
 else if catType<>"" then
   strSQL="SELECT products.*,supplier.supp_name FROM products,supplier _
     WHERE products.supp_uid=supplier.supp_uid AND products.prod_type='" _
     & catType & "' ORDER BY prod_name;"

  strResult=catType
 else if catClass<>"" then
  strSQL="SELECT products.*,supplier.supp_name FROM products,supplier _
    WHERE products.supp_uid=supplier.supp_uid AND products.prod_class='"_
     & catClass & "' ORDER BY prod_name;"

  strResult=catClass
 end if
 'response.write(strSQL & "DDDDDDDD")
    oComm=New OleDBCommand(strSQL,oConn)

    oDataReader=oComm.ExecuteReader()
    'response.write (oDataReader)
    strResult="<table border='1' cellpadding='3' rules='rows'>"


strResult+="<tr><th>Vendor</th><th>Name</th><th>Description</th><th>_
  Picture</th><th>Price</th><th>Qty</th><th>Add to cart</th></tr>"

    Do While oDataReader.Read()
    counter=counter+1
    strResult+="<tr>"
    strResult+="<td>" & oDataReader("supp_name") & "</td>"
    strResult+="<td>" & oDataReader("prod_name") & "</td>"
    strResult+="<td>" & oDataReader("prod_desc") & "</td>"
     if oDataReader("prod_imgurl") <> "" then
       imgpath=oDataReader("prod_imgurl")
       strResult+="<td><img src='" & imgpath & "' alt='" &
oDataReader("prod_name") & "'/></td>"
     else
```

Continued

Figure 10.11 Continued

```
        strResult+="<td>[No Image Available]</td>"
     end if

    strResult+="<td>" & oDataReader("prod_price") & "</td>"
    strResult+="<form action='shopcartadd.aspx'><td>"
    strResult+="<input style='width:0.6cm;' type='text' value='1' _
      name='ord_qty' /></td>"
    strResult+="<input  type='hidden' name='supp_uid' value='" & _
      oDataReader("supp_uid") & "' />"
    strResult+="<input  type='hidden' name='supp_name' value="""" & _
      oDataReader("supp_name") & """ />"
    strResult+="<input  type='hidden' name='prod_name' value='" & _
      oDataReader("prod_name") & "' />"
    strResult+="<input  type='hidden' name='prod_code' value='" & _
      oDataReader("prod_code") & "' />"
    strResult+="<input  type='hidden' name='prod_price' value='" & _
      oDataReader("prod_price") & "' />"
    strResult+="<td><input style='width:1cm;' type='submit' value='Add' _
      /></td></form>"

    strResult+="</tr>"
    loop

   if counter=0 then
    strResult+="<tr><th colspan='7'>No Catalog entries match the _
      search criteria</th></tr>"
   end if

    strResult+="</table>"
    'response.write(counter)
    outResult.InnerHtml = strResult

 'housekeeping
     oDataReader.Close()
     oConn.Close()
```

Continued

Figure 10.11 Continued

```
   end sub
</script>
<h3 id="outResult" runat="server"></h3>
```

Analysis of Code

First, the namespaces are imported in a standard fashion, and the *Page_Load* subroutine is run.

```
<%@ Import Namespace="System.Data"%>
<%@ Import Namespace="System.Data.OleDb"%>

<script language="VB" runat="server">

 Sub Page_Load(Source As Object,E as EventArgs)
```

For readers familiar with ASP, the way the table is produced is a little different. Here are the *OleDb* objects that we will be using:

```
'database variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataReader as OleDbDataReader
```

The only part of the following code that needs explanation is the namespace parameter passed from the "*vendors*" user control. In fact in Vendors.ascx we used an asp.net control to produce our drop-down list.

```
  <asp:DropDownList id="vendors"
   DataTextField="supp_name" DataValueField="supp_uid"
  runat="server"/>
```

And because we did this, the value gets passed with the index number of the control attached to it.

```
'make sure the correct namespace prefix is used from the array of user
controls.
  vendors=request.params("ctrl1:vendors")
  searchval=request.params("searchval")
  catClass=request.params("class")
  catType=request.params("type")
```

```
strConn="Provider=Microsoft.Jet.OLEDB.4.0"
strConn=strConn & "; Data Source=
  c:\inetpub\wwwroot\syngress\dbases\catalog.mdb"
```

We create a Connection object and then write a suitable SQL statement depending on which of the methods of browsing the catalog is selected; for example, by search string, vendor name, or category.

```
oConn=New OleDBConnection(strConn)
oConn.open

if catClass="" AND vendors="" AND searchval="" then
 ' response.write ("<span style='font-weight:bold;font-size:14pt;_
   '>Thankyou for shopping at Agora markets. In order to see our _
   products, please select either a category or a vendor</span>")
  strSQL="SELECT * FROM products WHERE prod_type='XXXXXXX';"

else if searchval <> "" then
          strSQL="SELECT products.*,supplier.supp_name FROM _
            products,supplier WHERE products.supp_uid=supplier.supp_uid_
            AND products.prod_name LIKE '%" & searchval & "%' ORDER BY _
            prod_name;"
          response.write(strSQL)
          'exit sub

else if vendors <> "" then
          strSQL="SELECT products.*,supplier.supp_name FROM _
            products,supplier WHERE products.supp_uid=supplier.supp_uid _
            AND supplier.supp_uid='" & vendors & "' ORDER BY _
            prod_name;"


else if catType<>"" then
  strSQL="SELECT products.*,supplier.supp_name FROM products,supplier _
    WHERE products.supp_uid=supplier.supp_uid AND products.prod_type='"_
     & catType & "' ORDER BY prod_name;"

 strResult=catType
else if catClass<>"" then
```

```
  strSQL="SELECT products.*,supplier.supp_name FROM products,supplier _
    WHERE products.supp_uid=supplier.supp_uid AND products.prod_class='"_
    & catClass & "' ORDER BY prod_name;"


  strResult=catClass
 end if
 'response.write(strSQL & "DDDDDDDD")
```

We now create a Command object (oComm), passing it both our Connection object and our SQL string.

```
oComm=New OleDBCommand(strSQL,oConn)
```

In order to read this, we have to create a data reader by using the Command object's ExecuteReader() method.

```
oDataReader=oComm.ExecuteReader()
'response.write (oDataReader)
strResult="<table border='1' cellpadding='3' rules='rows'>"


strResult+="<tr><th>Vendor</th><th>Name</th><th>Description</th>_
  <th>Picture</th><th>Price</th><th>Qty</th><th>Add to cart</th>_
  </tr>"
```

Now we use the *Read()* method of our *DataReader* object to obtain the contents of that object. Note that this method is a Boolean, and will remain true until all of the content of our *DataReader* is read out. Unlike ASP, there is no need to use the .moveNext method to advance the reader. For those of us coming from ASP, this appears to be a little strange!

```
Do While oDataReader.Read()
counter=counter+1
strResult+="<tr>"
strResult+="<td>" & oDataReader("supp_name") & "</td>"
strResult+="<td>" & oDataReader("prod_name") & "</td>"
strResult+="<td>" & oDataReader("prod_desc") & "</td>"
 if oDataReader("prod_imgurl") <> "" then
   imgpath=oDataReader("prod_imgurl")
   strResult+="<td><img src='" & imgpath & "' alt='" & _
     oDataReader("prod_name") & "'/></td>"
 else
```

```
        strResult+="<td>[No Image Available]</td>"
      end if

      strResult+="<td>" & oDataReader("prod_price") & "</td>"
      strResult+="<form action='shopcartadd.aspx'><td>"
      strResult+="<input style='width:0.6cm;' type='text' value='1' _
        name='ord_qty' /></td>"
      strResult+="<input  type='hidden' name='supp_uid' value='" & _
        oDataReader("supp_uid") & "' />"
      strResult+="<input  type='hidden' name='supp_name' value="""" & _
        oDataReader("supp_name") & """ />"
      strResult+="<input  type='hidden' name='prod_name' value='" & _
        oDataReader("prod_name") & "' />"
      strResult+="<input  type='hidden' name='prod_code' value='" & _
        oDataReader("prod_code") & "' />"
      strResult+="<input  type='hidden' name='prod_price' value='" & _
        oDataReader("prod_price") & "' />"
      strResult+="<td><input style='width:1cm;' type='submit' value='Add' _
        /></td></form>"

      strResult+="</tr>"
      loop

    if counter=0 then
      strResult+="<tr><th colspan='7'>No Catalog entries match the search_
        criteria</th></tr>"
    end if

      strResult+="</table>"
      'response.write(counter)
```

We read out all the contents of the dataset into a string, and then publish the string using the innerHtml method.

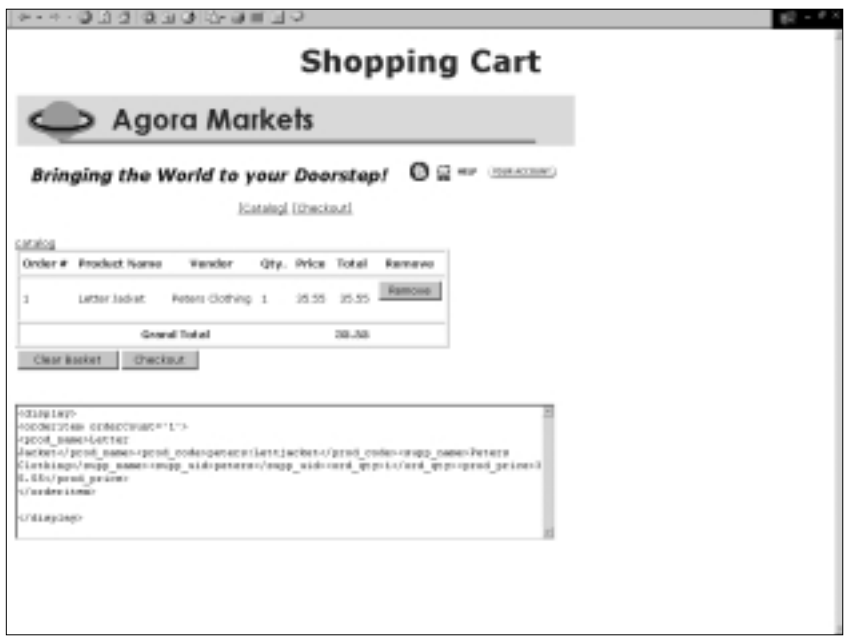
```
      outResult.InnerHtml = strResult

  'housekeeping
       oDataReader.Close()
```

```
        oConn.Close()

    end sub
</script>
<h3 id="outResult" runat="server"></h3>
```

GUI: The Shopping Cart Page(s)

When the **Add** button is clicked on a catalog page, the item is automatically added to the shopping cart. Figure 10.12 shows what will be seen if the **Letter Jacket** is selected.

Figure 10.12 Letter Jacket Selection



NOTE

For the purposes of this book, we have added a "textarea" that shows what is going on behind the screens. The textarea displays the XML file that represents the shopping cart. Of course, this would be removed in a real world application.

The shopping cart is represented by an XML file that is stored as a *Session* variable.

Developing & Deploying…

Session Objects in ASP.NET

In ASP, session variables could be unreliable in large-scale applications that employed a server farm. To overcome this, we had to write quite complex code to make sure that a session call was carried out on the same server on which the session variable was stored. In ASP.NET, we no longer have to worry about this.

The details of how ASP.NET accomplishes this are outside the scope for this book and are explained fully in the text of *The ASP.NET Web Developer's Guide* (ISBN: 1-928994-51-2), also from Syngress Publishing.

Note that if the user refreshes this page, the item will be added again to the shopping cart! In a real-life application, this problem can be overcome by writing all the code that adds the XML to the XML string on one page, and then redirecting to a display page. This has not been done here for reasons of clarity.

Figure 10.13 shows the listing for shopcartadd.aspx. The source code for shopcartadd.aspx can be found on the Solutions Web site for this book (www.syngress.com/solutions).

SYNGRESS syngress.com

Figure 10.13 Listing for shopcartadd.aspx

```
<%@ Register tagprefix="netforce" tagname="header3" _
  src="usercontrols/header3.ascx" %>

<%@ Import Namespace="System.XML"%>
<html>
<title>Shopping Cart Add</title>
 <link rel="stylesheet" type="text/css" href="genstyle.css">

<script language="VB" runat="server">

 Sub Page_Load(Source As Object,E as EventArgs)

 'declare variables
```

Continued

Figure 10.13 Continued

```
    Dim prod_code as string
    Dim prod_name as string
    Dim supp_uid as String
    Dim supp_name as string
    Dim ord_qty as string
    Dim prod_price as string
'
    Dim ordXML as string
    Dim cartXML as string
    Dim displayXML as string
    Dim nl as string
    Dim orderCount as integer
    Dim strResult as string
'declare table values
    Dim qty as string
    Dim iQty as integer
    Dim price as string
    Dim dPrice as double
    Dim total as string
    Dim dTotal as double
    Dim dGrandTotal as double

'xml variables
    Dim oXML as new XmlDocument
    Dim xList as XmlNodeList
    Dim oNode as XmlNode
    nl=chr(13) & chr(10)
'retrieve variables

     prod_code=request.params("prod_code")
     prod_name=request.params("prod_name")
     prod_price=request.params("prod_price")
     supp_uid=request.params("supp_uid")
```

Continued

Figure 10.13 Continued

```
    supp_name=request.params("supp_name")
    ord_qty=request.params("ord_qty")

    session("orderCount")=session("orderCount")+1
    orderCount=cStr(session("orderCount"))
'add the order

    ordXML="<prod_name>" & prod_name & "</prod_name><prod_code>" & _
      prod_code & "</prod_code><supp_name>" & supp_name & _
      "</supp_name><supp_uid>" & supp_uid & "</supp_uid><ord_qty>" & _
      ord_qty & "</ord_qty><prod_price>" & prod_price & "</prod_price>"
    session("cartXML")=session("cartXML") & "<orderitem orderCount='" _
      & orderCount & "'>" & nl & ordXML & nl & "</orderitem>" & nl
  'response.write (ordXML)

     displayXML="<display>" & nl  & session("cartXML") & nl &  _
       "</display>"



 Try
  oXML.loadXML(displayXML)
  'response.write("<b>*Document Loaded</b>")
 Catch oError as exception
   response.write("<b>* Error while accessing document</b>.<br />" _
         & oError.Message & "<br />" & oError.Source)
   exit sub
    End Try

  xList=oXML.documentElement.childNodes

  strResult="<table border='1' cellpadding='5' rules='rows'>"
  strResult+="<tr><th>Order #</th><th>Product Name</th><th>_
    Vendor</th><th>Qty.</th><th>Price</th><th>Total</th><th>Remove_
    </th></tr>"
```

Continued

Figure 10.13 Continued

```
For Each oNode In xList


            strResult+="<tr>"
            strResult+="<td>" & oNode.Attributes.item(0).Value & _
              "</td>" 'GetAttribute("ordercount")& "</td>" '
            strResult+="<td>" & oNode.FirstChild().FirstChild().Value &_
              "</td>"
            strResult+="<td>" & oNode.FirstChild().nextSibling_
              .nextSibling.FirstChild().Value & "</td>"
            strResult+="<td>" & oNode.FirstChild().nextSibling_
              .nextSibling.nextSibling.nextSibling.FirstChild().Value _
              & "</td>"
            strResult+="<td>" & oNode.FirstChild().nextSibling_
              .nextSibling.nextSibling.nextSibling.nextSibling_
              .FirstChild().Value & "</td>"
             iQty=cInt(oNode.FirstChild().nextSibling.nextSibling_
              .nextSibling.nextSibling.FirstChild().Value)
             dPrice=cDbl(oNode.FirstChild().nextSibling.nextSibling_
              .nextSibling.nextSibling.nextSibling.FirstChild().Value)
              if iQty < 0 then
                iQty=0
              end if
           strResult+="<td align='right'>" & iQty*dPrice & "</td>"
           dGrandTotal=dGrandTotal +0 + (iQty*dPrice)


           strResult+="<td><form action='shopcartremove.aspx'><input _
             type='hidden'  name='removeoc' value='" & _
             oNode.Attributes.item(0).Value & "' /><input type='hidden'_
             name='remove' value='remove' /><input type='submit' _
             value='Remove' /></form></td>"
           strResult+="</tr>"


  Next
```

Continued

Figure 10.13 Continued

```
    strResult+="<tr><th colspan='5'> Grand Total</th><th _
      align='right'>" & dGrandTotal & "</th><th> </th></tr>"

   strResult+="</table>"
   cartitems.innerHtml=strResult
         ta.innerHTML=displayXML



 end sub
</script>
  <netforce:header3 runat="server" />
<br /><a href="catpage3.aspx">catalog</a>
<div id="cartitems" runat="server"></div>

<table>
<tr><td>
<form action="shopcartremove.aspx">
<input type="hidden" name="clearall" value="clearall"/>
<input type="submit" Value="Clear Basket" />
</form>
</td>
<td><form action="checkout.aspx">
<input type="hidden" name="checkout" value="checkout"/>
<input type="submit" Value="Checkout" />
</form>
</td></tr>
</table>
<form>
<textarea id="ta" rows="10" cols="80" runat="server"></textarea>
</form>

</html>
```

Analysis of Code Listing 'shopcartadd.aspx'

Shopcartadd.aspx takes the variables passed to it from catpage1.aspx and builds an XML fragment that is added to the XML fragment that is already stored in the session variable. This is converted to a well-formed XML document by wrapping it in a root element, it is loaded into an XML object, and this object is then used to print out our display. Only the parts of the code pertinent to creating the XML object are analyzed here.

The namespace is imported:

```
<%@ Import Namespace="System.XML"%>
```

Here are the XML variables from this namespace that we will be using:

```
'xml variables
    Dim oXML as new XmlDocument
    Dim xList as XmlNodeList
    Dim oNode as XmlNode
```

We need to be able to keep track of all the orders. This will be important if we want to remove an order at a later date. We do this simply by using a session *orderCount* variable, and advancing it every time this page is called.

```
    session("orderCount")=session("orderCount")+1
    orderCount=cStr(session("orderCount"))
```

We now build an XML fragment from the information passed fromcatpage1.aspx.

```
'add the order


    ordXML="<prod_name>" & prod_name & "</prod_name><prod_code>" & _
      prod_code & "</prod_code><supp_name>" & supp_name & _
      "</supp_name><supp_uid>" & supp_uid & "</supp_uid><ord_qty>" & _
      ord_qty & "</ord_qty><prod_price>" & prod_price & "</prod_price>"
```

We might already have an order list stored as an XML fragment in session (*"cartXML"*). If so, we add this XML fragment to session (*"cartXML"*); if not, we create the new session variable:

```
    session("cartXML")=session("cartXML") & "<orderitem orderCount='" _
      & orderCount & "'>" & nl & ordXML & nl & "</orderitem>" & nl
```

To make this XML fragment into a well-formed XML document, we need to enclose it in a root element.

```
displayXML="<display>" & nl  & session("cartXML") & nl &  _
  "</display>"
```

Now we'll use the Try/Catch to load our XML document into a DOM object.

```
Try
 oXML.loadXML(displayXML)
 'response.write("<b>*Document Loaded</b>")
Catch oError as exception
  response.write("<b>* Error while accessing document</b>.<br />" _
        & oError.Message & "<br />" & oError.Source)
  exit sub
   End Try
```

Now, it is just a simple matter of iterating through this object and writing it out for display.

```
xList=oXML.documentElement.childNodes


strResult="<table border='1' cellpadding='5' rules='rows'>"
strResult+="<tr><th>Order #</th><th>Product Name</th>_
  <th>Vendor</th><th>Qty.</th><th>Price</th><th>Total</th>_
  <th>Remove</th></tr>"


For Each oNode In xList



          strResult+="<tr>"
          strResult+="<td>" & oNode.Attributes.item(0).Value & _
            "</td>" 'GetAttribute("ordercount")& "</td>" '
          strResult+="<td>" & oNode.FirstChild().FirstChild().Value _
            & "</td>"
          strResult+="<td>" & oNode.FirstChild().nextSibling_
            .nextSibling.FirstChild().Value & "</td>"
```

```
            strResult+="<td>" & oNode.FirstChild().nextSibling_
              .nextSibling.nextSibling.nextSibling.FirstChild().Value _
              & "</td>"
            strResult+="<td>" & oNode.FirstChild().nextSibling_
              .nextSibling.nextSibling.nextSibling.nextSibling_
              .FirstChild().Value & "</td>"
            iQty=cInt(oNode.FirstChild().nextSibling.nextSibling_
              .nextSibling.nextSibling.FirstChild().Value)
            dPrice=cDbl(oNode.FirstChild().nextSibling.nextSibling_
              .nextSibling.nextSibling.nextSibling.FirstChild().Value)
              if iQty < 0 then
                iQty=0
              end if
          strResult+="<td align='right'>" & iQty*dPrice & "</td>"
          dGrandTotal=dGrandTotal +0 + (iQty*dPrice)

          strResult+="<td><form action='shopcartremove.aspx'><input _
            type='hidden'  name='removeoc' value='" & _
            oNode.Attributes.item(0).Value & "' /><input _
            type='hidden' name='remove' value='remove' /><input _
            type='submit' value='Remove' /></form></td>"
          strResult+="</tr>"

      Next
      strResult+="<tr><th colspan='5'> Grand Total</th><th _
        align='right'>" & dGrandTotal & "</th><th> </th></tr>"

    strResult+="</table>"
    cartitems.innerHtml=strResult
          ta.innerHTML=displayXML



  end sub
</script>
  <netforce:header3 runat="server" />
```

```
<br /><a href="catpage3.aspx">catalog</a>
<div id="cartitems" runat="server"></div>

<table>
<tr><td>
<form action="shopcartremove.aspx">
<input type="hidden" name="clearall" value="clearall"/>
<input type="submit" Value="Clear Basket" />
</form>
</td>
<td><form action="checkout.aspx">
<input type="hidden" name="checkout" value="checkout"/>
<input type="submit" Value="Checkout" />
</form>
</td></tr>
</table>
<form>
<textarea id="ta" rows="10" cols="80" runat="server"></textarea>
</form>

</html>
```

When our client is at this page, he or she has the option to remove an individual item from the basket, clear the whole basket, check out, or, of course, go back to the catalog and add another item!

If the client chooses to remove an item, the item will be removed in shopcartremove.aspx. We will not show the complete listing here; we will just show the part where the item is removed.

The session XML string fragment is retrieved and converted to a well-formed document.

```
displayXML="<display>" & nl  & session("cartXML") & nl &  "</display>"
```

It is then loaded into an XML object.

```
Try
   oXML.loadXML(displayXML)
   'response.write("<b>*Document Loaded</b>")
```

```
Catch oError as exception
  response.write("<b>* Error while accessing document</b>.<br />" _
        & oError.Message & "<br />" & oError.Source)
  exit sub
   End Try
```

You will remember that we gave each item in the shopping basket a unique number. When the client clicked on the button to remove the item, this number was passed to *:shopcartremove.aspx:* as *:removeoc:*. In the following code, a search is made for this number, and when found, the element is removed from the XML document.

```
xList=oXML.documentElement.childNodes
  'response.write (xList.count)
' here we need to remove the node, and redeclare the xList
dim i as integer
 if remove="remove" then
   For i=0 to  xList.count-1
    if xList.item(i).Attributes.item(0).Value=removeoc then
      'response.write("CCCCC" & removeoc)
      oXML.documentElement.removeChild(xList.item(i))
      exit for
    end if

   next
 end if
```

The session (*"cartXML"*) string is then rebuilt with this value removed, and the page is displayed.

If the **Checkout** button is clicked, the client is taken to the page seen in Figure 10.14.

After this, the client is taken to a Thank You page (Figure 10.15).

NOTE

These pages displayed in Figures 10.14 and 10.15 feature no new techniques, so we will not analyze them here, but the commented code is available on the Solutions Web site for the book.

Figure 10.14 GUI: The Checkout Page

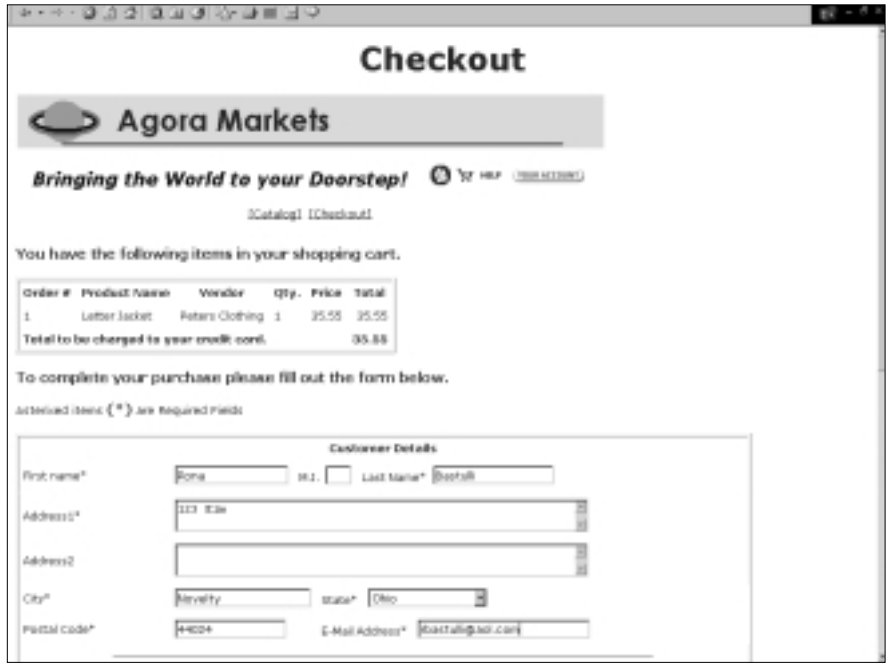


Figure 10.15 checkout2.aspx

Business and Web Services

This catalog is fine, and is like thousands of other similar Internet applications that are available on the Web. What Agora Markets would like to do, though, is implement the catalog as a Web service. They would like any application anywhere to be able to pass a parameter—say, a search string to this service—and get back an XML file of all the items from the catalog.

In ASP.NET it is unbelievably easy to convert an Internet application such ours into a Web service. We will look at how this is done in the rest of this chapter, first converting the catalog into a "home rolled" business service, and then with the alteration of a few lines of code, converting it into an ASP.NET Web service.

Business versus Web Services

These terms are relatively new terms in the Internet Lexicon, and are often used interchangeably. Like many new words, though, their meaning is evolving. A consensus seems to be emerging that a *business service* is a transaction or some other interchange instituted between cooperating business partners. On the other hand, a *Web service* is an interchange between anonymous business partners. In any case, this is the way we will be using the two terms here.

In a business service, the trading partners will know what to expect of one another. In a Web service, we must not only supply the service, we must also let the client know what the service is about, what he can expect to receive when he initiates it, and in what format he will receive his information.

Coding a Business Service

First, we will institute a business service. We will code an ASP.NET page that, after being passed a search string for an article in our catalog, will return an XML recordset containing all the pertinent records that match the string. The code in Figure 10.16 is a modification of the code that we used in the *"body1" userControl* called bservices2.aspx that can be found on the Solutions Web site for the book. Our code will be recyclable; this can make our coding really simple.

SYNGRESS
syngress.com

Figure 10.16 Code Listing of bservices2.aspx

```
<%@ Import Namespace="System.Data"%>
<%@ Import Namespace="System.Data.OleDb"%>
```

Continued

Figure 10.16 Continued

```
<html>
<title>bservice1 file</title>
<script language="VB" runat="server">

 Sub Page_Load(Source As Object,E as EventArgs)

'declare variables
    Dim prod_name as string
    Dim strResult as string

    prod_name=request.params("prod_name")



'database variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataAdapter as OleDbDataAdapter
    'Dim oDataSet as DataSet()
    Dim strSQL as String
    Dim strConn as String

    strConn="Provider=Microsoft.Jet.OLEDB.4.0"
    strConn=strConn & "; Data Source=
c:\inetpub\wwwroot\syngress\dbases\catalog.mdb"



    strSQL="SELECT products.*,supplier.supp_name FROM products,supplier _
      WHERE products.supp_uid=supplier.supp_uid AND products.prod_name _
      LIKE'%" & prod_name & "%' ORDER BY prod_name;"



    oConn=New OleDBConnection(strConn)
    oConn.open
```

Continued

Figure 10.16 Continued

```
    oDataAdapter=New OleDbDataAdapter(strSQL,oConn)
    oDataSet=New DataSet()
    oDataAdapter.Fill(oDataSet,"item")



    'oDataSet.WriteXML("c:\frank\dataset.xml")
    strResult=oDataSet.GetXML
     outResult.InnerHtml =replace(strResult,"<","&lt;")

    'response.write (strResult)
'housekeeping
 oConn.close
 'note let and set not needed or supported
 end sub
</script>

<pre id="outResult" runat="server"></pre>
```

In this example, the code will be printed out as a Web page, so that it can be clearly presented in the book. However, in reality, you would probably want to wrap it in something more accessible to client-side code, such as a hidden attribute in a form. A screenshot of a sister application is shown in Figure 10.17. This application accepts a vendor's user IDs as a parameter and returns an XML file of all that vendor's products. Both sets of code (which are almost identical) can be found on the Solutions Web site for the book.

Analysis of Code

NOTE

We will only discuss the aspects of code that differs from the discussion of the code in "body1.ascx".

Figure 10.17 bservice1.aspx Showing an XML Record Set of the Products of King's Hardware

We have imported the namespaces in the usual way, and the first part of the code is similar to our *userControl "body1.ascx"*, right up to where we open our Connection object. Of course, because we are only going to be servicing one type of parameter, we have removed some of the string declarations. We have also declared two new objects:

```
Dim oDataAdapter as OleDbDataAdapter
Dim oDataSet as DataSet()
```

The *DataAdapter* object serves as a pipeline and provides the logic to populate the *DataSet* object. The *DataSet* object provides the basis for the storage and manipulation of data that we have downloaded from our data store. It allows us to work with it while disconnected from our data store; then, if necessary, we can pass all the changes back into our data store when we have finished.

The *DataAdapter* object is instantiated, and takes our SQL query and the Connection object as a parameter. This fills it with the equivalent of a recordset.

```
oDataAdapter=New OleDbDataAdapter(strSQL,oConn)
```

Once the *DataAdapter* has been instantiated, we can use it to fill the *DataSet*.

```
oDataSet=New DataSet()
oDataAdapter.Fill(oDataSet,"item")
```

The Fill method fills the *DataSet* that is passed to it with the data that has been downloaded from the data store. The second parameter *"item"* is the name we wish to give the table in the *DataSet*.

Now that we have all the information in the *DataSet*, we need to write it out as XML. In ASP .NET, this is almost trivial (although of course the "behind the scenes" code is far from trivial). The *GetXML* method returns all the dataset as an XML string.

```
'oDataSet.WriteXML("c:\frank\dataset.xml")
strResult=oDataSet.GetXML
```

Note that it is equally easy to write the XML to file for debugging or other purposes! The line of code that does this is commented out here.

Once we get the XML, all we need to do is clean up, and display the XML in the Web browser.

```
 outResult.InnerHtml =replace(strResult,"<","&lt;")


 'response.write (strResult)
'housekeeping
 oConn.close
 'note let and set not needed or supported in ASP.NET
 end sub
</script>


<pre id="outResult" runat="server"></pre>
```

Note that in ASP.NET, it is no longer necessary to set the Connection object to Nothing after we have closed it. This is due to the efficient method of Garbage Collection that the .NET Framework employs.

Creating a Web Service

This code is fine between cooperating business partners, because all the partners know what to expect and how to handle it. What if we want to release the code on the wide world that has no idea as to our methods? To do this, we will need to use a Web service. Luckily, once we have the code in place, ASP.NET makes it is a simple matter to convert it to a Web service.

An Overview of Web Services

Every day, millions of Web pages are opened in browsers, information is scanned, and, if necessary, it is acted upon. This is the default operation of the World Wide Web. This is accomplished via. HTML streams sent using the HTTP protocol. However, all these operations have one thing in common: they require a human intermediary.

Although impressive, current usage is just scratching the surface of the Web's capabilities. Among others, the inventor of the Web—Tim Berners-Lee—envisages a world where Web application will be able to talk to Web application without human intervention. To do this, resources must be both discoverable and programmable. This is where Web services will be required.

A Web service allows any user application to access our service, find out whether it is suitable for their use and matches their needs, and if it is, incorporate it into their own application. The implementation is language neutral. Our Web service is written in VB using ASP.NET, but it could just as well have been written in Perl or Java, and the accessing application can also be written in any language. Using a standard information exchange protocol such as SOAP (Simple Object Access Protocol), SOAP-enabled applications will be able to speak to other machines and use the methods built into these machines, thus allowing cooperation between various distributed applications.

For all this to happen, though, in order to add our catalog to the list of such cooperating machines, we must first expose it as a Web service.

Coding a Web Service

In order to convert the code from our own "hand-rolled" "business service' into an ASP.NET Web service, it is necessary to make a few alterations, and save it with the extension ".asmx". Figure 10.18 is the code listing for bservice2.asmx, and the source code in its entirety can be found on the Solutions Web site for the project.

SYNGRESS
syngress.com

Figure 10.18 Code Listing bservice2.asmx

```
<%@ Webservice class="Catalog" Language="VB"%>

 Imports System.Web.Services
 Imports System.Data
 Imports System.Data.OleDb



 <WebService(Description:="Get all the items in the catalog that match _
   the string passed to the web Service", Namespace:=_
   "http://www.nforce.com/webservices/")> Public Class Catalog


 <WebMethod> Public Function getCatItems(prod_name as String) as String
'declare variables
    'Dim prod_name as string
    Dim strResult as string



'database variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataAdapter as OleDbDataAdapter
    'Dim oDataSet as DataSet()
    Dim strSQL as String
    Dim strConn as String

    strConn="Provider=Microsoft.Jet.OLEDB.4.0"
    strConn=strConn & "; Data Source=
c:\inetpub\wwwroot\syngress\dbases\catalog.mdb"

    'vendors="toolking"
    strSQL="SELECT products.*,supplier.supp_name FROM products,supplier _
      WHERE products.supp_uid=supplier.supp_uid AND products.prod_name _
      LIKE'%" & prod_name & "%' ORDER BY prod_name;"



    oConn=New OleDBConnection(strConn)
```

Continued

Figure 10.18 Continued

```
    oConn.open

    oDataAdapter=New OleDbDataAdapter(strSQL,oConn)
    Dim oDataSet as New DataSet()

    oDataAdapter.Fill(oDataSet,"item")

    strResult=oDataSet.GetXML

    Return strResult
'housekeeping
 oConn.close
 'note let and set not needed or supported
 end function
end class
```

You can see that we have only made a few changes from bservice2.aspx.

Analysis of Code

The main thrust of our changes has been to convert the page from a standalone page to a *WebService* class. We announce this as follows:

```
<%@ Webservice class="Catalog" Language="VB"%>
```

Next, we need to change the way we import our ASP.NET namespaces:

```
 Imports System.Web.Services
 Imports System.Data
 Imports System.Data.OleDb
```

Next, we need to declare the class as Public, and to associate it with a namespace and also provide a description of the Web service that we are offering (note that this should all be on one line).

```
<WebService(Description:="Get all the items in the catalog that match _
  the string passed to the web Service", Namespace:=_
  "http://www.nforce.com/webservices/")> Public Class Catalog
```

If we omit the Webservice namespace declaration, then ASP.NET will provide a default namespace. However, eventually, in order to make our class unique amid

the millions of other ASP.NET classes, we will have to provide one, so we might as well do it from the get go.

Next, we declare our function. Note that this is a function and not a subroutine (as it was in *"bservices2.aspx"*) (note that this should all be on one line).

```
<WebMethod> Public Function getCatItems(prod_name as String) as String
```

The rest of the code is identical until:

```
    Return strResult
'housekeeping
 oConn.close
 'note let and set not needed or supported
 end function
end class
```

The *strResult* variable is now mapped as a return value for the function. We close both the function and the class.

Now that we have persisted our application as a Web service, we need to test it.

Testing the Web Service

We now have a functional Web service *"catalog."* ASP.NET allows us to test it by requesting the URI in our browser. Here is what we get when we enter the *"bservice2.asmx"* address. (Keep in mind that this functionality is only for testing our product, it is not part of what we are going to be sending to the customers of our Web service.)

The first line shows the description that we have given to our Web service. The bulleted item shows the public function that we have declared. Figure 10.19 shows our catalog as a Web service.

Figure 10.19 bservice2.amx

When we click on the bulleted item, we will get a screen that will invite us to insert a parameter (Figure 10.20).

Figure 10.20 Insert Parameter Screen

Note also that the default way to send the parameter is as a SOAP message.

Then, when we insert the parameter **Jacket** and click the **Invoke** button, we will get a screen showing the return value, which in our case is an XML file (Figure 10.21).

If you click the **View/Source** header, you will see that all our opening angle brackets (“<”) have been converted to the XML attribute *“<”*. This is because we have returned the information as a string in our function, and the testing application chooses to display it as a string in the browser.

We have gone to all this trouble to create a Web service, but who is going to use it? Indeed, why should we create a Web service? We create a Web service so that a third party can use it. Although a full description of using Web services is beyond the scope of this book, in the next section we will look at the current technologies available, and point you to where you can find out more about using Web services.

Figure 10.21 The Invoke Return Value

Using Web Services

In the previous section we used ASP.NET to create a Web service out of our catalog. This section just gives a brief overview as to how this service could be used by a third-party application.

For clients to use a Web service, they first have to find a suitable Web service, and then they have to get a description of that service. If this process is to be automated, these processes must be accessible to machines. We will see how this can be done using UDDI and WSDL.

Universal Description, Discovery, and Integration

UDDI is an open venture initially sponsored by Ariba, IBM, and Microsoft in September 2000. Since then, more than 200 other companies have joined the project. For more information on UDDI, you should refer to Microsoft's UDDI documentation at http://uddi.Microsoft.com; you can find the UDDI white paper from Microsoft at http://uddi.microsoft.com/developer/default.aspx.

In order to register a service, the service must be described in a standard way. One way to do this, and a way made almost trivial by ASP.NET, is to use the Web Service Description Language (WSDL). We will look at this now.

Web Service Description Language

The Web Service Description Language (WSDL) can be found in a W3C note dated March 21 2001 at www.w3.org/TR/wsdl. In an nutshell, WSDL is a way to use XML to describe various network services as endpoints of messages; each message can contain document or procedural information.

In order to expose our Web service, we need to create a WSDL document. We have in fact already done all the work to create a WSDL file when we created bservice2.asmx! You will obtain an interface identical to Figure 10.19 when we went to test our Web service.

If we click on **Service Description**, we will get the screen shown in Figure 10.22.

Figure 10.22 WSDL Service Description Screen

View/Source will show us the underlying WSDL file (Figure 10.23)!

Figure 10.23 bservice2.asmx WSDL Source

```
<?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/"
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="http://www.nforce.com/webservices/"
targetNamespace="http://www.nforce.com/webservices/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
```

Continued

Figure 10.23 Continued

```
<types>
  <s:schema attributeFormDefault="qualified" _
    elementFormDefault="qualified" targetNamespace=_
    "http://www.nforce.com/webservices/">
    <s:element name="getCatItems">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="prod_name" _
            nillable="true" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="getCatItemsResponse">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" _
            name="getCatItemsResult" nillable="true" type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="string" nillable="true" type="s:string" />
  </s:schema>
</types>
<message name="getCatItemsSoapIn">
  <part name="parameters" element="s0:getCatItems" />
</message>
<message name="getCatItemsSoapOut">
  <part name="parameters" element="s0:getCatItemsResponse" />
</message>
<message name="getCatItemsHttpGetIn">
  <part name="prod_name" type="s:string" />
</message>
<message name="getCatItemsHttpGetOut">
  <part name="Body" element="s0:string" />
</message>
```

Continued

Figure 10.23 Continued

```
<message name="getCatItemsHttpPostIn">
  <part name="prod_name" type="s:string" />
</message>
<message name="getCatItemsHttpPostOut">
  <part name="Body" element="s0:string" />
</message>
<portType name="CatalogSoap">
  <operation name="getCatItems">
    <input message="s0:getCatItemsSoapIn" />
    <output message="s0:getCatItemsSoapOut" />
  </operation>
</portType>
<portType name="CatalogHttpGet">
  <operation name="getCatItems">
    <input message="s0:getCatItemsHttpGetIn" />
    <output message="s0:getCatItemsHttpGetOut" />
  </operation>
</portType>
<portType name="CatalogHttpPost">
  <operation name="getCatItems">
    <input message="s0:getCatItemsHttpPostIn" />
    <output message="s0:getCatItemsHttpPostOut" />
  </operation>
</portType>
<binding name="CatalogSoap" type="s0:CatalogSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" _
    style="document" />
  <operation name="getCatItems">
    <soap:operation soapAction="http://www.nforce.com/webservices/_
      getCatItems" style="document" />
    <input>
      <soap:body use="literal" />
    </input>
    <output>
```

Continued

Figure 10.23 Continued

```
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <binding name="CatalogHttpGet" type="s0:CatalogHttpGet">
    <http:binding verb="GET" />
    <operation name="getCatItems">
      <http:operation location="/getCatItems" />
      <input>
        <http:urlEncoded />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>
  <binding name="CatalogHttpPost" type="s0:CatalogHttpPost">
    <http:binding verb="POST" />
    <operation name="getCatItems">
      <http:operation location="/getCatItems" />
      <input>
        <mime:content type="application/x-www-form-urlencoded" />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>
  <service name="Catalog">
    <documentation>Get all the items in the catalog that match the string _
      passed to the web Service</documentation>
    <port name="CatalogSoap" binding="s0:CatalogSoap">
      <soap:address location="http://localhost/syngress/bservice2.asmx" />
    </port>
```

Continued

Figure 10.23 Continued

```
    <port name="CatalogHttpGet" binding="s0:CatalogHttpGet">
      <http:address location="http://localhost/syngress/bservice2.asmx" />
    </port>
    <port name="CatalogHttpPost" binding="s0:CatalogHttpPost">
      <http:address location="http://localhost/syngress/bservice2.asmx" />
    </port>
  </service>
</definitions>
```

This document is divided into five major sections:

- **Types** The types section of the document is simply an XML schema used to define the message section.
- **Message** For each Web service, there will be at least one message. In our case, there is e reply, so there is a minimum of two messages. Two message elements, an in and an out, are generated for the SOAP, GET, and POST protocols. Here are the message elements for the SOAP protocol:

```
<message name="getCatItemsSoapIn">
    <part name="parameters" element="s0:getCatItems" />
  </message>
  <message name="getCatItemsSoapOut">
    <part name="parameters" element="s0:getCatItemsResponse" />
  </message>
```

The information that is expected—namely, a string in and a string out—is defined in the types section of the document. You will notice that the messages describing the GET and the POST messages carry the parameter type and the response type as attributes.

Installation: Migrating to SQL Server

Like many developers, I prototype and develop many applications in Access with the idea of migrating to a more robust database at a later stage. This makes sense for most of us; I travel a lot, and although I do have a copy of SQL Server on my laptop, it is much handier to just use Access as the test bed. Furthermore, not all of my applications will be deployed on SQL Server; they might be deployed on numerous other databases. The description of the migration process that is given

here is for SQL Server, but the general principles apply to other databases. The general processes that we need to go through are:

1. Change the connection string.
2. Make sure that the data types are compatible.
3. Make any necessary changes to the SQL strings.
4. If converting to SQL Server, change all namespaces to the SQL equivalent.

Changing the Connection String

Here is the connection string I have been using on my laptop:

```
strConn="Provider=Microsoft.Jet.OLEDB.4.0"
strConn=strConn & "; Data Source=
c:\inetpub\wwwroot\syngress\dbases\catalog.mdb"
```

Here is a typical SQL server connection string:

```
strConn="Provider=SQLOLEDB;Data Source=D1C4FN01;Initial Catalog=catolog;"
strConn=strConn & "User Id=frankb;Password=dogs;Network Library=dbmssocn;"
```

You can see that the provider name has been changed, the data source is no longer a path, but the address of the database server, and the name of the database to be accessed is given by the parameter *"Initial Catalog"*. (Somewhat confusingly our database is named *"catalog,"* but if it was named *"tradelist"*, then the parameter and value would be *"Initial Catalog=tradelist;"*) The other information just passes the password and user ID to the server.

Connection strings for other providers are listed in Table 10.1.

Table 10.1 Database Connection Strings

MySQL	`"DRIVER={MySQL};SERVER=MySQL_development; UID=frankb;PWD=dogs;DATABASE=MySQL"`
Oracle 8 and 8i	`"Provider=MSDAORA;Data Source=Oracle8_development; User ID=frankb;PWD=dogs;"`
IBM DB2	`"DSN=catalog;UID=frankb;PWD=dogs;Database=catalog;"`

Compatible Data Types

Different databases use different data type definitions. For example, the text data type in Access can hold a maximum of 255 characters. It's equivalent in SQL Server is the "char" data type that can hold 1024 characters. Obviously, there is

no problem going from Access to SQL Server in this case, but there could be if we were going in the opposite direction.

SQL Strings

The main problem here is with date and time data types. Most databases want these values in either a "bare" form or as character date.

Consider the following SQL string:

```
SELECT * FROM invoices WHERE inv_date > '1/1/01';
```

This would select all records dated after January 1, 2001. This would work perfectly well with SQL Server, as would the following "bare" date.

```
SELECT * FROM invoices WHERE inv_date > 1/1/01;
```

NOTE

The ANSI SQL standard calls for dates to be put in single quotes, so they should always be used where supported by the DB. To my knowledge, Access97 and earlier is the only mainline DB that does not support this format.

However, both of these would cause an error in Access. Access requires the date data type to be between hash marks, thus:

```
SELECT * FROM invoices WHERE inv_date > #1/1/01#;
```

Unfortunately, this form is not ANSI SQL, so it will fail in just about every other type of database. This means that you should go through and convert all your dates to the correct format. (A simple regular expression can be used for Access to ANSI SQL, but it is not so simple the other way around.)

Converting to SQL Server

In order to convert to SQL Server, the names of the namespaces must be altered, and all the "oledb" methods need to be altered to their SQL equivalents. For example, the following code contains the headers and data variables from one of our applications:

```
<%@ Register tagprefix="netforce" tagname="header5" _
  src="usercontrols/header5.ascx" %>
<%@ Import Namespace="System.Data"%>
<%@ Import Namespace="System.Data.OleDb"%>
```

```
<%@ Import Namespace="System.XML"%>

'data variables
    Dim oConn as OleDBConnection
    Dim oComm as OleDBCommand
    Dim oDataReader as OleDbDataReader
    Dim strSQL as String
    Dim strSQL2 as String
    Dim strConn as String
    Dim oTransaction as OleDBTransaction
```

Change these as follows:

```
<%@ Register tagprefix="netforce" tagname="header5" _
  src="usercontrols/header5.ascx" %>
<%@ Import Namespace="System.Data"%>
<%@ Import Namespace="System.Data.SQLClient"%>
<%@ Import Namespace="System.XML"%>

 'data variables
    Dim oConn as SQLConnection
    Dim oComm as SQLCommand
    Dim oDataReader as SQLDataReader
    Dim strSQL as String
    Dim strSQL2 as String
    Dim strConn as String
    Dim oTransaction as SQLTransaction
```

This is quite easily done with the search and replace functions in the IDE or text editor that you use.

Summary

In this chapter, we looked at using ASP.NET to build a real-world application; namely, a catalog attached to a shopping cart. We also looked at how this catalog could be automatically updated using XML messages, and how the catalog could be converted to both a business service and a Web service.

We took a look at some real-world situations in which modeling took place. A business model is essential when working for almost any company; although, if you are working as a contractor or subcontractor, the business model might be given to you by the client or another contractor. Having a business model also helps you with your database design.

XML versus traditional database design was another topic we touched on; XML, while a powerful tool for transmitting database information, does not currently have adequate encryption and cannot perform as fast as a database when dealing with large recordsets. However, XML is perfect for dealing with small parts of data that do not pose any type of security problem. We also looked at how to transport this data and how it might be wrapped.

Our final database application enveloped all of the skills found in this book, along with some pretty neat coding. By being able to work with both XML and ADO.NET, a developer will be able to both expand his programming skills and facilitate data communication.

Solutions Fast Track

Basic Design Considerations

- ☑ While XML is a robust data storage language, it is unable to cope with large transactions, which are typically handled by databases such as SQL Server. A combination of both XML and a database is the best solution.
- ☑ EDI, while an option, is usually too expensive for a small or medium-sized business. A .NET application with XML can usually compensate for an EDI system without requiring EDI itself.
- ☑ Proper organization at the beginning stages of an application development is vital for the overall development, since it helps to remove possible errors and unneeded functionality early on.

Coding the Project

- ☑ A database needs to avoid redundant entries of data; this is normally referred to as the *normalization* process.
- ☑ There are seven levels of normalization for a database, but the majority of database applications only require at least three levels of normalization.
- ☑ A database can be either OLTP (transitional) or OLAP (application).

XML Packages Design

- ☑ An XML file needs to be able to meet goals, just like a database, and provide the needed functionality.
- ☑ XML files also need to adhere to the normalization levels a database needs.
- ☑ In many applications, including B2B, a schema for validating the XML file is agreed upon by the parties involved. This can mean that the schema might be in DTD.

Customer Interface Design

- ☑ Our customer interface always needs to be pleasing to the eye and able to allow customers to easily find the information they require.
- ☑ By using XHTML with ASP.NET, we are able to provide a majority of the functionality we require for our customers to use.
- ☑ By properly using a combination of XSLT and XML, we are able to accurately portray the items to the customers.

Business and Web Service

- ☑ A business service is a transaction between two businesses that are working together and have full knowledge of the data that is being passed to each other. This also means that both sides know what to expect at all points and are familiar with the service being used.
- ☑ A Web service, on the other hand, is a transaction between two users who are anonymous and are not related to each other. This means that

we need to supply the service, information about the service, what data will be received, and the format of the data that will be received.

- ☑ There are various platforms for Web services, such as SOAP or LDAP. If you are working 100 percent with .NET, you might want to consider using SOAP, since it is Microsoft's initiative and is integrated into .NET.

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** and click on the **"Ask the Author"** form.

Q: What is OLAP?

A: OLAP, Online Analytical Processing, is a database that is geared toward the analytical processing of its data. These databases are usually configured for speed and are used heavily when serious analytical processes are required.

Q: What is OLTP?

A: OLTP, Online Transactional Processing, is a database that is geared toward perfecting the integrity of a database. These databases will usually fulfill all three normalization levels or more.

Q: What is meant by ADO.NET?

A: ADO.NET is the .NET version of ADO. ADO.NET is able to take advantage of the improvements .NET provides, and uses XML heavily internally.

Q: What is SOAP, and how can I find out more about it?

A: SOAP, the Simple Object Access Protocol, is a protocol for processing remote objects through standard HTTP requests. You can find out more about SOAP at www.microsoft.com/mind/0100/soap/soap.asp.

Q: When should I validate a document?

A: Anytime you have to work with an XML structure that is either used repeatedly or repeatedly generated dynamically, you should provide validation in order to promote proper XML formation and to test to see if the XML file generated is correct.

Q: When I validate a document, should I use a DTD or XML schemas?

A: This answer is a toss-up, really. Generally, you should use what you are comfortable with, but, as this example showed us, sometimes the choice is not left to us. While .NET is geared toward working with schemas, DTDs have been around longer and many still use them. You should learn to work with both types of validation and understand their strengths and weaknesses.

Index

3DES. *See* Triple DES

A

Abstract classes, 31–32
acceptNode(), 121
Access
  database. *See* Disconnected access database; Microsoft access database
  object, creation. *See* Data
  permissions. *See* Code
    limitation, 166
  security. *See* Code access
Access (Microsoft), 285, 313, 445
  database, 286
  usage, 456
Account element, 254, 261
Active connection, 417
ActiveState, 69
ActiveX Data Objects (ADO), usage, 441, 443. *See also* Message board creation
addCategory() method, 149
addEntry() method, 151
AddRef, 4
Address block, returning, 34
addressBook
  element, 122, 146–147
  node, 150
Adjust Security Wizard, 200
admin.aspx, 372
Admin/banned status, 390
Administrative interface, construction, 389–402, 405
Administrator policy, applying, 16–17
ADO. *See* ActiveX Data Objects
ADO.NET, 93, 136
  architecture, 409, 410
  comparison, 414. *See* ADO
  Framework, 273
  understanding, 408–414, 448
  usage, 233. *See also* Data
Advanced Network eXchange (ANX), 455, 457
Algorithm. *See* Asymmetric key algorithm; Hash algorithm; Symmetric key algorithm
All Code, 170, 202
All_Code, 194
  code, 205
  group, 201
Allocated space, 33
American Standards Institute, 456
Analytical databases, 463
Anchor tag. *See* HyperText Markup Language
ANSI SQL, 531
ANX. *See* Advanced Network eXchange
API. *See* Application Programming Interface
AppBindingMode element, 14
AppDomain element, 14, 16
Application. *See* Self-describing application
  creation. *See* Console applications
  design, 321–357, 404
  domains, 46–47
  enforcement, 23
  exposure, 180
  interaction, 4

role-based verification, 188
Application Programming Interface (API), 4, 441. *See also* Metadata
capabilities. *See* Document Object Model
differences, 233
set, 18
ApplyStyles method, 365
AppSettings, 361, 365
ArgumentException, 330–331, 338, 343
ArrayList, 347
Arrays, 20
ASC X12 standards, 455, 456
ASCII text, 429–431
ASP
code, 258
programmers, 483
scripting, 233
scripts. *See* Top-down ASP scripts
usage, 484–485
version 4, 232
ASPcontrol declaration, 293
ASP.NET, 5, 46, 232, 288, 290, 457
applications, copying, 357, 358
classes, 522
development, 65
FAQs, 281–282
languages, choice, 233
namespaces, 521
page, 460
platform, 232–233, 278–279
session objects, 503
solutions, 278–281
usage, 457, 465, 503
XML control, 298
ASPX, 70
ASPX code, 298
Assembly, 6. *See also* Policy; Static assembly
access, 22
binding, 12
cache, 11–12
usage. *See* Global assembly cache
code. *See* Unmanaged assembly code
creation, 5–17, 50, 174. *See also* Multifile assemblies
dependencies, 21
description, 2
enabling, 166
enumeration, 22
filename, 6
files. *See* Private assembly files; Shared assembly files
granting, 166
identification, metadata usage, 18–19
incorporated evidence, 168
locating, 12–17
location, 22, 168
members. *See* Local assembly members
options, 8–10
references. *See* External assembly
request. *See* Permissions
trust, 176, 194
trustworthiness, 169
unit, 3
AssemblyRef, 12, 15
Assert Override (command), 179–182
Assert (security action), 179
Assigned permission list, 196, 197
Asymmetric algorithm, 209
Asymmetric key algorithm, 208, 209
Asymmetric key-pair, 209
attribute(), 22
Attribute (Attr) element, 117
AttributeCount property, 234

Attributes, 22, 89, 122
  interface, 117
  label, 90
  list, 113
  name, 92
  tag name, 113
  value, 113
Authentication, 165, 207
  determination. *See* Principal
  methods, 165
AuthenticationType (property), 192
Author
  comment, display. *See* Message
  e-mail address, identification, 284
  identification, 284
Authorization, 165
Auto hide (setting), 75
Automatic resource management, reliance, 32–41, 51

B

Backward compatibility, 23. *See also* Dynamic Link Library
Backward-compatible class, creation, 24
Balance node, 262
Bank/Account, 258
Bank/Account/Name, 257
Banner user, 393
Base64
  algorithm, 430, 432
  encoded data, reading, 436–440
  encoding
    process, 439
    usage, 408
  format, binary data conversion, 431–436
  formatted data, 431
  function, 429–431
  usage, 429, 437. *See also* Binary data
Basket, clearing, 511
Batches
  looping, 485–486
  loops, 486
Binary conversion algorithm, 429
Binary data
  conversion. *See* Base64
    Base64, usage, 428–440, 449
  handling, 429
Binary format, 432
Binary formatted data, 439
Binding mode, 14
BindingRedir element, 14
Bit stream, encoding, 429
bmp
  format, 431
  node, 439
  subelement, 435
Board
  browsing, 373–382
  class, design, 335–357
  deletion, 396, 399
  name, 378
BoardDescription, 313, 314
BoardID, 313, 314, 340, 345
BoardName, 313–314
Body field, 353
Boolean operators, 105
Boolean property, 360
Boundary. *See* Reference; Security; Types;Version boundary
  points, 122
Boyce-Codd level, 464
Browsing. *See* Board; Message
  interface, design, 373–382, 404
btnCreate_Click method, 395

btnEditPost_Click method, 402
BtnLogin_Click, 372, 385
btnRegister_Click subroutine, 368
Build errors, 67
Built-in classes, 65
Built-in XSL transformer, 264
Business
  services, 462, 519
    coding, 514–518, 534
  Web services, interaction, 514–532, 534–535
    contrast, 514
Business-logic processing, 409
Business-related information, 161
Business-to-business (B2B)
  application, 452, 483
  design, 462, 469–490
  e-communications, 459–460
  system architecture, 84

C

C#, 5, 69, 99, 232
  code, 121, 122
  pages, XML document comments (addition), 74–75
  security, 162
  Web application, 59
  Windows application, 145
C++, 5, 69
  ATL
    code, 322
    COM, 322
  user, 58
CA. *See* Certificate authority
Cache. *See* Assembly; Download cache; Machinewide code cache
  usage. *See* Global assembly cache
Caller identity, usage, 166
Calling chain, 173
Calling code class, 22
CanonicalizationMethod element, 220
cartXML, 508, 512
CAS. *See* Code Access Security
Cascading Style Sheets (CSS), 69
  file, 361
  script, 301
caspol.exe, usage, 201
Catalog, 462
  analysis, 457–462
  construction. *See* Wholesale catalog
  content
    page access, 490
    searching, 491
  element, 89
  image, 492
  implementation, 456–457
  nodes, 91
  page, 490–501
  project, coding, 462–463
  requirements, 457
  updating, 459
Category
  creation/deletion, 149–151
  node, 123
C/C++, 24, 208. *See also* ISO C/C++
  usage, 31
CDATA section, 113
  node, 118
Certificate authority (CA), 209
Certificate file, importation, 203
CFG files, 181
Character date, 531
CharacterData, 122
Check. *See* Role-based security
  overriding. *See* Security

routine, 39
Checkout page, 490
Child nodes, 144. *See also* DocumentElement
ChildNode, 242
ChildNodes.Count, 242
ChildPost, 348–350
ChildThread, 338
CipherText, 216
element, 218
Class. *See* Built-in classes; Count class; External class; Framework; Skeleton class
design. *See* Board; MessageBoard class; Post; PostList class; Thread; ThreadList class; User
loader, 42
organization, 28–29, 51
view, 65–66
Class contracts, 20
ClassAct class, 176, 178
ClassActing, 178
Class-level variable, 147
ClassName, 22
ClassReacting, 178
Class_Terminate event, 39
CLI. *See* Command-line interface
Click event, 239
Client application, 428–429
Client-server applications, 441
Client-server approach, 442
Client-server Web, 441
Client-side cursor location, 409
Client-side JavaScript, 366
Client-side transformations, 162
Client-side XSL transformations, 212
Close, 239
CLR. *See* Common Language Runtime
CLS. *See* Common Language Specification
Code. *See* ASP; ASPX code; Just-in-time; Managed code; Submit button; Type-safe code
access, permissions, 163
amount, 192
analysis, 482, 493–501, 516–518, 521–522
change. *See* Development code
class. *See* Calling code class
compilation, 27
compiling, 168
completion technology, 68
groups, 166, 169–171, 206. *See also* Zone-based code groups
construction, 194
hierarchy, 193
structure, modification, 200–207
identity, 166, 168–169
impersonation. *See* WindowsPrincipal
protection, 166
sandboxing, 162
trust, 163, 179
window, 59–60
Code Access Security (CAS), 43–44, 162, 166–185, 224–226
model. *See* .NET code
CodeAccess Permission namespace, 22, 186
Codebases, 11. *See also* Uniform Resource Locator (URL)
usage, 16
Code-behind, 369, 379
pages, 69, 401
update, 374
COFF. *See* Common Object File Format
Colors, manipulation, 301–305

COM, 33. *See also* C++; Unmanaged COM
  centricity, 414
  components, 233
  marshalling, 408
  object, 113, 321. *See also* Visual Basic
COM+
  applications, 164
  components, 185
Command, 411
  object, 487, 500
Command-line applications, 26
Command-line Interface (CLI), 192
Command-line programs, 24
Command-line security tools, 210
Command-line tools, 233
Command-line VB application, 27
CommandText property, 424–426
Comment, 88, 285
  display. *See* Message
  replying, 285
Common Language Runtime (CLR), 2–4, 50, 162, 173
  CLR-compliant compiler, 29
  usage, 193, 194, 232
Common Language Specification (CLS), 4
Common Object File Format (COFF), 27
Common Type System (CTS), 2, 29–32, 51, 233
Communication, sequence, 459
Compatibility version, 23
Compiler, 18. *See also* Just-in-time
Compile-time description, 19
complexType data structures, 94, 95
Compression, 34
Con object, 424, 432
Confidentiality, concerns, 161–162
Configuration file, 12
Connection, 411
  object, 517
    creation, 499
  string, changing, 530
Connection-oriented protocol, 441
ConnectionString, 361
Console applications, creation, 334
Console I/O, 26–27
Constructor, calling, 34
Content nodes, 237
Contracts, usage, 20–21
Control.ChildControls ControlCollection, 366
ControlCollection, 365
ControlEvidence, 168
Copy-only installations, 3
Count class, 29
Count() function, 417
Create Post page, 386, 401
CreateBoard method, 338
CreatePost method, 350
CreateRange, 123
CreateThread method, 341, 350
CreateUser method, 329, 333, 368
Creator field, 350
CreatorID, 314
Cross-language support, 29
Cross-platform development, 3
CryptoAPI, 207
Cryptographic Service Providers (CSPs), 207, 208
Cryptography, 207–210, 227
CSPs. *See* Cryptographic Service Providers
CSS. *See* Cascading Style Sheets
CssClass property, 366

CTS. *See* Common Type System
CType, 347
CurRec, 417, 419
CurrentUser
  object, 371
  property, 372
Current.Value property, 260
Custom DataSet, 373
Custom permissions, 163, 167, 184–185, 194
Custom principal, 164
Custom token, 67–68
Customer
  element, 254
  interface design, 462, 490–513, 534
Customer_Id attribute, 255
CustomPrincipal, 186
CustomValidator control, 366
Cyclic objects, 33

D

Data
  access
    advantages/disadvantages. *See* Remote data access
    database, ADO.NET usage, 414–428, 448–449
    object, creation, 323–325
  conversion, Base64 usage. *See* Binary data conversion
  integrity, 212
  management, 442
  mismatches, error trapping, 462
  retrieval, XML document navigation (usage), 236–239
  source, 377. *See also* Disconnected data sources
  storage, 453, 456, 458–459
  stream. *See* Forward-only data stream
  structures, 91. *See also* complexType data structures; simpleType data structures
  transformation. *See* HyperText Markup Language
  types, 124, 461. *See also* Decimal data type; Derived data type; Integer data type; Primitive data type; Text; User-defined data types
    compatibility, 530–531
  typing entries, 461–462
  variables, 483
Data Adapter, 412
Data Encryption Standard (DES), 208
Data Reader, 411
Data Signature Algorithm (DSA), 208
Databases. *See* Access; Analytical databases; Mini database; Virtual database
  ADO.NET, usage. *See* Data
  contrast. *See* Extensible Markup Language
  design, 415, 431–436, 462, 463
  name, 444
  node, 317
  query, usage. *See* Extensible Markup Language document
  setup. *See* Message board creation
  usage, 273–277, 281
  value, reading, 488
  viewer, construction. *See* Remote database viewer
DataBind, 379, 390
DataBinds, 377
DataColumns, 414
Data-consuming application, 409
DataControl, 323–325
  class, 361

DataDocument, 251
  object, 247
DataGrid, 249, 253
DataReader, 411
  content, 500
  creation, 483
  DataSet, contrast, 484
DataRow, 331–332, 355, 414
  objects, 346–347
DataRowCollection, 417, 446
  creation, 435
DataSet, 136, 255, 273, 305, 390. *See also* Custom DataSet; XmlDataDocument
  class, 435
  contrast. *See* DataReader
  creation, 378
  object, 517
  property, 247
  table, 375
  usage, 325, 332, 340, 409, 413–414
  XML document, reading, 276–277
DataTable, 247, 255–256
  usage, 413–414
DataType object, 296
DB. *See* Second-level normalized DB
DB2, usage, 453
dbCmd object, 417, 422, 434
Debugging, 39
  situations, 45
Decimal data type, 97
Declaration, 88. *See also* Descriptive declarations
Declarative code,VB.NET syntax, 172
Declarative security, 42, 166, 172–173
Default rights, 193
Default tokens, 68
Delete
  form, 422–428
  method, 342
DeletePost method, 342, 344
DeleteThread method, 342–344
Demands, 44, 164, 191. *See also* Inheritance
  request, 183
Deny Override (command), 182–183
Deny (security action), 179
Deployment unit, 8
Depth, 234
Derived data type, 97
DES. *See* Data Encryption Standard
descendant::Account, 258
descendant::Name, 257
Description, 313
Descriptive declarations, 22
DESCryptoServiceProvider, 208
Design window, 59
Desktop application, 3
Deterministic finalization, 32
Development code, change, 2
different-tag-name, 105
DigestMethod, 220
DigestValue, 220
Digital certificates, 212
  verification, 219
Digital signatures, 7, 218. *See also* Extensible Markup Language
Disconnected access database, 409
Disconnected data sources, 413
DisplayNode(node as XmlNode), 245
DisplayRec() function, 419
Distributed applications, 166
DLL. *See* Dynamic Link Library
DNS. *See* Domain Name System
Dockable (setting), 75
Docking windows, 62–63

DOCTYPE declaration, 467
Document
  format. *See* Self-defined document format
  generation. *See* Extensible Markup Language documents
  interface, 116
  navigation. *See* Extensible Markup Language
    XPathDocument/XPathNavigator objects, usage, 261–264
  parsing. *See* Extensible Markup Language documents
Document Object Model (DOM), 242
  API capabilities, 112–113
  document. *See* Extensible Markup Language
  DOM-compliant System.Xml classes, 145
  exploration. *See* Extensible Markup Language DOM
  levels, 131. *See also* Extensible Markup Language DOM
  object, 509
  range, 122–123
  Ranges, 114
  recommendations. *See* World Wide Web Consortium
  specifications, 114
  Traversal, 114
  traversal, 118–122
  tree, 244, 247, 256
  XML parser, 116
  XPath, 123
Document Type Definition (DTD), 89, 93, 483
  display, 467
  maintenance, 457
  usage, 465, 482
Documentation generation, 73–75
DocumentElement, 244
  child nodes, 150
DocumentFragment, 122
DocumentRange interface, 123
Documents. *See* Extensible Markup Language
  root element, 92
Do.Loop structure, 180
DOM. *See* Document Object Model
Domain Name System (DNS), 233
Domains. *See* Application
dotBoard, 322
  construction, 373
dotBoardObjects, 334, 358
dotBoardObjects.User object, 360
dotBoardUI, 357, 358
Download cache, 11
Drop-down list, 391
DropDownList control, 390, 392
dRow, 435
DSA. *See* Data Signature Algorithm
DSACryptoServiceProvider, 208
DSN, creation, 443, 444
DTD. *See* Document Type Definition
dtSet object, 434, 435
Dynamic Help, 66–67
Dynamic Link Library (DLL), 4
  backward compatibility, 22
  breaking, 8
  file, 4, 10, 12, 25
    usage, 23
  problems, 22–24
Dynamic reference, 12

E

E-business, 458
E-communications. *See* Business-to-business
Economy JIT, 28
EDI. *See* Electronic Document Interchange
Electronic Document Interchange (EDI), 454–456
  communication, 461
  disadvantage, 456
Element, 89, 122, 237. *See also* Root element
  authentication, 440
  content, 465
  nesting, 92
  tag name, 113
  usage, 92
Element-type node, 91
e-mail, 285
  address, 313, 382
    identification. *See* Author
  validation, 294
e-mail element, nesting, 299
Embedded commenting. *See* Extensible Markup Language
Embedded XML tagging structure, 73
Emoticon element, 303
emp table, 417
empCmd, 424–428
emp_code, 415, 424
emp_firstname, 415
emp_lastname, 415
Employee code, 425–427
Employee Code field, 420
empReader, 424, 425
Empty element, 91
EnableAssemblyExecution (security permission), 193
Encoded data, reading. *See* Base64
Encrypted data element, 213, 216
Encrypted XML instances, 217
Encryption. *See* Extensible Markup Language
EncryptionMethod, 216
End Element, 237
EndEntity nodes, 237
End-tag, 92, 212
End-user security, 12
Enterprise
  security
    level, 170, 192
    policy, 200
  systems, 233
Enterprise Manager. *See* Structured Query Language
EntityReference, 122, 237
Entries, creation/editing/deletion, 151–155
Entry
  addition/viewing, 285
  element, 152
  list box, 153
  node, 123
  point, 8
    text areas, location, 285
EOF, 234
Equals() method, 440
ErrorMessage property, 368
Errors. *See* Partial error
  catching, 461
  contrast. *See* Fatal errors
  description, 26
  message, 198, 487
  trapping. *See* Data

Try/Catch method, 469
Event contract, 21
EveryThing (permission set), 193
Exception
class, 25
handling, 24–26
object, 24
ExecuteNonQuery method, 323, 428
ExecuteReader() method, 424, 500
Execution
engine, 30
permission set, 176, 193
Extensible Markup Language Schema Definition (XSD), 93–94, 124
Authority, 93
Extensible Markup Language (XML)
address book
construction, 145–155
loading, 145–149
code, 302
compatibility. *See* SGML
construction, 286–288
control. *See* ASP.NET
data, 99, 119, 135, 233, 250
querying, XPathDocument/XPathNavigator (usage), 256–264, 280
representation, class decisions, 138
design, 85
Designer, 87
digital signatures, 218–221
documentation
comments, addition, 74
file, 73
DOM documents, 106
Editor, 70–72
element, 91, 102, 140
embedded commenting, 73–75
encryption, 160, 212–218
FAQs, 109–110, 228–230
file, 14, 268, 274, 436–440
creation, 66
usage, 460–461
file, values (reading), 487–488
format, 268, 444, 461
fragment, 217, 508
fundamentals, 84
goals, 85
guestbook, functional design requirements, 285–288
interaction. *See* World Wide Web
markup, terseness, 85
overview, 84–92
parser, 114, 118. *See also* Document Object Model; Microsoft XML parser
parsing, 234–239, 279
processors, 455
reading, 234–239, 279
recordset, 514
schema. *See* World Wide Web Consortium
data types, 97–98
securing, practices, 212–221, 227
security, understanding, 160
solutions, 107–108, 223–227
string, 503. *See also* Session
tagging structure. *See* Embedded XML tagging structure
tags, 363
traditional databases, contrast, 453–454
transformation, XSLT usage, 98–105
tree, 234
usage, 273–277, 281, 360. *See also* Message board creation; .NET framework
risks. *See* .NET framework

validation. *See* Visual Studio.NET
vocabularies, 456
XML-code files, 201
XML-coded file, 199
XML-coded permission sets, 177, 195–197
Extensible Markup Language (XML) documents, 85–86, 102, 512. *See also* Well-formed XML documents
appearance, 85–86
comments, addition. *See* C#
components, 88–91
creation, 86–88. *See also* Visual Studio.NET
database query, usage, 274–276
fragment, 120
generation, XmlTextWriter (usage), 239–241
loading, 247
navigation, usage. *See* Data
object model, 112–123
parsing, 235–236
XmlDocument object, usage, 244–246
reading. *See* DataSet
structure, 91–92
transformation, 268–273. *See also* HyperText Markup Language document
XSLT, usage, 264–273, 280
validity, 93–98
viewing, 72
writing, XmlTextWriter class (usage), 239–241, 279
Extensible Markup Language (XML) DOM
core interfaces, 114
exploration, 242–256, 280
levels, 113–114
mapping. *See* System.Xml namespace
structure model, 115–118
Extensible Markup Language (XML) packages
acceptance, 482
design, 462, 465–490, 534
validation, 482–483
Extensible Stylesheet Language Transformation (XSLT), 56, 456
code, 269, 271
file, 269
plug-in, 69
style sheet, 101
transformations, 220
usage. *See* Extensible Markup Language; Extensible Markup Language document
Extensible Stylesheet Language (XSL). *See* Built-in XSL transformer
debugger. *See* HyperText Markup Language
debugging, 105
pattern usage, 102–105
style sheet, 102
External assembly, 21
references, 18
External class, 65

F

Family access, invoking, 22
Fatal errors, nonfatal errors (contrast), 469–470
Fields, 19, 20, 22
subvalues, 20
FileCodeGroup, 201, 202
FileIO, 197
FileIOPermission, 172, 177
addition, 197

granting, 176
permission, 183
FileMode.Open, 296
Files
locking, 297
uploading, 232
FileStream
class, 440
object, 296, 307
fillData method, 332
Finalization. *See* Deterministic finalization
FirstChild, 242
Floating (setting), 75
Flow layout, 65
Flush, 239
Foreach iteration, 144
foreach, usage. *See* XmlNodeList class
Formatting, 239
FormBase class, 360, 368
Forward-only data stream, 411
Foundation class libraries, 233
Foward-only cursor, 242
Framework
permission classes, 184
security, 43–47
Free store (freestore), 33
frmViewBitmap, 437
FtpChannel, 207
FullTrust (permission set), 176, 193

G

Garbage Collection (GC), 26, 33, 518
calling, 34
namespace, 37
usage. *See* Managed heap
GC. *See* Garbage Collection
Generations, assigning, 40–41
Generic principal, 164
GenericPrincipal
(command), 187–188
usage, 190
GET messages, 529
getAttribute, 465
GetBoards, 356
GetDataSet, 323
GetElementsByTagName method, 117, 145, 149
GetStyleName function, 365, 366
GetXML method, 273, 518
GetXmlSchema, 273
GIF file, 212
Global assembly cache, 11–12
usage, 16–17
Global pointers, 35
Granted permissions, 193
Grants, 164
Graphic drawing, 233
Graphical add-ons, 307
Graphical User Identification (GUID), 4
change, 23
inclusion, 21
Graphical User Interface (GUI), 59, 288, 490–513
GUI-based format, 447
Guest user, 327
Guestbook
creation. *See* XML.NET guestbook
entry page, 301
interface, advanced options, 301–307, 309
records, addition, 288–297, 308–309
viewing, 298–300, 309
Guests, message entry, 284
GUI. *See* Graphical User Interface
GUID. *See* Graphical User Identification

H

HACK, 68
Hackers. *See* Malicious hackers
HasAttributes property, 234
HasChildNodes, 242
Hash algorithm, 208, 209. *See also* One-way hash algorithm
Hash digest, 209–210
Hash value, 219
HasValue, 234
Header, announcements, 491
Heap. *See* Managed heap
  objects, placement, 34
  strain, 39
  unnecessary usage, 39
Help link, discovery, 301
HelpLink, 25, 26
Hide (setting), 75
Hierarchy, root, 170
HiveKey, 200
HKEY value, 198
HKEY_LOCAL_MACHINE, 198
Host, evidence, 168
href attribute, 299
HRESULTS, 4
HTML. *See* HyperText Markup Language
HtmlAnchor control, 396
HTTP. *See* HyperText Transfer Protocol
HttpChannel, 207
HTTPS, 212
HyperText Markup Language (HTML), 69, 84. *See also* Quasi-HTML; XHTML
  anchor tag, 400
  code, 292
  data transformation, 104
  document, 92, 99, 264–265
  XML document transformation, 266–267
  file, 12, 292
  form, 292
  format, 442
  functions, 298
  HTML-based XSL debugger, 105
  page, 441
  streams, 519
  string, 382
  table, 99, 100, 266
HyperText Transfer Protocol (HTTP), 84, 212, 457
  method, 482
  protocol, 441, 442, 519

I

IDE. *See* Integrated Development Environment
Identity
  manipulation, 188–190
  permissions, 163
if (statement), 382
IIS, access, 207
IIS server, 76
Images, 491. *See also* Catalog
  generation, 232
  manipulation, 301–305
Imperative security, 166, 172–173
Impersonate (method), usage, 189
Industry-related information, 161
InferXmlSchema, 273
inflate (method), 331, 332
Information. *See* Business-related information; Industry-related information
  storage. *See* Metadata
  transport methods, 454–455

Informational version, 23
InheritAct, 178
Inheritance
  capabilities, 233
  demand, 178
Initialization
  code, 305
  times, increase, 39
InitializeThreads method, 341–347
initMenuItems method, 148
In-memory data, 17
InnerException, 25, 26
InnerText, 242
  property, 243
Input/output (I/O). *See* Console I/O
Integer data type, 97
Integer variables, 487
Integrated Development Environment (IDE), 56, 77. *See also* Visual Basic; Visual Interdev IDE; Visual Studio.NET IDE
  customization, 68, 75
  view, 62
Integrity checks, 12
IntelliSense, 68–70
Interactive user, 189
Interdev user, 58
Interface
  contract, 20
  name, 18
  type, 19
  visibility, 18
Internal security. *See* .NET internal security
Internet
  developers, 482
  (permission set), 176, 193
Internet Explorer, 92, 99, 264–265
  version 5 (IE5), 468
Internet Protocol (IP) address, 444
Internet Services Manager, 357
Internet Zone, 204
Internet_Zone, 204, 206
Interoperability, 3, 27, 233, 409
  allowing, 21
Intranet zone, 171
Invoke button, 523
IP. *See* Internet Protocol
IPermission, 185
iRows, 427
IsAdmin property, 393
IsAnonymous (property), 192
IsAuthenticated, 186
IsBanned property, 393
IsDefault, 234
IsEmptyElement, 234
IsGuest (property), 192
IsInRole (method), 191
ISO C/C++, 3
IsSystem (property), 192
Item, 234
  function, 353
  property, 344, 346
Iterator object, 260, 262
IUnrestrictedPermission, 185

J

Java, 24, 65
  script, 494
JavaScript. *See* Client-side JavaScript
JIT. *See* Just-in-time
JOIN
  clause, 409
  usage, 414
jpeg format, 431

Jscript, 232
Jscript.NET, 5
Just-in-time (JIT), 42. *See also* Economy JIT; Normal JIT
  code, 6
  compilation, 22, 323
    phase, 165, 177
  compiler, 2, 27–28, 35

K

Kerberos, 192, 207
KeyPress, 428

L

LAN. *See* Local Area Network
Languages, usage. *See* .NET-compliant programming languages
Last in first out (LIFO), 25
Last known good system, 23
LastChild, 242
Layout paradigm, 490
Legacy-code platform, 42
LevelFinal, 206
Levels, number (limitation), 170
LIFO. *See* Last in first out
Link demand, 177
LinkButton, 394
loadAddressBook() method, 148
Loader optimization, 6
Local Area Network (LAN), 441
Local assembly members, 18
Local variables, 42
LocalIntranet (permission set), 176, 193, 195, 206
LocalIntranet_Zone group, 201, 205–207
Location
  option, 6, 10
  process, 16
Logged-in user, 360, 368, 372, 391, 402
Log-in interface, construction, 366–372, 404
LogonUser, 189
Loops. *See* Product

M

Machine
  policy, 46
  security
    level, 170, 192
    policy, 200
Machinewide code cache, 11
Main, 8
Maintainability, 409–410
Malicious code, 163, 169, 173
Malicious hackers, 455
Managed code, 4, 192. *See also* Unmanaged code
  usage, 193
Managed extensions, 5
Managed heap, 33–34
  garbage collection, usage, 35–38
Manifest, usage, 8–11
MD5. *See* Message Digest 5
MD5CryptoServiceProvider, 208
Members. *See* Local assembly members; Public members
  defining, 19–20
  profile, editing, 383–385
Membership conditions, 171
  usage, 170
Memory
  management, 32
  overhead, 10

resources, 6
Memory-intensive objects, 41
Message, 25, 26, 529. *See also* Errors; GET messages; POST messages
author comment, display, 284
browsing, 379–382
digest, 209–210
display, 284, 298–300, 427
entry. *See* Guests
property, 368
sending. *See* Success message
Message board creation
ADO/XML usage, 312
FAQs, 405–406
solutions, 403–405
database, setup, 312–321, 403
general functions, setup, 358–366, 404
Message Digest 5 (MD5), 208
MessageBoard class, 375
design, 356–357
Metadata, 4, 7
APIs, 18
benefits, 18
information, storage, 8
storage, 22
understanding, 17–24, 50
usage. *See* Assembly
MethodInfo object, 178
Methods, 20. *See also* Static methods; Virtual methods
contract, 20
Microsoft Intermediate Language (MSIL), 2, 27–28, 41, 51
code. *See* Portable Executable
transformation, 17
Microsoft Management Console (MMC), 202
snap-in, 192
Microsoft (MS) access database, 313–317
Microsoft XML (MSXML) parser, 113
Mini database, 409
MMC. *See* Microsoft Management Console
Moderator, 313, 326
ModeratorID, 313
location, 342
Modify User button, 390, 392
Modules
enumeration, 22
location, 22
Monitors, visible area, 301
MoveNext method, 262
MoveToAttribute(i) method, 234
MoveToContent(), 237–239
MoveToElement method, 234
msdata, 288
MSDN, 206
Help files, 67
Subscribers, 203
MSIL. *See* Microsoft Intermediate Language
MSXML. *See* Microsoft XML
Multi-assembly
scenario, 7
situations, 12
Multidomain host, 6
optimization route, 6
Multifile assemblies, creation, 12
Multiple tables, viewing. *See* XmlDataDocument
Multiple-table views, providing, 247
myThread variable, 338, 340

N

Name, 313
  collision, 7
  information, 285
Namespace, 112–114, 498. *See also* CodeAccess Permission namespace; Garbage Collection; System.Collections; System.Reflection namespace; System.Xml namespace
  system, usage, 28–29, 51
  usage, 160
name=value pair protocol, 89
Naming option, 6
Navigation
  paradigm, 490
  strip, 491
Nested elements, 252, 256
.NET applications, 2, 408
.NET code, 21
  access security model, 166–185
.NET Configuration Tool, usage, 201
.NET data provider. *See* Structured Query Language server
  usage, 410–412
.NET DOM, 132
.NET environment, 274
.NET Framework, 2–3, 193, 443
  classes, 17
  definition, 3, 49
  FAQs, 52–53
  permissions, 193
  security system, 208
  software libraries, 232
  solutions, 49–52
  XML usage, 112
    FAQs, 158
    risks, 160–162, 223–224
    solutions, 157
.NET internal security, 162–166
.NET languages, 232–233
.NET provider. *See* OLEDB .NET provider
.NET security, 41, 224
  FAQs, 228–230
  framework, 163
  solutions, 223–227
  understanding, 160
NetCodeGroup, 201, 202
.NET-compliant language, 5
.NET-compliant programming languages, usage, 5, 50
Netforce, 452
Network-handling functions, 233
nextNode(), 119, 120, 122
NodeFilter interface, 118, 121–122
NodeIterator, 118–121
NodeList, 117–118
nodeName property, 130
Nodes, 113. *See also* Product; Subnodes
  authentication, 440
  interface, 115
Nodes, values retrieval, 248–249
nodeValue property, 130
Nonfatal errors, contrast. *See* Fatal errors
Nonpublic information, 21
Nonpublic types, enumeration, 22
Nonwhitespace, 237
Non-x86 architecture, 27
Normal JIT, 28
Normal mode, 14
Normalization, 464
Nothing (permission set), 176, 194
NTLM, 192, 207
Null values, 415, 422

O

OASIS, 456, 457
obj.Control.GetType().ToString(), 365
Object Browser, 65
Object-oriented (OO) approach, 321–322
Object-oriented (OO) language, 323
Object-oriented (OO) objects, 326
Object-oriented (OO) technique, 323
Object-oriented programming (OOP), 29
Objects, 19. *See also* ASP.NET; Exception
  creation, 34
  design, 323
  destruction, 37
  hierarchy, 397
  holding, 29
  lifetime, increase, 39
  navigation. *See* XmlDocument
  orientation, 233
  placement. *See* Heap
  recreating/reinitializing, 41
  relational view, usage. *See* XmlDataDocument
  usage. *See* XPathDocument; XPathNavigator
oComm, 500
ODBC
  connections, 286
  usage, 440
OldDbCommand method, 424
OLE objects, 408
OLEDB. *See* SQLOLEDB
  usage, 412
OleDb classes, 482
oledb methods, 531
OLEDB .NET provider, 412
OleDbConnection class, 422, 432, 445
OleDbDataAdapter class, 417, 422, 434, 445–446
OLTP. *See* Online Transaction Processing
One-way hash algorithm, 209–210
Online Analytical Processing (OLAP), contrast. *See* Online Transaction Processing
Online forms, 296
Online Transaction Processing (OLTP), Online Analytical Processing (OLAP) contrast, 463
OnLoad, 290
On-the-fly call, 12
OO. *See* Object-oriented
OOP. *See* Object-oriented programming
Open function, usage, 411
Operator, usage, 34
Oracle, 445, 453
orderCount variable, 508
OuterXml, 137

P

Packages. *See* Extensible Markup Language packages
Page. *See* Real-world page; Shopping cart
  access. *See* Catalog
  design. *See* World Wide Web
  load code, 305
Page.Init event, 371
Page_Init subroutine, 371
Page_Load
  event, 237, 243
  method, 375, 385, 402
  sub, 307
Page.Load event, 368

Page_Load() event, 245
Page_Load subroutine, 368
pagename.aspx.cs, 69
Pages. *See* C#; Code-behind pages; User-created page
  output, modification, 305–307
Parent-child relationships, 93
Parents, 113
Parser, usage, 465
Parsing attributes, 465
Partial error, 470
Partial references, usage, 15
Passwords, 444, 530. *See also* User
  information, 469
Patterns, usage. *See* Extensible Stylesheet Language
PE. *See* Portable Executable
Performance, 409
Perl, 232
Permission sets, 170, 193, 206. *See also* Extensible Markup Language
  creation, 195–200
  determination, 194
  management/configuration, 166
  usage, 200
Permissions, 163–164. *See also* Code; Custom permissions; Granted permissions; Identity; .NET framework; Role-based security permissions; Security
  assembly request, 173
  assignation, 170
  classes. *See* Framework
  creation, 195
  demanding, 166, 177–179
  granting, 43–45, 170
  having, 180
  layer, 44
  list. *See* Assigned permission list
  needing, 22
  obtaining, 166
  releasing, 171
  requesting, 164, 166, 173–177
PermissionState, 191
PermitOnly Override, 183–184
PermitOnly (security action), 179
Persistence, snap-in type, 18
Pid, 91
Plain text, 408
Plaintext e-mail, 210
PName node, 91
pnlAdd panel, understanding, 292–293
PnlThank, usage. *See* Thank You panel
Pointer. *See* Global pointers; Static pointers
  creation, 33
  type, 19
Policy. *See* Security
  assemblies, 194
  checking. *See* Version
  level, 200, 206
Policyholders, 46
Portable Executable (PE)
  format, 5
  MSIL code, 8
  syntax, 27
Post, 313
  class, design, 353–355
  creation, 385–389
  deletion, 394
  editing, 399
POST messages, 529
PostID, 314
PostList class, design, 350–353
Precoding analysis, 458
previousNode(), 119
Price node, 91

Primitive data type, 97
Princauthenticated, 191
Principal, 43, 164, 186–190. *See also* Custom principal; Generic principal; Windows
  authentication, determination, 192
  usage. *See* Representation
PrincipalPermission, 186, 190–191
PrincState, 191
Print page, 489–490
Private assembly files, 17
Private fields, updating, 329
PrivatePath attribute, 14, 16
PrivatePermissions, 205
Probing, usage, 16
prod_code, 486
Prod_Code, status (checking), 486–487
Product
  loops, 487
  nodes, 91
  selection page, 243
  values, 487
Product Name data, 236
ProductID, 95, 98, 265
ProductName, 89, 98, 265
  node, 237
Programming
  elements, 22
  language, 99
    users. *See* Third-party programming language users
  tool, 113
Projects, 64, 76
  construction, 77
  creation, 76–77
  debugging, 77
Properties, 19, 22
  contract, 20
Properties Explorer, 63–64
Protected operations, access, 193
Protected resources, access, 166, 180, 193
Public access, invoking, 22
Public key
  algorithm, 209
  security services, 212
Public members, 22
Public types, 22
  enumeration, 22
Publishers certificate, 170

Q

QFE. *See* Quick Fix Engineering
Quasi-HTML, 373, 376, 379
Query Analyzer, 61. *See also* Structured Query Language
Query expressions, 257–258
Query string, 378
Quick Fix Engineering (QFE), 14, 24
  usage, 16
Quotes, 465

R

RAD. *See* Rapid Application Development
Random Number Generator (RNG), 208
Rapid Application Development (RAD), 63
RC1. *See* Release Candidate 1
RC2. *See* Rivest Cipher 2
RC2CryptoServiceProvider, 208
RDBMS. *See* Relational Database Management System
Read(), 237, 238
ReadIN (variable), 27

ReadState, 234
ReadString(), 238
ReadXml method, 273, 276
readXML object, 439–440
ReadXmlSchema, 273
Real-time validation, 285
Real-world page, 492
Records
  addition. *See* Guestbook
  form, addition, 419–422
  navigation, 415–422
  updating, 426
RecordSet, 413, 414
Recordset. *See* Extensible Markup Language
Recursive procedure, 245
Red-colored controls, 366
Reference. *See* Dynamic reference; External assembly references; Static reference; Strong reference
  addition, 77
  counting, 33
  locating, 12
  placement, 28
  scope boundary, 8
  types, 19
  usage. *See* Partial references; Weak references
Reflection, 21–22
  emit services, 18
  services, 18
ReflectionPermission, 21–22
Register page, 366–369, 383
Registered User, 326, 358, 382, 385. *See also* Unregistered user
Registry, 4
  keys, 180
  resource, 163
RegistryPermission, 172
  addition, 197
  demand, 177
Relational collection, 413
Relational data, 250
Relational DataBase Management System (RDBMS), 453
Relational table, 253
Release Candidate 1 (RC1), 144
Remote data access, advantages/disadvantages, 442–445
Remote database, definition, 441–442
Remote database viewer
  construction, 408
    FAQs, 449–450
    solutions, 448–449
  design, 440–447, 449
  implementation, 440, 445–447, 449
RemoveChild() method, 150
Repeater, 307
  code, 377
  control, 373–375, 379
Representation, gaining (principal, usage), 45–46
RequestMinimum, 173
RequestOptional, 174
RequestRefuse, 174
Requests, 164
  username, 333
REQUIRED attribute, 465, 466
RequiredFieldValidator, 293
Reset (command), usage, 200
Response.Redirect, 240
Restore Policy, usage, 200
Restrictive policy, 192
Reverse engineering, 93
Rivest Cipher 2 (RC2), 208

Rivest Shamir Adleman (RSA) algorithm, 209
RNG. *See* Random Number Generator
RNGCryptoServiceProvider, 208
Role-based security, 43–44, 162, 185–192, 226
  checks, 190–192
  permissions, 163
Role-based validations, 186, 187
Role-based verification, 189. *See also* Application
Root element, 89, 136. *See also* Documents
Root tags, 287
Rows collection, 375
RSA. *See* Rivest Shamir Adleman
RSACryptoServiceProvider, 209
Rules, built-in check system, 21
Runtime
  execution, 8
  security policy levels, 192
  usage, 33–34

S

Safe mode, 14
Sandboxing, 166. *See also* Code
Scalability, 409
Schema, 89, 93–98. *See also* Text-based schema
  data types. *See* Extensible Markup Language
  specification, 93
Schema Object Model (SOM), 125
SchemaProject folder, 126
Screen scrape, 232
SDK, 11
Second-level normalized DB, 464
Secure Hash Algorithm 1 (SHA1), 209, 210
Secure Sockets Layer (SSL), 160
  usage, 203
Security. *See* C#; Code Access Security; Declarative security; Extensible Markup Language; Framework; Imperative security; .NET internal security
  boundary, 8
  characteristics, 22
  checks, overriding, 167, 179–184
  issues, 207
  levels, 194
  model. *See* .NET code
  permission, 18. *See also* Execution; Role-based security; SkipVerification
  policy, 43, 46–47, 165, 192–207, 226
    application, 166
    level, 204. *See also* Runtime
  profile, 173
  remoting, 207
  services, 41–47, 52
  system. *See* .NET framework
  tools, 210–212, 227
  vulnerabilities, 163, 164
security.config, 200
security.config.cch, 200
security.config.old, 200
SecurityPermission
  permission, 187
  properties, modification, 197
SelectedIndex property, 244
SelectedItem, 392
Selection page. *See* Product
SelectSingleNode function, 365
Self-defined data, 84
Self-defined document format, 84

Self-describing application, 23
Self-describing value, 19
Self-description, 27
Server database. *See* Structured Query Language
Server Explorer, 60–61
Server.MapPath(), usage, 296
Service account, 189
Serviceable page, 491
Session
  key, 209
  objects. *See* ASP.NET
  userId value, 371
  variable, 502, 508
  XML string, 511
SetAttributeNode method, 141
Sets. *See* Permission sets
SetXmlFileName property, 437
SGML
  creation, 84
  XML compatibility, 85
SHA1. *See* Secure Hash Algorithm 1
SHA1CryptoServiceProvider, 209
ShadowCopy attributes, 14
Shared assembly files, 17
Shared names, 7
Shell host, 46
Shopcartadd.aspx, code listing (analysis), 508–513
Shopping cart
  articles, loading, 491
  content, 490
  page, 502–513
ShowTransformed() subprocedure, 271
Side-by-side deployment, 23
Side-by-side execution, 8
SignatureMethod element, 220
SignedInfo element, 220
Simple Object Access Protocol (SOAP), 519
  protocol, 529
simpleType data structures, 94
Single domain, 6
Skeleton class, 31
SkipVerification (permission set), 176, 193, 194
Snap-in type. *See* Persistence
SOAP. *See* Simple Object Access Protocol
Solution Explorer, 64–65
Solutions, 64
SOM. *See* Schema Object Model
Source Safe, usage, 67
SQL. *See* Structured Query Language
SQLOLEDB, 412
SSL. *See* Secure Sockets Layer
Stack slots, 42
Stack walking, 166–168, 181
StackTrace, 25–26
Start-tag, 89, 92, 212
Static assembly, 6
Static methods, 20
Static pointers, 35
Static reference, 12
Storage, 453–454
streamFile object, 440
Strings
  construction, 382
  description, 23
Strong reference, 41
Structured Query Language (SQL), 256
  database, 286
  Enterprise Manager, 320, 443
  query, 273, 411, 422, 426–427
    deletion, 428
  Query Analyzer, 317, 443

script, 317, 325
statements, 138, 313, 323, 333, 391
construction, 342, 346
generation, 335, 339
usage, 417, 499
strings, 486, 531
construction, 488
Structured Query Language (SQL) Server, 312, 445, 453, 457
conversion, 531–532
database, 285, 317–321
migration, 529–532
.NET data provider, 412
version 7.0, 464
Northwind database, 274
version 2000, 443
Style sheet, 266. *See also* Extensible Stylesheet Language; Extensible Stylesheet Language Transformation
Subject line, 285
Submit button
code, 294
handler code, 294–298
Subnodes, 113
Subroutine, exiting, 487
Success message, sending, 488–489
Supplier
interface, 462, 469–490
user identification/password, checking, 483–485
Symmetric key algorithm, 208, 209
System
account, 189
architecture. *See* Business-to-business
files, 180
object, 2
services, usage, 24–27, 50–51
System.Collections namespace, 338
System.Collections.IEnumerable, 142
System.Data namespace, 286, 288
System.Object, 142
System.Reflection namespace, 21
System.Security.Cryptography, 208
X509 certificates, 208
System.Security.Cryptography.Xml, 208
System.Web.UI.Page, 359
System.Web.UI.WebControls, 365
System.Xml classes. *See* Document Object Model
selection, 132–145
usage, 122
System.Xml namespace, 124–145, 286
usage, 145–155
XML DOM, mapping, 130–132
System.Xml.Schema classes, 124–129
System.Xml.Schema namespace, 127

T

Table collection, 413, 414
Tag name, 113. *See also* Attributes; Elements; User-given tag names
tag-name, 105
targetNamespace attribute, 288
Task List explorer, 67–68
TCP/IP. *See* Transmission Control Protocol/Internet Protocol
Template-based declarative language, 98
template-name, 105
Termination control, 39
Testing situations, 45
Text
areas, location. *See* Entry
boxes
control, 402

validation, 154
data type, 530
file, 85
Text-based approach, 286
Text-based schema, 125
Text-type, 91
Thank You page, 512
Thank You panel (addition), PnlThank (usage), 294
Third-party application, 524
Third-party programming language users, 5
Thread
browsing, 376–379
class, 353
design, 347–350
creation, 385–389
ID, 313
object, 347
panel, 387
ThreadID, 314, 316, 336–338, 388
field, 389
ThreadList class, 350, 352
design, 344–347
TIBCO. *See* XML Authority
TODO, 68
Token. *See* Custom token; Default tokens
property, 192
Toolbox, 61–62
Top-down ASP scripts, 321
Top-level root, 116
Transaction, beginning, 486
Transform method, 271
Transform() method, 98
TransformSource attribute, 267
Translators, 455
Transmission Control Protocol/Internet Protocol (TCP/IP), 233
Transport protocols, 457
TravelDownATree(tree as XmlNode), 245
Tree-oriented view, 120
TreeWalker, 118, 120–121
Triple DES (3DES), 209
TripleDESCryptoServiceProvider, 209
try block, 422, 424
Try/Catch
method. *See* Errors
segment, 296
set, 25
statement, 26, 483
support, 462
system, 24
usage, 509
txtEmpCode, 422
txtSalary, 428
Types, 19–22, 529. *See also* Public types
boundary, 8
conversions. *See* Variable type conversions
enumerating. *See* Nonpublic types
exportation, 18
safety, 32, 165–166
values, checking, 488
Type-safe code, 32

U

UDA. *See* Universal data access
UDDI. *See* Universal Description Discovery and Integration
UDL. *See* Universal Data Link
UID, 486
UIPermission, 167, 180, 183
UML. *See* Unified Modeling Language

UNC names, 198
UNDONE, 68
Unified Modeling Language (UML), 323
  diagram, 336, 351
Uniform Resource Locator (URL)
  codebase, 16
  format, 488
Unique Identifier, 313
Universal data access (UDA), 408
Universal Data Link (UDL), 61
Universal Description Discovery and Integration (UDDI), 524
Unmanaged assembly code, 21
Unmanaged code, 4, 180, 189
  interop, 42
Unmanaged COM, 21
Unregistered user, 382
Update
  form, 422–428
  method, 329, 335
Update(), 326
updatecat1.aspx
  code listing, analysis, 482–490
  coding, 470–481
URI, 522
URL. *See* Uniform Resource Locator
URN, 26
User. *See* Banned user
  account, usage, 189
  class, 336–342
    design, 325–333
  code group, 194
  constructor, 331
  Control, 366
  functions, creation, 382–389, 405
  ID, 331–332, 360, 372, 516, 530
    obtaining, 383, 392
  identification/password, checking. *See* Supplier
  name, 191
  object, 326–327, 368, 383
    password, 385
  security level, 170, 193
  table, 331, 352, 393
User interface (UI), 290, 312, 321, 399
  design, 357–358
  determination, 404
userArea.ascx control, 368
UserControls, 493, 517
User-created page, 75
User-defined data types, 97
User-given tag names, 92
UserID, 316, 331
userId value. *See also* Session
User Interface processing, 409
UserLatestBuildVersion, 14
Username, 383, 424, 444. *See also* Requests
  field, 332
UTF-8 encoded string, 217

V

Validate method, 329–330, 333
Validation. *See* Real-time validation
  expression, 293
ValidationSummary, 366, 383
Value, 20, 234
  property, 234
Value-added network (VAN), 454, 456
VAN. *See* Value-added network
Variable type conversions, 233
VB. *See* Visual Basic
Vendors, 498
Verification. *See* Role-based verification

Version
  boundary, 8
  policy, checking, 12
Versioning
  constraints, 17
  support, 23–24
VersionNew, 14
VES. *See* Virtual Execution System
View/Source header, 523
Virtual database, 484
Virtual Execution System (VES), 30, 42
Virtual methods, 20
Virtual Private Network (VPN), 160
Virtual Private Networking (VPN), 455
Visual Basic (VB), 162
  application. *See* Command-line VB application
  COM objects, 322
  IDE, 63
  Profile, 334
  recursive procedure, 244
  VB.NET, 5, 56, 69, 99. *See also* Declarative code
    class, 325
  VBScript, 232
Visual Interdev IDE, 63
Visual Studio.NET IDE, 56
  FAQs, 81
  solutions, 79–80
Visual Studio.NET (VS.NET), 56–58, 86, 251–254
  components, 58–68
  features, 68–75
  usage, 250
  XML Designer, XML document creation, 87–88
  XML validation, 96
Vocabularies, 457. *See* Extensible Markup Language
VPN. *See* Virtual Private Network; Virtual Private Networking
VS.NET. *See* Visual Studio.NET

W

W3C. *See* World Wide Web Consortium
Weak references, usage, 41
Web Service Description Language (WSDL), 524–529
  file, 77
WebControl, 366
WebService class, 521
Well-formed document, 92
Well-formed XML documents, 92–98
Whitespace node, 238
Wholesale catalog
  construction, 452–453
    FAQs, 535–536
    solutions, 533–535
  design considerations, 453–462, 533
WindowIdentity object, 189, 190
WindowIdentity properties, 192
Windows
  2000, 207
    domain, 165
    environment, 164
    servers, 209
  directory, 16
  NT
    environment, 164
    systems, 23
  platform, 189
  principal, 164
  token, 189
  user, 164, 186

WindowsIdentity object, 187, 189
WindowsImpersonationContext (command), 189
WindowsPrincipal (command), 186–187, 191
  code, impersonation, 189
WindowsPrincipal.Identity .IsAuthenticated, 192
WindowsPrincipal.Identity.Name (value), 191
WinMain, 8
Wizard_Machine_Policy (code group), 200–201
World Wide Web Consortium (W3C), 84, 98, 105, 218, 264
  DOM recommendations, 130
  recommendations, 88, 90, 112–113
  software development, 123
  specification, finalization, 114
  standards, 130
  XMLSchema, 70
  XPath 1.0 recommendation, 256
World Wide Web (WWW / Web)
  applications, 104, 232
  development, 233, 288
  page, 516
    design, 494
  server, 212, 441
  Web-based applications, 409
  Web-related classes, 28
  Web-usable application, 3
  XML interaction, 232
    FAQs, 281–282
    solutions, 278–281
World Wide Web (WWW / Web) services, 56, 76
  coding, 519–521
  interaction. *See* Business
  overview, 519
  testing, 522–524
  usage, 524–529
WriteAttributes, 239
WriteAttributeString, 239
WriteComment, 239
WriteElementString, 239
WriteEndAttribute, 239
WriteEndDocument, 239
WriteStartDocument, 239
WriteStartDocument() method, 435
WriteState, 239
WriteXml method, 273, 275
WriteXML class, 297
WriteXmlSchema, 273
  method, 275
WSDL. *See* Web Service Description Language

X

X509 certificates. *See* System.Security.Cryptography
XHTML, 493, 494
XLink, 218
XML. *See* Extensible Markup Language
XML Authority (TIBCO), 96
XmlAddressBook folder, 145, 151
XmlAttribute, 140, 149
XmlAttributeElementProject folder, 140
XmlConfigFile, 361
XmlDataDocument, 136–138, 144
  class, usage, 247–256
  creation, 137
  DataSet, 247
  object, 247
    multiple tables, viewing, 252–256
    relational view (usage), 249–252
  usage, 145

XmlDataDocumentProject folder, 136
XmlDocument
  class, 133–134, 234, 485
  creation, 141
  loading, 248–249
  object, 365
    navigation, 243–244
    usage. *See* Extensible Markup Language documents
XmlDocumentProject, 135
XmlElement
  class, 139
  creation, 149
XML.NET guestbook
  creation, 284–285
    FAQs, 310
    solutions, 308–309
  functional design requirements, 308
XmlNode, 139
  base class, 133
  class, 132–134, 485
XmlNodeList, 144
  class, 485
  collection, 248
XmlNodeList class, 142
  foreach usage, 144
xmlns attribute, 288
XmlSchemaComplexType, 125
XmlSchemaObject class, 125
XmlTextReader, 138, 234, 239, 439
  object, 235, 236, 237, 243
XmlTextWriter, 435
  class, 234
    usage. *See* Extensible Markup Language documents
  object, 239–240
xmlWrite object, 435
XMLWriteMode.WriteSchema, 297
XPath, 105–106. *See also* Document Object Model
  expression, 153, 257
  queries, 257
    string, 365
XPathDocument
  objects, usage, 259–260. *See also* Document
  usage. *See* Extensible Markup Language
XPathExpression, 123
XPathIterator object, 248
XPathNavigator
  objects, 256
    usage, 259–260. *See also* Document
  usage. *See* Extensible Markup Language
XPathNodeIterator, 258, 259
XPathResult interface, 123
XPathSetIterator, 123
XPointer, 218
XSD. *See* Extensible Markup Language Schema Definition
XSL. *See* Extensible Stylesheet Language
XSLT. *See* Extensible Stylesheet Language Transformation
XSLTransform class, 98, 264
XSLTransform object, 271

Z

_Zone (ending), 200
Zone evidence, 201
Zone-based code groups, 201
Zoned evidence, 201

SYNGRESS SOLUTIONS...

AVAILABLE NOW
ORDER at
www.syngress.com

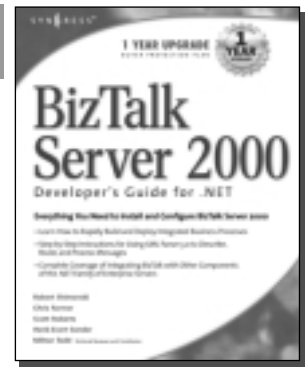
.NET Mobile Web Developer's Guide

The *.NET Mobile Web Developer's Guide* provides a solid foundation for developing mobile applications using Microsoft technologies. With a focus on using ASP .NET and the .NET Mobile Internet Toolkit, this book will give you the insight to use Microsoft technologies for developing mobile applications. It will also show how to avoid having to customize the output of your application.

ISBN: 1-928994-56-3

Price: $49.95 USA, $77.95 CAN

AVAILABLE NOW
ORDER at
www.syngress.com



BizTalk Server 2000 Developer's Guide for .NET

BizTalk Server 2000 is part of the .NET family of Enterprise Servers designed to work together to provide e-business solutions. The .NET Enterprise Servers are based on open Web standards, such as XML, to allow an organization to integrate and orchestrate their applications and service needs into a single comprehensive solution. This book shows how to use BizTalk Server 2000 to create, integrate, manage, and automate business processes for the exchange of business documents.

ISBN: 1-928994-40-7

Price: $49.95 USA, $77.95 CAN

AVAILABLE NOW!
ORDER at
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

solutions@syngress.com

SYNGRESS®