

Companion
eBook Available

THE EXPERT'S VOICE® IN SQL SERVER

Pro ASP.NET for SQL Server

High Performance Data Access
for Web Developers

Mastering the bridge between ASP.NET and SQL Server

Brennan Stehling

Apress®

spine = 0.82" 432 page count

Pro ASP.NET for SQL Server

High Performance Data Access
for Web Developers



Brennan Stehling

Pro ASP.NET for SQL Server: High Performance Data Access for Web Developers

Copyright © 2007 by Brennan Stehling

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-59059-860-3

ISBN-10 (pbk): 1-59059-860-1

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Lead Editors: Jonathan Gennick, Jim Huddleston

Technical Reviewer: Vidya Vrat Agarwal

Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jason Gilmore,
Jonathan Hassell, Matthew Moodie, Jeffrey Pepper, Ben Renow-Clarke, Dominic Shakeshaft,
Matt Wade, Tom Welsh

Project Manager: Kylie Johnston

Copy Editors: Sharon Wilkey, Ami Knox

Assistant Production Director: Kari Brooks-Copony

Production Editor: Ellie Fountain

Compositor: Diana Van Winkle, Van Winkle Design

Proofreader: April Eddy

Indexer: Carol Burbo

Cover Designer: Kurt Krames

Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail orders-ny@springer-sbm.com, or visit <http://www.springeronline.com>.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit <http://www.apress.com>.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

The source code for this book is available to readers at <http://www.apress.com> in the Source Code/Download section.

Contents at a Glance

About the Author	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii
CHAPTER 1 Getting Started	1
CHAPTER 2 Data Model Choices	37
CHAPTER 3 Database Management	55
CHAPTER 4 Databound Controls	73
CHAPTER 5 SQL Providers	111
CHAPTER 6 Caching	147
CHAPTER 7 Manual Data Access Layer	191
CHAPTER 8 Generated Data Access Layer	233
CHAPTER 9 Deployment	255
CHAPTER 10 A Sample Application	289
APPENDIX Photo Album	343
INDEX	399

Contents

About the Author	xi
About the Technical Reviewer	xiii
Acknowledgments	xv
Introduction	xvii

CHAPTER 1	Getting Started	1
	Preparing Your Environment	1
	Project Organization	2
	Common Folders	4
	Datasource Configuration	5
	Code and Database Separation	7
	Managing Provider Services	7
	Using the Command Line	8
	Mixing and Matching Providers	11
	Configuring Providers	11
	Membership Configuration	13
	Roles Configuration	14
	Profile Configuration	15
	Creating Users and Roles	18
	Securing the Admin Section	34
	Creating the Admin User	35
	Summary	36
CHAPTER 2	Data Model Choices	37
	The Data Access Application Block	37
	Data Access Code Snippets	39
	Sample Database	46
	Trivial Data Examples	47
	Nontrivial Data Examples	49
	Typed DataSet	49
	Nontyped DataSet	51
	DataReader	53

DataObject	53
What's the Downside?	54
Summary	54

■ CHAPTER 3 Database Management

Using Database Projects	55
Visual Studio	56
SQL Server Management Studio	57
Managing Stored Procedures	57
Managing Indexes and Constraints	62
Performance Considerations	65
Stability Considerations	70
Unit Tests for Data	70
Continuous Integration	71
Summary	71

■ CHAPTER 4 Databound Controls

DetailsView	73
FormView	74
GridView	76
Editing and Validating Fields	77
Binding Input Parameters	81
Binding Input Parameters with a Control	81
Binding Input Parameters Programmatically	82
Binding a User Control	83
Embedding Databound Controls	84
ViewState and Databinding	87
Session and ViewState	87
Paging	88
Disabling ViewState	89
ControlState vs. ViewState	89
Creating a Databound Control	91
Getting the Data	94
Getting the Total Rows Count	97
Wiring the Pager Events	98
Creating PersonRow	101
Persisting ViewState Manually	105
Working Without ViewState	107
Walking the Debugger	109
Summary	110

CHAPTER 5	SQL Providers	111
	The SqlMembershipProvider	112
	Using XML Implementations	113
	Setting the Database Connection	113
	Creating a Password Policy	114
	The SqlRoleProvider	115
	Controlling Access by Role	115
	Controlling Behavior by Role	117
	The SqlProfileProvider	118
	Why Anonymous Profiles?	119
	Configuring Anonymous Profiles	119
	Managing Anonymous Profiles	120
	Anonymous and Authenticated Profile Differences	121
	Migrating from Anonymous to Authenticated	121
	Creating a User	122
	Dynamic Profiles and Profiles as BLOBs	124
	Using the Provider-Powered ASP.NET Controls	126
	Building a SQL Photo Album Provider	129
	Provider Requirements	130
	Configuration Section Class	130
	Provider Collection Class	131
	Abstract Provider Class	132
	The Provider Implementation	134
	Provider Service Class	135
	Unit Testing	137
	The Finished Product	138
	Building a SQL SiteMap Provider	139
	SiteMap Requirements	140
	Implementing SiteMapProvider	140
	Summary	145
CHAPTER 6	Caching	147
	Alternatives to Caching	148
	Application State	149
	Session	150
	ViewState	150
	Current Context	150
	Caching Options	153
	Output Caching	153
	Data Caching	159

Invalidating Cached Data	164
Absolute Expiration	164
Sliding Expiration	164
Cache Dependency	165
Manual Removal	165
SQL Cache Dependencies	165
Using the SqlDependency and SqlCacheDependency	165
Polling	168
Query Notifications	172
Troubleshooting Query Notifications	176
Problems with Caching	186
Performance Strategies	187
Data Warehousing	187
Lazy Loading	188
Summary	189
 CHAPTER 7 Manual Data Access Layer	 191
Using DataSets, Inline SQL, and Stored Procedures	191
DataSets	192
Inline SQL	196
Stored Procedures	198
Using DataObjects and the ObjectDataSource	199
Design Contract	200
Data Contract	201
Testing the Design and Data Contracts	203
Building the Database	203
Creating the Database Structure	203
Consolidating the Data	204
Managing Relationships	204
Created and Modified	205
What About Nulls?	206
Using Database Projects	208
The Data Access Layer	217
Building the Website	225
Connecting the Data Access Layer	225
Creating User Controls	226
Creating the Pages	229
Summary	231

CHAPTER 8	Generated Data Access Layer	233
	Code Generation	233
	Build Providers	234
	CodeDom Namespace	237
	Templating	241
	SubSonic	242
	SubSonic Templating	243
	Partial Classes	245
	Query Tool	247
	Scaffolding	248
	Blinq	249
	Summary	253
CHAPTER 9	Deployment	255
	Automation with MSBuild	256
	Deploying the Website	261
	Website Deployment Projects	261
	Automating Configuration Changes	262
	PostBuild Deployments	264
	Deploying the Database	271
	Generating Change Scripts	272
	Automating Database Updates	274
	Custom Configuration Sections	285
	Summary	288
CHAPTER 10	A Sample Application	289
	Understanding Performance and Scalability	289
	Concurrent Requests	290
	Bottlenecks	291
	Traffic Spikes	291
	Distributing Traffic	293
	Distributing Content	294
	Distributing Services	295
	Distributing the Back End	296
	Planning for Scalability	298
	The Sample Application	298
	Creating the Database	299
	Get, Save, and Delete	300

Creating Data Access Providers	303
EventProvider Object	303
Revised DomainObject	307
Managing Relationships	310
Using Locations	311
Custom Configuration	314
Configuration Grouping	315
Declaring the Custom Configuration	317
Configuring the Providers	318
Creating New Providers	321
Implementing a LINQ Provider	321
Implementing a WCF Provider	332
WCF Service Requirements	333
Hosting the Service	335
Defining the DataContracts	336
Configuring the Provider	336
Using the Providers	337
Summary	341

■ APPENDIX	Photo Album	343
	Photo Album Provider	343
	Configuration	343
	Classes	344
	Table Scripts	371
	Constraints Scripts	372
	Stored Procedure Scripts	373
	Website Classes	379
	SQL SiteMap Provider	385
	Classes	385
	Table Scripts	394
	Stored Procedure Scripts	394

■ INDEX	399
----------------------	------------

About the Author

■ **BRENNAN STEHLING** is a developer who has a long background in web development. He created SmallSharpTools.com, which is a collection of open source C# projects designed to augment the .NET Framework with small components that are extensible and interoperable. He is also a member of the Wisconsin .NET User Group executive committee.

About the Technical Reviewer



■ **VIDYA VRAT AGARWAL** is a Microsoft .NET purist and an MCPD, MCTS, MCT, MCSD.NET, MCAD.NET, and MCSD. He works with Lionbridge (NASDAQ: LIOX), and his business card reads Subject Matter Expert (SME). He is also a lifetime member of the Computer Society of India (CSI). He started working on Microsoft .NET with its beta release. He has been involved in software development, evangelism, consultation, corporate training, and T3 programs on Microsoft .NET for various employers and corporate clients.

He lives with his beloved wife, Rupali, and lovely daughter, Vamika (“Pearly”). He believes that nothing will turn into a reality without them. He is the follower of the concept “No Pain, No Gain” and believes that his wife is his greatest strength. He is a bibliophile; when he is not working on technical stuff, he likes to play with his one-and-a-half-year-old daughter and also likes reading spiritual and occult science books. He blogs at <http://dotnetpassion.blogspot.com>.

Acknowledgments

I would like to thank everyone who made this book possible. A great deal of work went into creating every detail, and I received a lot of support and encouragement from the Apress staff, which includes Kylie Johnston, Jonathan Gennick, Sharon Wilkey, Ami Knox, and Ellie Fountain. Each of them has impressed me. I am glad I wrote this book with Apress.

I also want to thank others who helped me work through technical details. Julie Lerman helped me through some strange technical problems and also deserves credit for being such a great supporter and a speaker for INETA. The development community could use more developers like her.

Introduction

This book covers the middle ground between ASP.NET and SQL Server that is not covered sufficiently by books that focus on these two big pieces independently. I wrote this book to focus on the bridge between these two systems to reveal all of the techniques and features available to developers so that you could learn to fully leverage these two technologies, which were designed very cleverly to work together. As you read the book and work through the sample projects, my hope is that you will discover features that you did not know about previously that will help you make your websites faster and easier to maintain.

Who This Book Is For

This book is for the developer who wants to dig deeper into what can be done with ASP.NET and SQL Server. If you ever wanted to know more about how databound controls worked with the `ObjectDataSource` or to build your own provider model implementations, this book is for you.

How This Book Is Structured

This book is made up of the following ten chapters with a sample project for each chapter:

Chapter 1: Getting Started

This chapter starts with preparing the development environment that will be used throughout the book. Then it moves into managing and configuring the provider services and finally creating users and roles.

Chapter 2: Data Model Choices

There are many ways to access the database from ASP.NET, so this chapter looks into those choices and explains how to decide what to use in different situations as well as the reasons why.

Chapter 3: Database Management

Many ASP.NET developers do not leverage the tools that are available in Visual Studio that make it very easy to manage the tables, stored procedures, and other resources that are in the database. This chapter walks through how to use a Database Project to manage the scripts to create tables and stored procedures as well as manage the indexes and constraints. Finally, it covers how you can test changes to the database regularly to ensure your application and database stay in sync.

Chapter 4: Databound Controls

ASP.NET has a broad collection of databound controls, and this chapter goes over the ones that you will use the most and then digs into techniques you can employ to minimize the amount of code that you would place in the code-behind files. Then it covers how to create a databound control from the ground up and walks through how every part works including the use of `ViewState` and `ControlState`.

Chapter 5: SQL Providers

The provider model is very useful, and this chapter covers the three most used providers. It then covers the creation of a completely custom SQL Photo Album provider that works with a custom SQL SiteMap provider implementation.

Chapter 6: Caching

Improving the speed of an application can be done using caching, but there are so many ways this can be done. ASP.NET does include a caching mechanism, which is covered extensively, but there are also other simpler techniques that you can leverage that can also boost performance. All of these techniques are explained.

Chapter 7: Manual Data Access Layer

A manually constructed data access layer gives you the most control and greatest flexibility, and this chapter goes through everything that you can do to produce a fully functioning website built on top of a finely tuned data access layer.

Chapter 8: Generated Data Access Layer

Instead of manually building the data access layer, you can choose to generate some or all of the data access code using various utilities that are available. This chapter covers how to use code generation to automatically create working software and then explores two powerful utilities that generate complete data access layers for your website.

Chapter 9: Deployment

Once your website is built, you will need to deploy it, and this can often be the hardest part of the job. Fortunately, there are ways to make it the easiest task. This chapter covers how to use MSBuild to automate the build and deployment of a website including the database changes.

Chapter 10: A Sample Application

The final chapter puts it all together with a sample application that starts out with an explanation of performance and scalability and then shows how all of the concerns about speed are addressed by building a highly flexible website that can adapt to changing needs.

Prerequisites

To work with the sample projects for this book, you will need Visual Studio 2005 and SQL Server 2005 for the majority of the chapters. Chapters 8 and 10 make use of the .NET Framework 3.5, so you will need either the LINQ CTP or Visual Studio 2008.

Downloading the Code

The source code for this book is available to readers at <http://www.apress.com> in the Downloads section of this book's home page as well as from <http://SmallSharpTools.com/Apress>.

Contacting the Author

You can contact the book's author, Brennan Stehling, at brennan@smallsharptools.com or access his blog at <http://brennan.offwhite.net/blog/>.



Getting Started

Performance is always an issue. This book will show you how to optimize ASP.NET 2.0 application access to SQL Server databases. You can leverage the close integration of ASP.NET 2.0 and SQL Server to achieve levels of performance not possible with other technologies. You'll investigate in detail the middle ground between ASP.NET 2.0 and SQL Server and how to exploit it.

This book demonstrates all concepts with professional code, so the first thing you need to do is set up your development environment. I'll cover related issues along the way.

This chapter covers the following:

- Preparing your environment
- Managing provider services
- Configuring providers
- Creating users and roles

Preparing Your Environment

I typically run several Microsoft Virtual PC environments and move between them as my needs change. When I first set up my initial virtual environments, I create as many as I need. One will be my main development environment. This has been helpful as I keep a backup image of the initial environment immediately after I create it. As I try out beta releases or third-party add-ons, I may begin to dirty up my environment. Because the uninstall process may not completely clean up Windows installations, the virtual environments come in handy.

For example, at a local user group session, a presenter was demonstrating a tool that integrated with Visual Studio. Instead of installing the tool in my current development environment, I cloned a fresh environment and used it to try the tool. Afterward, I turned off the environment and deleted it to clear up space for another environment. My primary environment wasn't affected at all.

I also create two drives within my virtual environment. The C: drive holds the operating system, while the D: drive holds my data. These drives are represented by virtual hard drives. And if I decide to clone my data drive, I can use it with another system as I see it. Because I typically develop every project by using source control, I can start with a fresh environment, pull down the current project versions, and start development again.

To further leverage virtualization, you can also create a virtual hard-drive installation of your development environment and set the files to read-only. Then you can create a new virtual image that uses the read-only copy as a base image while keeping all changes on the

secondary image. When you are given the option to create a new virtual hard drive, you can choose the Differencing option, as shown in Figure 1-1.

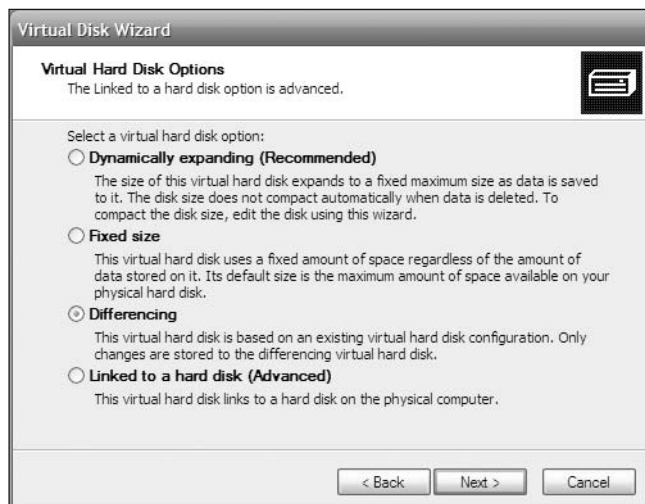


Figure 1-1. *Creating a differencing virtual hard disk*

The differencing image allows you to leave your base image untouched as you begin to use a fresh environment for development. This conserves space on a hard drive, which can fill up quickly; a typical virtual image can grow to 20 GB, not including the data image. And because you can have multiple virtual images using the same base image, your environment can quickly be prepared for your work as you need it.

One time my coworker needed to start work on a short .NET 1.1 project the next day. She had only Visual Studio 2005 installed and so was facing a long install process that typically takes several hours to complete. But because I had already prepared virtual images for .NET 1.1 and .NET 2.0 development, I had her ready to go in under 20 minutes. The longest wait was for copying the image off the file server. After she was finished with the short project, she was able to remove the temporary environment.

Having access to these prebuilt environments has been a major time-saver. I can experiment with tools and techniques that I would not have a chance to try if I had to cope with the consequences of having alpha- and beta-release software mixed in with my main development environment.

Project Organization

For each environment, I place all of my projects into `D:\Projects`. And I explicitly have the solutions at the root of each project. I have seen how some teams create solution files only as a side effect of creating projects, but that undercuts the advantages you get when you properly manage your projects within a solution.

For a typical web project, I start with the ASP.NET website in a blank solution. I then add a class library and call it `ClassLibrary`. I put as much of the code for the website into this class library, for reasons I'll cover later. Then I associate the class library to the website as a reference, which the solution records as a dependency. This is quite helpful, as the new ASP.NET 2.0

website project model does not include a project file that maintains a manifest for files and dependencies. I add a database project called Database, which holds all of my database scripts for creating tables and stored procedures, and scripts to prepare the database with supporting data. (Database projects require Visual Studio 2005 Professional Edition.) Finally, I create a solution folder called Solution Items and add a text file named README.txt that provides the basic information for the project, such as the name, description, requirements, dependencies, and build and deployment instructions.

The result of all this is the solution structure shown in Figure 1-2.

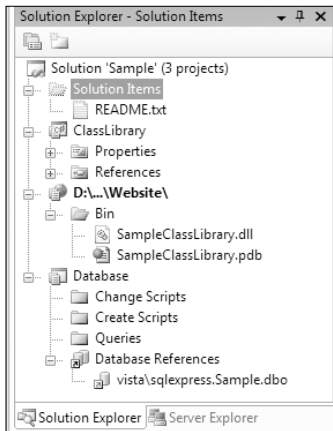


Figure 1-2. *Typical solution environment*

When you first set up a blank solution and add your first project or website to it, you may find that the solution goes away. This is a default setting for Visual Studio, which you can change. From the Tools menu, choose Options and then click the Projects and Solutions item, as shown in Figure 1-3.

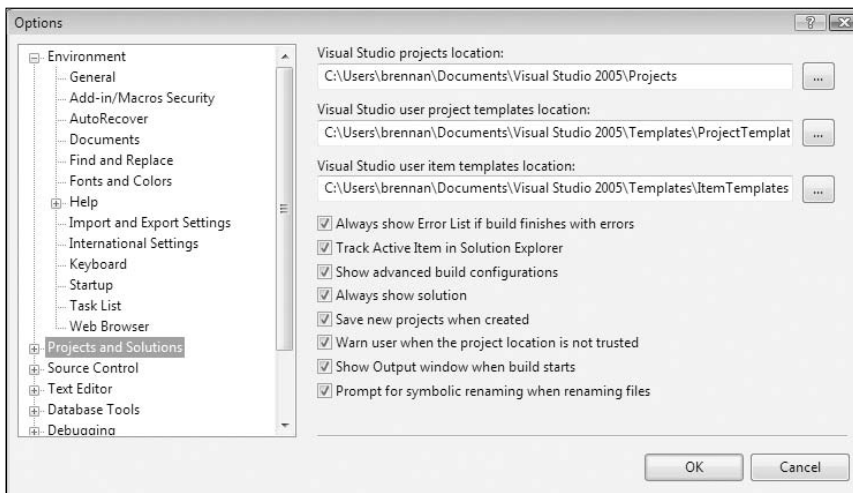


Figure 1-3. *Setting the Always Show Solution option*

This sample solution will be the template for all the examples in this book. Using the same basic structure for all projects provides a consistent basis for build automation. It also keeps everything in the same place for every application you develop. Being able to easily search the solution for a reference to a database table and then get the table creation script and any stored procedures using that table is very convenient. Most developers simply write and keep their table Data Definition Language (DDL) and stored procedures in the database and move them around by using tools within the database, never saving them as scripts that can be version-controlled in a Visual Studio project. This common practice fails to leverage one of the great strengths of Visual Studio 2005 and makes it harder to re-create database objects from scratch. As a result, changes to improve the application when it comes to database updates are avoided because of all the extra work necessary to make the change. When your environment allows you to work in a very agile way, you can take on tasks that may not be attempted otherwise.

Common Folders

As you work on many projects, you'll accumulate tools, templates, and scripts that are useful across multiple projects. It's helpful to place them into a common folder that you manage with source control so that the developers among multiple teams can leverage them as they do their work. With the Solution layout in Figure 1-2, it's trivial to drop in an MSBuild script and CMD scripts to a new project to provide build automation. You'll use the following folders for tools, templates, and scripts throughout the book (see Figure 1-4):

D:\Projects\Common\Tools

D:\Projects\Common\Templates

D:\Projects\Common\Scripts

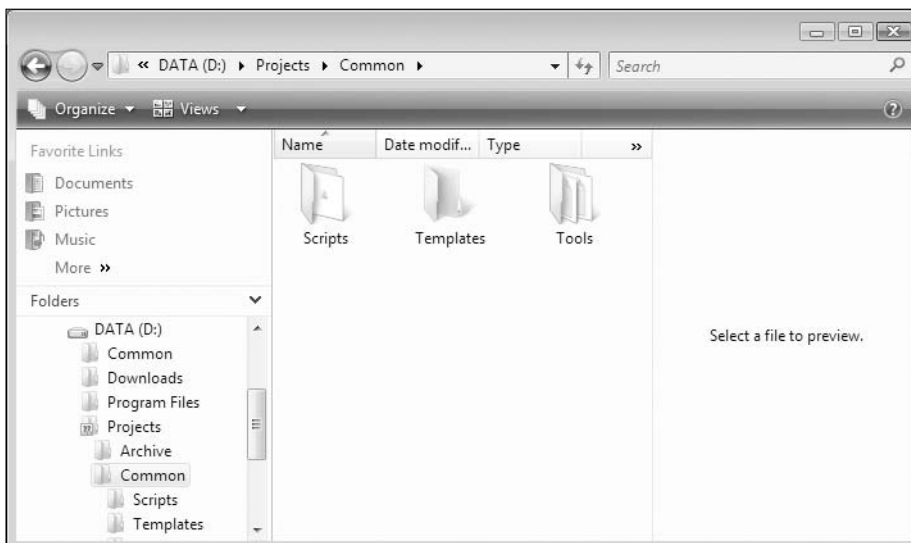


Figure 1-4. *Folders for tools, scripts, and templates*

To make it easier to script against these folders, let's add some system environment variables for these locations (see Figure 1-5):

```
DevTools = D:\Projects\Common\Tools
```

```
DevTemplates = D:\Projects\Common\Templates
```

```
DevScripts = D:\Projects\Common\Scripts
```

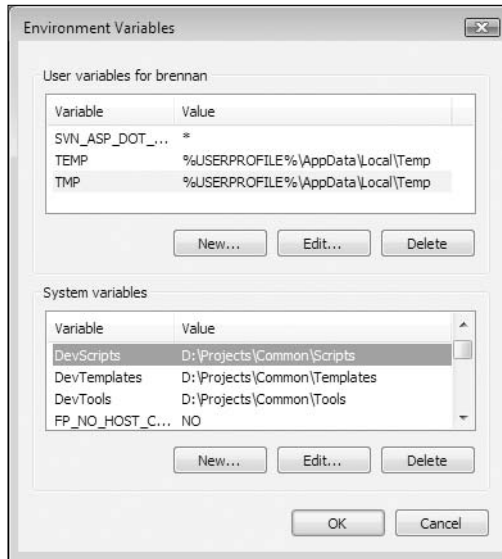


Figure 1-5. *Environment variables for development*

You'll build on these folders and variables throughout the book to enhance your common development environment.

Datasource Configuration

The datasource is the mechanism used to connect an ASP.NET 2.0 web application to data. For an ASP.NET application, the data is typically in a SQL Server database. To connect to a data-source, you use a connection string that sets various options for connecting to a database. There are many options available with connection strings. For the most basic connection string, you need the location of the database and the authentication details to access the database. The following is an example of such a connection string:

```
server=localhost;database=Products;uid=webuser;pwd=webpw
```

Notice that this includes server, database, uid, and pwd parameters, which provide all that is necessary to access the Products database on the local machine. This form should be very familiar to an ASP.NET developer. However, there are alternatives I will explain shortly.

MIXED-MODE AUTHENTICATION

With SQL Server, you can choose to allow for Windows authentication, SQL Server authentication, or both. This choice is presented to you when SQL Server is installed. SQL Server accounts are useful when the database is hosted on a remote server without access to the Windows domain. Windows authentication is helpful during development, when your users and the database server are running on the domain. For development and staging databases, you may grant different levels of access to groups such as database developers and web developers and then place people in those groups accordingly. For the production databases, you use a user account that is not shared with the entire development team so that only those limited users who should have access, do. The applications working with these databases just need the connection string updated to change authentication modes.

In an ASP.NET application, datasources are configured in the `Web.config` file in a section named `connectionStrings`. Each connection string is added to this section with a name, connection string, and provider name—for example:

```
<connectionStrings>
  <add name="SampleDatabase" connectionString="..."
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

In a team environment, you'll likely use a source-control system. You'll place your `Web.config` file into source control so that each member of the team has the same configuration. However, sharing a source-controlled configuration file can present some problems.

For starters, a common set of authentication values may be used. Using a Windows user account (for example, of one of the team members) exposes the password to the whole team—which is bad practice, especially if the password policy requires routine updates. Another approach is to use a SQL Server account that is shared by the team. This way is better than using a shared Windows account, but the following option works best. Each project should be configured with a trusted connection string so that the current user's account is used to connect with the database—for example:

```
server=localhost;Trusted_Connection=yes;database=Products;
```

Instead of providing the `uid` and `pwd` parameters, this uses Windows authentication to access the database. All developers use the same connection string but individually access the database by using their own accounts.

A trusted connection allows you to control who has access to what in the databases. When there is a universally shared user account, everyone will have access to everything, and no individual role management is possible. This way, you can set up the web development team with access to create and modify stored procedures while not giving them access to directly alter tables or data. Meanwhile, you can have your database team manage changes to the tables and stored procedures. And if you do have business analysts or management accessing the database, you can give them enough access to do what they need to do, but nothing that would cause trouble with the databases.

A side benefit is that your web team can focus on their concerns without having responsibility for what is happening to the database internals. It also gives the database team the

freedom to change the database structure while using updated stored procedures to provide the same public interface for the web team, taking in the same parameters and returning the same result columns. Keep in mind that sometimes some team members will wear hats for the web and database teams. That is perfectly fine. Some senior members of the team may have the right level of mastery on every part of the system to handle those responsibilities. But not every team member will be ready to take on that level of responsibility. Isolating the latter group of developers to what they can manage will give them and the project leader some peace of mind knowing that the project is in the right hands.

Code and Database Separation

On one recent project, I took care of changes to the database while developing the new website from the ground up. The catch was that this website was being built as a front end for multiple online stores and had a unique database structure for each store. That presented a real challenge. I planned to build just one basket and order management component but had to work with different databases while minimizing the effort to integrate the data with various front ends.

To make the same front end work with these multiple back ends, I created a clean integration point by using a set of stored procedures that had the same name and a set of parameters for each of the databases. These stored procedures each returned the same columns or output parameters. Internally, each would join differently named tables and gather column data from different locations than the next database, but ultimately would return data that the front end could easily interpret and display to customers. To run this online store with a different database, all that was needed was to implement those stored procedures. No coding changes were needed for the websites.

Another side benefit was that my team member was proficient with T-SQL and intimately familiar with each of the databases. By isolating her focus in that space, she was very productive and able to complete the integrations for a new website rollout much more quickly than I could have. As she worked on those changes, I was able to focus on requested changes to the front end as well as performance optimization.

As an alternative to using a set of stored procedures as a reliable integration layer, you could consider a set of web services (Windows Communication Foundation, or WCF) as a service-oriented architecture (SOA). I am sure that could work, but when a database already allows a wide range of platforms to talk with it, you already have a cross-platform approach. It may not use Extensible Markup Language (XML) or Web Services Interoperability Organization (WSI) specifications, but SQL Server does work with .NET, PHP, Java, Delphi, and a whole range of languages and platforms over an Open Database Connectivity (ODBC) connection. And the developers using those languages and platforms will already have some basic proficiency with SQL, so they can jump right into this approach without touching a single line of C# or Visual Basic (VB).

Managing Provider Services

The provider model was introduced to ASP.NET 2.0. It is to web applications what plug-ins are to web browsers. Included with the ASP.NET providers are the Membership, Roles, and Profile providers. Each of these providers has a SQL Server implementation, among others. By default, the SQL Server implementations are preconfigured for an ASP.NET website. However,

you must support these implementations with resources in the database. To prepare these resources, you will use the `aspnet_regsql.exe` utility.

The Visual Studio 2005 command prompt provides easy access to this utility from the console. The utility itself is located in the .NET 2.0 system directory, which is included on that path defined for the special command line used by the Visual Studio 2005 command prompt. The utility can be run with no arguments to start the wizard mode, which is a visual mode. Despite the name, the wizard mode is not nearly as powerful as the command line, which offers more options.

For starters, the visual mode turns on support for all providers. To enable just the Profile or Membership provider, you can request just those features from the command line and not add support for the Roles provider (which you may implement with a custom provider or not at all). Get a full list of the available options with the following command:

```
aspnet_regsql.exe -?
```

The available services supported by the utility include Membership, Role manager, Profiles, Personalization, and SQL Web event provider. Each can be selectively registered with the database.

Using the Command Line

Most of the time that I work with a website using the available providers, I use only the Membership, Roles, and Personalization support and leave the others off. To add just the desired services, use the following `Add Provider Services.cmd` script in Listing 1-1.

Listing 1-1. *Add Provider Services.cmd*

```
@echo off
set REGSQL="%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_regsql.exe"
set DSN="Data Source=.\SQLEXPRESS;Initial Catalog=Chapter01; ➤
Integrated Security=True"
%REGSQL% -C %DSN% -A mrpc
pause
```

This script can be placed at the root of a project, alongside the solution file, to make it easily available during development. It sets the location of first the utility and then the data-source to be used to host these features. Finally, the command is executed to add support for the Membership and Roles provider in the database.

The `-C` switch specifies the connection string, while `-A` specifies the list of services you want added to the database. To add all of them, you can simply specify `all` instead. The script takes a short time to complete. When it is done, you can see there will be new tables and stored procedures in the selected database.

Adding only the services that are going to be used is a practice of minimalism. Features have a tendency to be used if they are available, so by withholding the services that you do not plan to use, you save yourself the trouble of watching over them.

While you are developing a website with these features, you will occasionally want to reset everything and start from scratch. To do so, you can use a script to remove the provider services. However, the utility does not let you remove these services if there is data in tables

created by the utility. To help it out, you must delete the data in the right order because of the foreign key constraints among the tables.

To do so, use the following script named `WipeProviderData.sql` in Listing 1-2.

Listing 1-2. *WipeProviderData.sql*

```
--
-- WipeProviderData.sql
-- Wipes data from the provider services table so the services can be removed
-- (and added back fresh)
-- SELECT name FROM sysobjects WHERE type = 'U' and name like 'aspnet_%'
--

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
name = 'aspnet_WebEvent_Events')
BEGIN
    delete from dbo.aspnet_WebEvent_Events
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
name = 'aspnet_PersonalizationAllUsers')
BEGIN
    delete from dbo.aspnet_PersonalizationAllUsers
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
name = 'aspnet_PersonalizationPerUser')
BEGIN
    delete from dbo.aspnet_PersonalizationPerUser
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
name = 'aspnet_Membership')
BEGIN
    delete from dbo.aspnet_Membership
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND name = 'aspnet_Profile')
BEGIN
    delete from dbo.aspnet_Profile
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
```

```
name = 'aspnet_UsersInRoles')
BEGIN
    delete from dbo.aspnet_UsersInRoles
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND name = 'aspnet_Users')
BEGIN
    delete from dbo.aspnet_Users
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND name = 'aspnet_Roles')
BEGIN
    delete from dbo.aspnet_Roles
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND name = 'aspnet_Paths')
BEGIN
    delete from dbo.aspnet_Paths
END
GO

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
name = 'aspnet_Applications')
BEGIN
    delete from dbo.aspnet_Applications
END
GO
```

You can place this script into your scripts folder, D:\Projects\Common\Scripts, for use in multiple projects. When you want to use it, load it into SQL Server Management Studio and run it in the context of the database you want wiped. Then you can run the removal script, `Remove Provider Services.cmd`, shown in Listing 1-3.

Listing 1-3. *Remove Provider Services.cmd*

```
@echo off
set REGSQL="%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_regsql.exe"
set DSN="Data Source=.\SQLEXPRESS;Initial Catalog=Chapter01; ➡
Integrated Security=True"
%REGSQL% -C %DSN% -R mrpc
Pause
```

This script is identical to `Add Provider Services.cmd`, except for the simple change in the command-line switch from `-A` to `-R`, which specifies that the services are to be removed. After the data is wiped, this script will run successfully.

Mixing and Matching Providers

Because of the nature of the providers, it is possible to deploy these provider services to either the same database that the rest of the website is using or an entirely different database. It is possible to even deploy the services for each individual provider to a separate database. There can be good reason to do so. In one instance, you may be running a website connected to a massive database that needs to be taken offline occasionally for maintenance. In doing so, it may not be completely necessary to take the provider services offline as well. You may also find that you get better performance by having your provider services hosted on a database on a different piece of hardware.

Your configuration options go much further, as explained in the next section.

Configuring Providers

When you first create a new ASP.NET 2.0 website with Visual Studio 2005, it is already preconfigured to work with a set of defaults. These defaults are set in the *Machine.config* file, which is a part of the .NET 2.0 installation. Typically it is located at *C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\CONFIG*, or wherever .NET 2.0 has been installed onto your computer. The defaults for the provider configuration are near the bottom, in the *system.web* section, as shown in Listing 1-4.

Listing 1-4. *system.web* in *Machine.config*

```
<system.web>
  <processModel autoConfig="true"/>

  <httpHandlers />

  <membership>
    <providers>
      <add name="AspNetSqlMembershipProvider"
        type="System.Web.Security.SqlMembershipProvider, ➡
System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
        connectionStringName="LocalSqlServer"
        enablePasswordRetrieval="false"
        enablePasswordReset="true"
        requiresQuestionAndAnswer="true"
        applicationName="/"
        requiresUniqueEmail="false"
        passwordFormat="Hashed"
        maxInvalidPasswordAttempts="5"
        minRequiredPasswordLength="7"
        minRequiredNonalphanumericCharacters="1"
        passwordAttemptWindow="10"
        passwordStrengthRegularExpression="" />
    </providers>
  </membership>
```

```

    <profile>
      <providers>
        <add name="AspNetSqlProfileProvider"
connectionStringName="LocalSqlServer" applicationName="/"
        type="System.Web.Profile.SqlProfileProvider, System.Web, ↵
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </profile>

    <roleManager>
      <providers>
        <add name="AspNetSqlRoleProvider"
connectionStringName="LocalSqlServer" applicationName="/"
        type="System.Web.Security.SqlRoleProvider, System.Web, ↵
Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
        <add name="AspNetWindowsTokenRoleProvider" applicationName="/"
        type="System.Web.Security.WindowsTokenRoleProvider, XXX
System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a" />
      </providers>
    </roleManager>
  </system.web>

```

A default connection string named `LocalSqlServer` is also defined, which looks for a file called `aspnetdb.mdf` in the `App_Data` folder (see Listing 1-5).

Listing 1-5. *connectionStrings in Machine.config*

```

<connectionStrings>
  <add name="LocalSqlServer"
connectionString="data source=.\SQLEXPRESS;Integrated ↵
Security=SSPI;AttachDBFilename=|DataDirectory|aspnetdb.mdf;User Instance=true" ↵
providerName="System.Data.SqlClient"/>
</connectionStrings>

```

The default `LocalSqlServer` connection is referenced by the Membership, Roles, and Profile provider configurations in the `system.web` block. The `datasource` specifies that this database exists in the `DATA_DIRECTORY`. That `DATA_DIRECTORY` for an ASP.NET 2.0 website is the `App_Data` folder. If you were to create a new website and start using the Membership and Profile services, this SQL Express database would be created for you in the `App_Data` folder. However, if you have the Standard or Professional Edition of SQL Server installed, this process would fail because SQL Express is required. This default configuration can be quite handy, but also dangerous if you do not adjust it for the deployed environment.

For each provider configuration, the parent block has multiple attributes for the respective provider implementation. And within that block, you have the ability to add, remove, or even clear the provider implementations. In your new website's `Web.config` file, you will want to clear the defaults set by `Machine.config` and customize them specifically for your needs.

Next you will configure a new website to use the SQL implementations of the Membership, Roles, and Profile providers. Before those are configured, you must prepare the `datasource`.

Assuming you have a sample website that will work with a database called *Sample* and that has been prepared with the provider services as described in the previous section, use the configuration in Listing 1-6.

Listing 1-6. *Custom Web.config*

```
<connectionStrings>
  <add name="sampledb"
        connectionString="Data Source=.\SQLEXPRESS;Initial Catalog=Sample; ➡
Integrated Security=True"
        providerName="System.Data.SqlClient"/>
</connectionStrings>
```

During the development process, I use a trusted database connection and talk to the local machine. Now you are ready to configure the providers with the SQL implementations. And because the defaults in *Machine.config* already use the SQL implementations, let's start with them and then make our adjustments.

Membership Configuration

For starters, you'll organize the configuration block to make it more readable. You will be looking at this quite a bit during development, so making it easy to read at a glance will save you time. Place each attribute on a separate line and indent the attributes to line them up. Then make the name, applicationName, and connectionStringName the first few attributes. These are the critical values.

Next you want to ensure that this is the only Membership provider for this website. Add the `clear` element before the `add` element in the membership block to tell the ASP.NET runtime to clear all preconfigured settings. Then add an attribute called `defaultProvider` to the membership element and set it to the same value as the name for the newly added provider configuration. Listing 1-7 shows the Membership configuration. Table 1-1 covers the various settings that are available.

Listing 1-7. *Membership Configuration*

```
<membership defaultProvider="Chapter01SqlMembershipProvider">
  <providers>
    <clear/>
    <add
      name="Chapter01SqlMembershipProvider"
      applicationName="/chapter01"
      connectionStringName="chapter01db"
      enablePasswordRetrieval="true"
      enablePasswordReset="true"
      requiresQuestionAndAnswer="true"
      requiresUniqueEmail="false"
      passwordFormat="Clear"
      maxInvalidPasswordAttempts="5"
      minRequiredPasswordLength="7"
```

```

        minRequiredNonalphanumericCharacters="0"
        passwordAttemptWindow="10"
        passwordStrengthRegularExpression=""
        type="System.Web.Security.SqlMembershipProvider, ↵
System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    />
</providers>
</membership>

```

Table 1-1. *Membership Configuration Settings*

Setting	Description
name	Specifies the configuration name referenced by the membership element
applicationName	Defines the application name used as a scope in the Membership database
connectionStringName	Specifies the connection string to use for this provider
enablePasswordRetrieval	Specifies whether this provider allows for password retrieval
enablePasswordReset	Specifies whether this provider can reset the user password (enabled = true)
requiresQuestionAndAnswer	Specifies whether the question and answer are required for password reset and retrieval
requiresUniqueEmail	Specifies whether this provider requires a unique e-mail address per user
passwordFormat	Specifies the password format, such as cleared, hashed (default), and encrypted
maxInvalidPasswordAttempts	Specifies how many failed login attempts are allowed before the user account is locked
minRequiredPasswordLength	Specifies the minimal number of characters required for the password
minRequiredNonalphanumericCharacters	Specifies the number of special characters that must be present for the password
passwordAttemptWindow	The duration in minutes when failed attempts are tracked
passwordStrengthRegularExpression	A regular expression used to check a password string for the required password strength

Roles Configuration

Now you'll do much of the same with the Roles provider. This configuration block is called `roleManager`, and in addition to the `defaultProvider` attribute, it also has an attribute called `enabled`, which allows you to turn it on and off. Later you can easily access the `enabled` setting in your code by using the `Roles.Enabled` property. By default this value is set to `false`, so it must be set to add support for roles even if you have configured a provider. See the following roles configuration in Listing 1-8.

Listing 1-8. Roles Configuration

```

<roleManager defaultProvider="Chapter01SqlRoleProvider" enabled="true">
  <providers>
    <clear/>
    <add
      name="Chapter01SqlRoleProvider"
      connectionStringName="chapter01db"
      applicationName="/chapter01"
      type="System.Web.Security.SqlRoleProvider, ↵
System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"
    />
  </providers>
</roleManager>

```

Profile Configuration

Finally, you can add support for the Profile provider. And as with the preceding two provider configurations, you will set the `defaultProvider` attribute to match the newly added configuration and also add the `clear` element to ensure that this is the only configured provider. The unique feature for the Profile provider configuration is the ability to define custom properties. In the sample configuration shown in Listing 1-9, a few custom properties have been defined: `FirstName`, `LastName`, and `BirthDate`. I will explain these properties in a bit; Table 1-2 lists the profile configuration settings.

Listing 1-9. Profile Configuration

```

<profile defaultProvider="Chapter01SqlProfileProvider"
  automaticSaveEnabled="true" enabled="true">
  <providers>
    <clear/>
    <add
      name="Chapter01SqlProfileProvider"
      applicationName="/chapter01"
      connectionStringName="chapter01db"
      type="System.Web.Profile.SqlProfileProvider, ↵
System.Web, Version=2.0.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a"/>
  </providers>
  <properties>
    <add name="FirstName" type="String" allowAnonymous="true" />
    <add name="LastName" type="String" allowAnonymous="true" />
    <add name="BirthDate" type="DateTime" allowAnonymous="true" />
  </properties>
</profile>

```

Table 1-2. *Profile Configuration Settings*

Setting	Description
name	Specifies the configuration name referenced by the profile element
applicationName	Defines the application name used as a scope in the Profile database
connectionStringName	Specifies the connection string to use for this provider

You can see those custom properties near the end. The Profile provider allows you to manage properties that are bound to the ASP.NET user. You can define anything that can be serialized. In the three examples, both String and DateTime objects can easily be serialized for storage in the SQL Server database. However, the properties can be much more complicated objects than these common primitive types. Perhaps you have a business object called Employee, which holds properties such as Title, Name, Department, and Office. Instead of configuring all these properties, you can instead just specify the Employee object as a property.

Then with code in either your code-behind classes or classes held with the App_Code folder, you can access this property as Profile.Employee. Thanks to the dynamic compiler, which is part of the ASP.NET runtime, these properties are immediately available within Visual Studio with IntelliSense support. That sounds really powerful, doesn't it? It is.

However, you can quickly paint yourself into a corner if you set up many complex objects as Profile properties. What happens when you need to add or remove properties to the Employee object but you already have many serialized versions of the object held in the database? How will you upgrade them?

When ASP.NET 2.0 first came out, I saw all kinds of examples of how you could create an object called ShoppingBasket, add objects called Items to the ShoppingBasket, and set up the ShoppingBasket as a Profile property. I immediately knew that would not be something I was going to attempt. In the year previous to the launch of .NET 2.0, I was building a commerce website to manage a basket with items and customer orders, and I never used the Profile properties. I still had my ShoppingBasket object, which held lots of Item objects, but I managed the tables and stored procedures used with those objects so that I could add new properties to them and easily manage the changes. If the Item started out with just one value for price called Price, and later I needed to add a couple more such as SalesPrice and SeasonalPrice, it was easy enough to add them. And to associate them to the current website user, I used the following technique.

For this particular website, I had to work with anonymous and authenticated users. When a user first came to the site, that user was given a token to identify him as he moved from page to page. This was an anonymous user token. After the user created an account, that anonymous token would be deleted and the user would be migrated to an authenticated user token. Authenticated users were Members with the Membership provider. To seamlessly work with anonymous and authenticated users, I needed a way to uniquely identify these users. Unfortunately, there is no such default value as Profile.UserID. There is a property for Profile.UserName, but that is available only for authenticated users. So I ended up creating a wrapper property, which provided a Guid value whether the user was anonymous or authenticated.

ANONYMOUS PROFILES

To make use of properties on the profile, you must first have anonymous profiles enabled. An element in `system.web` named `anonymousIdentification` can be enabled to turn on this feature.

In the `App_Code` folder, I created a class called `Utility` and added the properties in Listing 1-10.

Listing 1-10. *Utility Methods*

```
public static bool IsUserAuthenticated
{
    get
    {
        return HttpContext.Current.User.Identity.IsAuthenticated;
    }
}

public static Guid UserID
{
    get
    {
        if (IsUserAuthenticated)
        {
            return (Guid)Membership.GetUser().ProviderUserKey;
        }
        else
        {
            Guid userId =
                new Guid(HttpContext.Current.Request.AnonymousID.Substring(0, 36));
            return userId;
        }
    }
}
}fcdd
```

Whenever I needed the unique identifier for the current user, I would access it with `Utility.UserID`. But then I had to handle the transition from anonymous to authenticated. To do so, I added a method to `Global.asax` called `Profile_OnMigrateAnonymous`, shown in Listing 1-11.

Listing 1-11. *Profile_OnMigrateAnonymous*

```
public void Profile_OnMigrateAnonymous(object sender, ProfileMigrateEventArgs args)
{
    Guid anonID = new Guid(args.AnonymousID);
    Guid authId = (Guid)Membership.GetUser().ProviderUserKey;
```

```
// migrate anonymous user resources to the authenticated user

// remove the anonymous user.
Membership.DeleteUser(args.AnonymousID, true);
}
```

In the case of the basket, I just added the items that were in the anonymous user's basket into the authenticated user's basket, and removed the anonymous account and all associated data.

I have considered adding a `Guid` property to the Profile properties called `UserID`, but the dynamic compiler for the ASP.NET runtime works only in code-behind files. It would not work in the Utility class held in the `App_Code` directory, which is where I place a good deal of the code. So the preceding technique was the only option.

Creating Users and Roles

When you work on a website with Visual Studio, you are able to use the Website Administration tool to create users and roles. But this utility is not a feature built into Microsoft Internet Information Server (IIS). To work with users and roles, you have to do it yourself—either in the database by carefully calling stored procedures or by creating an interface to safely use the Membership API. I chose to create a couple of user controls that I can easily drop into any website.

The two main controls, `UserManager.ascx` and `RolesManager.ascx`, do the work to manage the users and roles. These controls are held in another user control called `MembersControl.ascx`. This control switches between three views to create a new user, edit existing users, and edit roles. Listings 1-12 through 1-17 provide the full source for these controls.

Listing 1-12. *UserManager.ascx*

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="UserManager.ascx.cs" Inherits="MemberControls_UserManager" %>
<asp:Label ID="TitleLabel"
    runat="server" Text="User Manager"></asp:Label><br />
<asp:MultiView ID="UsersMultiView" runat="server"
    ActiveViewIndex="0">
    <asp:View ID="SelectUserView" runat="server"
        OnActivate="SelectUserView_Activate">

        <table>
        <tr>
        <td>
            <asp:GridView ID="UsersGridView" runat="server"
                AllowPaging="True"
                AutoGenerateColumns="False"
                OnInit="UsersGridView_Init"
                OnPageIndexChanging="UsersGridView_PageIndexChanging"
                OnRowCommand="UsersGridView_RowCommand">
```



```

GridLines="None">
<Columns>
    <asp:BoundField DataField="UserName" HeaderText="Username" />
    <asp:BoundField DataField="Email" HeaderText="Email" />
    <asp:TemplateField ShowHeader="False">
        <ItemTemplate>
            <asp:LinkButton ID="LinkButton1" runat="server"
                CausesValidation="false"
                CommandName="ViewUser"
                CommandArgument='<%=# Bind("UserName") %>'
                Text="View"></asp:LinkButton>
        </ItemTemplate>
    </asp:TemplateField>
</Columns>
<RowStyle CssClass="EvenRow" />
<HeaderStyle CssClass="HeaderRow" />
<AlternatingRowStyle CssClass="OddRow" />
</asp:GridView>
</td>
</tr>
<tr>
<td align="center">
    <asp:TextBox ID="FilterUsersTextBox" runat="server"
        Width="75px"></asp:TextBox>
    <asp:Button ID="FilterUsersButton" runat="server"
        OnClick="FilterUsersButton_Click" Text="Filter" />
</td>
</tr>
</table>

</asp:View>
<asp:View ID="UserView" runat="server" OnActivate="UserView_Activate">

<table>
    <tr>
        <td class="Label">
            <asp:Label ID="UserNameLabel" runat="server" Text="User Name: "
                Font-Bold="True"></asp:Label></td>
        <td class="Data">
            <asp:Label ID="UserNameValueLabel" runat="server"
                Text=""></asp:Label></td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="ApprovedLabel" runat="server" Text="Approved: "
                Font-Bold="True"></asp:Label></td>
        <td class="Data">

```

```

        <asp:Label ID="ApprovedValueLabel" runat="server"
            Text=""></asp:Label></td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="LockedOutLabel" runat="server" Text="Locked Out: "
                Font-Bold="True"></asp:Label></td>
        <td class="Data">
            <asp:Label ID="LockedOutValueLabel" runat="server"
                Text=""></asp:Label></td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="OnlineLabel" runat="server" Text="Online: "
                Font-Bold="True"></asp:Label></td>
        <td class="Data">
            <asp:Label ID="OnlineValueLabel" runat="server"
                Text=""></asp:Label></td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="CreationLabel" runat="server" Text="Creation: "
                Font-Bold="True"></asp:Label></td>
        <td class="Data">
            <asp:Label ID="CreationValueLabel" runat="server"
                Text=""></asp:Label></td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="LastActivityLabel" runat="server"
                Text="Last Activity:" Font-Bold="True"></asp:Label></td>
        <td class="Data">
            <asp:Label ID="LastActivityValueLabel" runat="server"
                Text=""></asp:Label></td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="LastLoginLabel" runat="server"
                Text="Last Login:" Font-Bold="True"></asp:Label></td>
        <td class="Data">
            <asp:Label ID="LastLoginValueLabel" runat="server"
                Text=""></asp:Label></td>
    </tr>
    <tr>
        <td colspan="2" class="Data">
            <asp:Label ID="UserCommentLabel" runat="server"
                Text="Comment:" Font-Bold="True"></asp:Label><br />

```

```

        <asp:Label ID="UserCommentValueLabel" runat="server"
            Text=""></asp:Label>
    </td>
</tr>
<tr>
    <td colspan="2">
        <asp:Button ID="EditUserButton" runat="server"
            Text="Edit User" OnClick="EditUserButton_Click" />
        <asp:Button ID="ResetPasswordButton" runat="server"
            Text="Reset Password" OnClick="ResetPasswordButton_Click"
            Visible="False" />
        <asp:Button ID="UnlockUserButton" runat="server"
            OnClick="UnlockUserButton_Click" Text="Unlock" />
        <asp:Button ID="ReturnViewUserButton" runat="server"
            Text="Return" OnClick="ReturnViewUserButton_Click" />
    </td>
</tr>
</table>

</asp:View>
<asp:View ID="EditorView" runat="server" OnActivate="EditorView_Activate">

<table>
    <tr>
        <td class="Label">
            <asp:Label ID="UserName2Label" runat="server" Text="User Name: "
                Font-Bold="True"></asp:Label></td>
            <td class="Data">
                <asp:Label ID="UserNameValue2Label" runat="server" Text="">
                </asp:Label></td>
        <td>
            &nbsp;   </td>
    </tr>
    <tr>
        <td class="Label">
            <asp:Label ID="EmailLabel" runat="server" Text="Email: "
                Font-Bold="True"></asp:Label>
        </td>
        <td class="Data">
            <asp:TextBox ID="EmailTextBox" runat="server"
                AutoCompleteType="Email"></asp:TextBox></td>
        <td>
            <asp:RequiredFieldValidator
                ID="EmailRequiredFieldValidator" runat="server"
                ErrorMessage="*"
                ControlToValidate="EmailTextBox"
                EnableClientScript="False"></asp:RequiredFieldValidator>
            <asp:RegularExpressionValidator

```

```

        ID="RegularExpressionValidator1" runat="server"
        ControlToValidate="EmailTextBox"
        EnableClientScript="False"
        ErrorMessage="*"
        ValidationExpression="\w+([-+.']\w+)*@\w+([-.\w+)*
*\.\w+([-.\w+)*"></asp:RegularExpressionValidator>
    </td>
</tr>
<tr>
    <td class="Label">
        <asp:Label ID="CommentLabel" runat="server" Text="Comment: "
        Font-Bold="True"></asp:Label>
    </td>
    <td class="Data">
        <asp:TextBox ID="CommentTextBox" runat="server"
        AutoCompleteType="Email" TextMode="Multiline"></asp:TextBox></td>
    <td>
        &nbsp;   </td>
</tr>
<tr>
    <td class="Label"><asp:Label ID="Approved2Label" runat="server"
    Text="Approved: " Font-Bold="True"></asp:Label></td>
    <td class="Data"><asp:CheckBox ID="ApprovedCheckBox" runat="server" /></td>
</tr>
<tr>
    <td class="Label"><asp:Label ID="RolesLabel" runat="server"
    Text="Roles " Font-Bold="True"></asp:Label></td>
    <td class="Data">
        <asp:CheckBoxList ID="RolesCheckBoxList" runat="server">
            </asp:CheckBoxList>
        </td>
</tr>
<tr>
    <td colspan="3">
        <asp:Button ID="UpdateUserButton" runat="server"
        Text="Update User" OnClick="UpdateUserButton_Click" />
        <asp:Button ID="CancelEditUserButton" runat="server"
        OnClick="CancelEditUserButton_Click"
        Text="Cancel" /></td>
</tr>
</table>

</asp:View>
</asp:MultiView>

```

Listing 1-13. *UserManager.ascx.cs*

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class MemberControls_UserManager : UserControl
{
    #region " Events "
    protected void Page_Init(object sender, EventArgs e)
    {
    }
    protected void Page_Load(object sender, EventArgs e)
    {
    }
    protected void FilterUsersButton_Click(object sender, EventArgs e)
    {
        BindUsersGridView();
    }
    protected void EditUserButton_Click(object sender, EventArgs e)
    {
        UsersMultiView.SetActiveView(EditorView);
    }
    protected void UnlockUserButton_Click(object sender, EventArgs e)
    {
        MembershipUser user = CurrentUser;
        if (user != null)
        {
            user.UnlockUser();
            Membership.UpdateUser(user);
            BindUserView();
        }
    }
    protected void ResetPasswordButton_Click(object sender, EventArgs e)
    {
        MembershipUser user = CurrentUser;
        if (user != null)
        {
            user.ResetPassword();
        }
        UsersMultiView.SetActiveView(UserView);
    }
    protected void ReturnViewUserButton_Click(object sender, EventArgs e)
    {
    }
}
```

```

        UsersMultiView.SetActiveView(SelectUserView);
    }
    protected void UpdateUserButton_Click(object sender, EventArgs e)
    {
        if (CurrentUser != null)
        {
            MembershipUser user = CurrentUser;
            user.Email = EmailTextBox.Text;
            user.Comment = CommentTextBox.Text;
            user.IsApproved = ApprovedCheckBox.Checked;
            Membership.UpdateUser(user);

            foreach (ListItem listItem in RolesCheckBoxList.Items)
            {
                string role = listItem.Value;
                if (Roles.RoleExists(role))
                {
                    if (!listItem.Selected &&
                        Roles.IsUserInRole(user.UserName, role))
                    {
                        Roles.RemoveUserFromRole(user.UserName, role);
                    }
                    else if (listItem.Selected &&
                        !Roles.IsUserInRole(user.UserName, role))
                    {
                        Roles.AddUserToRole(user.UserName, role);
                    }
                }
            }

            UsersMultiView.SetActiveView(UserView);
        }
    }
    protected void CancelEditUserButton_Click(object sender, EventArgs e)
    {
        UsersMultiView.SetActiveView(UserView);
    }
    protected void CancelRolesButton_Click(object sender, EventArgs e)
    {
        UsersMultiView.SetActiveView(UserView);
    }
    protected void SelectUserView_Activate(object sender, EventArgs e)
    {
        BindUsersGridView();
    }
    protected void UserView_Activate(object sender, EventArgs e)
    {

```

```

        BindUserView();
    }
    protected void EditorView_Activate(object sender, EventArgs e)
    {
        BindEditorView();
    }
    protected void UsersGridView_Init(object sender, EventArgs e)
    {
    }

    protected void UsersGridView_PageIndexChanging(
        object sender, GridViewPageEventArgs e)    {
        UsersGridView.PageIndex = e.NewPageIndex;
        BindUsersGridView();
    }
    protected void UsersGridView_RowCommand(
        object sender, GridViewCommandEventArgs e)
    {
        if ("ViewUser".Equals(e.CommandName))
        {
            CurrentUser = GetUser(e.CommandArgument.ToString());
            UsersMultiView.SetActiveView(UserView);
        }
    }
}
#endregion

#region " Methods "
private void BindUsersGridView()
{
    if (String.Empty.Equals(FilterUsersTextBox.Text.Trim()))
    {
        UsersGridView.DataSource = Membership.GetAllUsers();
    }
    else
    {
        List<MembershipUser> filteredUsers = new List<MembershipUser>();
        string filterText = FilterUsersTextBox.Text.Trim();
        foreach (MembershipUser user in Membership.GetAllUsers())
        {
            if (user.UserName.Contains(filterText) ||
                user.Email.Contains(filterText))
            {
                filteredUsers.Add(user);
            }
        }
        UsersGridView.DataSource = filteredUsers;
    }
}

```

```

        UsersGridView.DataBind();
    }
    private void BindUserView()
    {
        MembershipUser user = CurrentUser;
        if (user != null)
        {
            UserNameValueLabel.Text = user.UserName;
            ApprovedValueLabel.Text = user.IsApproved.ToString();
            LockedOutValueLabel.Text = user.IsLockedOut.ToString();
            OnlineValueLabel.Text = user.IsOnline.ToString();
            CreationValueLabel.Text = user.CreationDate.ToString("d");
            LastActivityValueLabel.Text = user.LastActivityDate.ToString("d");
            LastLoginValueLabel.Text = user.LastLoginDate.ToString("d");
            UserCommentValueLabel.Text = user.Comment;

            UnlockUserButton.Visible = user.IsLockedOut;
            ResetPasswordButton.Attributes.Add("onclick",
                "return confirm('Are you sure?');");
        }
    }
    private void BindEditorView()
    {
        MembershipUser user = CurrentUser;
        if (user != null)
        {
            UserNameValue2Label.Text = user.UserName;
            EmailTextBox.Text = user.Email;
            CommentTextBox.Text = user.Comment;
            ApprovedCheckBox.Checked = user.IsApproved;

            RolesCheckBoxList.Items.Clear();
            foreach (string role in Roles.GetAllRoles())
            {
                ListItem listItem = new ListItem(role);
                listItem.Selected = Roles.IsUserInRole(user.UserName, role);
                RolesCheckBoxList.Items.Add(listItem);
            }
        }
    }

    public void Reset()
    {
        UsersMultiView.SetActiveView(SelectUserView);
        Refresh();
    }

```



```
public void Refresh()
{
    BindUsersGridView();
}

public bool IsUserAuthenticated
{
    get
    {
        return HttpContext.Current.User.Identity.IsAuthenticated;
    }
}

public string GetUserName()
{
    if (IsUserAuthenticated)
    {
        return HttpContext.Current.User.Identity.Name;
    }
    return String.Empty;
}

public MembershipUser GetUser(string username)
{
    return Membership.GetUser(username);
}

#endregion

#region " Properties "
[Category("Appearance"), Browsable(true), DefaultValue("User Manager")]
public string Title
{
    get
    {
        return TitleLabel.Text;
    }
    set
    {
        TitleLabel.Text = value;
        EnsureChildControls();
    }
}

[Category("Appearance"), Browsable(true), DefaultValue(false)]
public bool TitleBold
{
    get
    {
        return TitleLabel.Font.Bold;
    }
}
```

```

    }
    set
    {
        TitleLabel.Font.Bold = value;
        EnsureChildControls();
    }
}

[Browsable(false)]
public CssStyleCollection TitleStyle
{
    get
    {
        return TitleLabel.Style;
    }
}
private MembershipUser CurrentUser
{
    get
    {
        return ViewState["CurrentUser"] as MembershipUser;
    }
    set
    {
        ViewState["CurrentUser"] = value;
    }
}
}
#endregion
}

```

Listing 1-14. *RoleManager.ascx*

```

<%@ Control Language="C#" AutoEventWireup="true" CodeFile="RolesManager.ascx.cs"
    Inherits="MemberControls_RolesManager" %>
<asp:Label ID="TitleLabel" runat="server"
    Text="Roles Manager" Font-Bold="true"></asp:Label>
<table>
    <tr>
        <td align="center">
            <asp:GridView ID="RolesGridView" runat="server"
                AutoGenerateColumns="False"
                OnRowCommand="RolesGridView_RowCommand"
                OnRowDataBound="RolesGridView_RowDataBound"
                GridLines="None"
                Width="100%">
                <Columns>
                    <asp:TemplateField HeaderText="Role">
                        <ItemTemplate>

```

```

        <asp:Label ID="Label1" runat="server"
            Text="Role"></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Users">
        <ItemTemplate>
            <asp:Label ID="Label1" runat="server"
                Text="Users"></asp:Label>
        </ItemTemplate>
    </asp:TemplateField>
    <asp:ButtonField CommandName="RemoveRole"
        Text="Remove" />
</Columns>
<EmptyDataTemplate>
    &nbsp;&nbsp;&nbsp;- No Roles -
</EmptyDataTemplate>
<RowStyle CssClass="EvenRow" />
<AlternatingRowStyle CssClass="OddRow" />
<HeaderStyle CssClass="HeaderRow" />
</asp:GridView>
</td>
</tr>
<tr>
    <td align="center">
        <asp:Label ID="Label3" runat="server"
            Font-Bold="True" Text="Role: "></asp:Label>
        <asp:TextBox ID="AddRoleTextBox" runat="server"
            Width="75px"></asp:TextBox>
        <asp:Button ID="AddRoleButton" runat="server"
            Text="Add" OnClick="AddRoleButton_Click" /></td>
    </tr>
</table>

```

Listing 1-15. *RoleManager.ascx.cs*

```

using System;
using System.ComponentModel;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class MemberControls_RolesManager : UserControl
{
    #region " Events "

    protected void Page_PreRender(object sender, EventArgs e)
    {
        BindRolesGridView();
    }
}

```

```

    }
    protected void RolesGridView_RowDataBound(object sender, GridViewRowEventArgs e)
    {
        if (e.Row.RowType == DataControlRowType.DataRow)
        {
            string role = e.Row.DataItem as string;
            foreach (TableCell cell in e.Row.Cells)
            {
                foreach (Control control in cell.Controls)
                {
                    Label label = control as Label;
                    if (label != null)
                    {
                        if ("Role".Equals(label.Text))
                        {
                            label.Text = role;
                        }
                        else if ("Users".Equals(label.Text))
                        {
                            label.Text = Roles.GetUsersInRole(role). ➡
Length.ToString();
                        }
                    }
                    else
                    {
                        LinkButton button = control as LinkButton;
                        if (button != null)
                        {
                            button.Enabled = Roles.GetUsersInRole(role).Length == 0;
                            if (button.Enabled)
                            {
                                button.CommandArgument = role;
                                button.Attributes.Add("onclick",
                                    "return confirm('Are you sure?');");
                            }
                        }
                    }
                }
            }
        }
    }
    protected void RolesGridView_RowCommand(
        object sender, GridViewCommandEventArgs e)
    {
        if ("RemoveRole".Equals(e.CommandName))
        {
            string role = e.CommandArgument as string;
            Roles.DeleteRole(role, true);
        }
    }
}

```

```

        BindRolesGridView();
    }
}
protected void AddRoleButton_Click(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        string role = AddRoleTextBox.Text;
        if (!Roles.RoleExists(role))
        {
            Roles.CreateRole(role);
            AddRoleTextBox.Text = String.Empty;
            BindRolesGridView();
        }
    }
}

#endregion
#region " Methods "

public void Refresh()
{
    BindRolesGridView();
}
private void BindRolesGridView()
{
    RolesGridView.DataSource = Roles.GetAllRoles();
    RolesGridView.DataBind();
}

#endregion
#region " Properties "

[Category("Appearance"), Browseable(true), DefaultValue("Roles Manager")]
public string Title
{
    get
    {
        return TitleLabel.Text;
    }
    set
    {
        TitleLabel.Text = value;
    }
}

#endregion
}

```

Listing 1-16. *MembersControl.ascx*

```

<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="MembersControl.ascx.cs"
    Inherits="MembersControl" %>
<%@ Register Src="UserManager.ascx" TagName="UserManager" TagPrefix="uc2" %>
<%@ Register Src="RolesManager.ascx" TagName="RolesManager" TagPrefix="uc1" %>

    <br />
    <br />
    <b>Select View:</b>
    <asp:DropDownList ID="NavDropDownList" runat="server"
        AutoPostBack="True"
        OnSelectedIndexChanged="NavDropDownList_SelectedIndexChanged">
        <asp:ListItem>Create User</asp:ListItem>
        <asp:ListItem>Manage Users</asp:ListItem>
        <asp:ListItem>Manage Roles</asp:ListItem>
    </asp:DropDownList><br />
    <br />

    <asp:MultiView ID="MultiView1" runat="server">
    <asp:View ID="UserCreationView" runat="server"
        OnActivate="UserCreationView_Activate">
        <asp:CreateUserWizard ID="CreateUserWizard1" runat="server"
            AutoGeneratePassword="True"
            LoginCreatedUser="False"
            OnCreatedUser="CreateUserWizard1_CreatedUser">
            <WizardSteps>
            <asp:CreateUserWizardStep
                ID="CreateUserWizardStep1" runat="server">
            </asp:CreateUserWizardStep>
            <asp:CompleteWizardStep
                ID="CompleteWizardStep1" runat="server">
            </asp:CompleteWizardStep>
            </WizardSteps>
        </asp:CreateUserWizard>
    </asp:View>
    <asp:View ID="UserManagerView" runat="server"
        OnActivate="UserManagerView_Activate">

        <uc2:UserManager
            ID="UserManager1" runat="server"
            Title="Users"
            TitleBold="true" />
    </asp:View>
    <asp:View ID="RolesManagerView" runat="server">

        <uc1:RolesManager

```

```

        ID="RolesManager1" runat="server"
        Title="Roles" />

</asp:View>
</asp:MultiView>

```

Listing 1-17. *MembersControl.ascx.cs*

```

using System;
using System.Web.UI;

public partial class MembersControl : UserControl
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            MultiView1.SetActiveView(UserManagerView);
            NavDropDownList.SelectedValue = "Manage Users";
        }
    }

    protected void NavDropDownList_SelectedIndexChanged(object sender, EventArgs e)
    {
        if ("Create User".Equals(NavDropDownList.SelectedValue))
        {
            MultiView1.SetActiveView(UserCreationView);
        }
        else if ("Manage Users".Equals(NavDropDownList.SelectedValue))
        {
            MultiView1.SetActiveView(UserManagerView);
            RefreshUserManager();
        }
        else if ("Manage Roles".Equals(NavDropDownList.SelectedValue))
        {
            MultiView1.SetActiveView(RolesManagerView);
            RefreshRolesManager();
        }
    }

    protected void CreateUserWizard1_CreatedUser(object sender, EventArgs e)
    {
        MultiView1.SetActiveView(UserManagerView);
        NavDropDownList.SelectedValue = "Manage Users";
        RefreshUserManager();
    }
}

```

```

private void RefreshUserManager()
{
    MemberControls_UserManager userManager =
        UserManagerView.FindControl("UserManager1")
        as MemberControls_UserManager;
    if (userManager != null)
    {
        userManager.Refresh();
    }
}

private void RefreshRolesManager()
{
    MemberControls_RolesManager rolesManager =
        UserManagerView.FindControl("RolesManager1")
        as MemberControls_RolesManager;
    if (rolesManager != null)
    {
        rolesManager.Refresh();
    }
}

protected void UserManagerView_Activate(object sender, EventArgs e)
{
    UserManager1.Reset();
}

protected void UserCreationView_Activate(object sender, EventArgs e)
{
    CreateUserWizard1.ActiveStepIndex = 0;
}
}

```

Securing the Admin Section

With the user and role management controls placed into a folder named `Admin`, it can be secured by requiring authenticated users in the `Admin` role. Near the end of `Web.config`, you can create a location configuration for the `Admin` path and allow members in the `Admin` role (see Listing 1-18).

Listing 1-18. *Securing the Admin Section with Web.config*

```

<?xml version="1.0"?>

<configuration>

    <!-- other configuration settings -->

```



```

<location path="Admin">
  <system.web>
    <authorization>
      <allow roles="Admin"/>
      <deny users="*/>
    </authorization>
  </system.web>
</location>

</configuration>

```

Creating the Admin User

After your Admin section is ready, you will naturally need the Admin user so you can log in to this section and use the controls. It is sort of a catch-22 scenario. But just as you can use the Membership API to manage users and roles, with these controls you can also programmatically create users and roles. To automatically ensure that your website has the necessary Admin user, you can add the code to do all of this work to the `Application_Start` event handler in the `Global.asax` file for the website. I first check whether the three default roles exist and add each one that does not exist. And then if there are no users, I have the method add the default Admin user, as shown in Listing 1-19.

Listing 1-19. Adding Roles and Users

```

public void Application_Start(object sender, EventArgs e)
{
    if (Roles.Enabled)
    {
        String[] requiredRoles = { "Admin", "Users", "Editors" };
        foreach (String role in requiredRoles)
        {
            if (!Roles.RoleExists(role))
            {
                Roles.CreateRole(role);
            }
        }
        string[] users = Roles.GetUsersInRole("Admin");
        if (users.Length == 0)
        {
            // create admin user
            MembershipCreateStatus status;
            Membership.CreateUser("admin", "CHANGE_ME", "admin@localhost",
                "Favorite color?", "green", true, out status);
            if (MembershipCreateStatus.Success.Equals(status))
            {
                Roles.AddUserToRole("admin", "Admin");
            }
        }
        else
    }
}

```

```
        {  
            LogMessage("Unable to create admin user: " + status, true);  
        }  
    }  
}
```

Summary

This chapter covered how to prepare your environment for working with ASP.NET websites and how to configure the link to the database. You learned how to configure and manage the provider services, including adding users and roles programmatically so a new website can be managed immediately after it is deployed.



Data Model Choices

The ASP.NET 2.0 data model allows for many methods to get the data from the database to the Web Form. The three top methods are DataSets, DataReaders, and DataObjects. This chapter reviews each of these options as well as the Data Access Application Block, which is a part of the Enterprise Library.

This chapter covers the following:

- Data Access Application Block
- Data Access code snippets
- Sample Person database
- Performance and ViewState considerations
- Typed DataSet
- Nontyped DataSet
- DataReader
- Subsets and sorting with ranges

There is not just one way to work with data in .NET. There are many distinct and sometimes intermingling ways, which give you a seemingly overwhelming set of choices. Although you may get by using a limited set of the available features, you will find that as your knowledge of your options deepens, you can come up with more-streamlined approaches that reduce the amount of work you need to do to get the job done. That translates quickly into higher productivity and less code to maintain.

And although the clever examples documented on the Microsoft Developer Network (MSDN) offer some amazing solutions, they really are simple examples. This chapter digs beyond the simplistic by throwing a wrench into the works and showing you how to get past it.

The Data Access Application Block

The Microsoft Patterns & Practices group provides a set of modules called Application Blocks, which give developers additional tools to work with the .NET framework. One of the modules is the Data Access Application Block, which provides a set of methods that consolidate the work you would normally have to do in order to work with the database. This layer of abstraction simplifies what can otherwise be a cumbersome task.

One goal of the Data Access Application Block is to give the developer an interface that is not specific to the underlying database. Software using this module will work with SQL Server and SQL Server CE as well as Oracle without modification of the C# code. This module also handles common tasks you must do if you are just using ADO.NET, such as opening and closing each database connection. Various tasks such as this one are handled automatically.

In addition to this alternate interface, the Enterprise Library includes configuration tools that can modify your web configuration file for you. In the case of the Data Access Application Block, such a configuration tool is unnecessary because all you need to configure is the connection string you will use to connect to the database. However, the configuration tool can set the default connection string to be used when it is not specified explicitly in the code.

At the core of the Data Access Application Block is the Database object, which abstracts away much of the complexity of the .NET framework related to database communications. You run your database commands through the Database object by using the DbCommand from the System.Data.Common namespace to do everything from selects, inserts, updates, and deletes to calling stored procedures that carry out more-complex tasks.

The majority of examples throughout this book use the Data Access Application Block. It is an easy-to-use interface and should be used to build your data access layer if you choose to build that layer manually.

To use the Enterprise Library, you will have to download it from the Microsoft Patterns & Practices website. When you run the installation, you may want to change the installation directory so that it is placed in your common folder, such as D:\Projects\Common\Microsoft Enterprise Library 3.1. As a part of the installer, you are given the option to install and compile the source. You can place the source in the same folder, as shown in Figure 2-1. When you build from the source, it will create assemblies that are not signed, so you will need to update the projects to use your own key if you plan to deploy the assemblies to the global assembly cache (GAC) or use them with projects that are strongly signed. Otherwise, you can use the assemblies in the bin folder, which are signed by Microsoft.

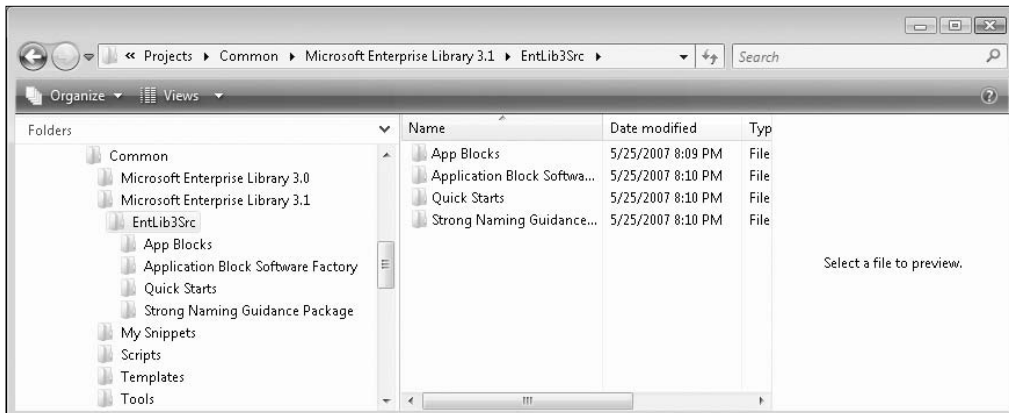


Figure 2-1. Enterprise Library 3.1 in the Common directory

The Enterprise Library is made up of many assemblies for all the Application Blocks. You do not need to reference every one of these assemblies to make use of just a single module. In the case of the Data Access Application Block, you need only three assemblies: Microsoft.

Practices.EnterpriseLibrary.Common.dll, Microsoft.Practices.ObjectBuilder.dll, and Microsoft.Practices.EnterpriseLibrary.Data.dll.

When I build my data access layer, I do so with a class library instead of placing all that code in the App_Code folder of an ASP.NET Website Project. Doing so allows it to be used as a dependency for multiple websites as well as console and desktop applications. It also makes it easy to version-control the data layer and run unit tests against it. When it is built inside the App_Code folder, it cannot be used externally, which makes versioning and unit testing difficult.

GUIDANCE AUTOMATION TOOL

The Enterprise Library is just one product of the Patterns & Practices team. The team also produces the Guidance Automation Extensions for Visual Studio, which uses templates and recipes for building applications that conform to their recommendations. Within the WCF and Web Service Software Factories is the Data Access Guidance Package, which can generate code for your data access layer.

Whenever you start work on an application, you do not want to be slowed down with all the tedious work of setting up your database with the various tables, stored procedures, indexing, and constraints only to spend a significant amount of time writing the intermediate layer between the database and the front end of the application. To speed up the process of creating all that code, you can use code snippets. This powerful feature of Visual Studio 2005 (and Visual Studio 2008) allows you to quickly select a template and fill in placeholders so you can avoid manually writing each line of code (typically, boilerplate code). Code snippets also help you avoid coding errors due to typos.

Data Access Code Snippets

I have created five code snippets to assist with writing code that uses the Data Access Application Block. In addition to adding lines of code to your class, the snippets can also specify the required namespace statements for the new block of code. The namespace imports work only in VB code, but the references will work if the assemblies can be found by Visual Studio. If you add the necessary assembly references to your project before adding one of the following code snippets, Visual Studio will automatically add the using statements to your class.

My collection of data access code snippets is as follows:

- Data Access Types
- Database Creation
- DataSet Method
- DataReader Method
- Nonquery Method

The code for these snippets follows in Listings 2-1 through 2-5.

Listing 2-1. *Data Access Types*

```

<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>1 Type Declarations</Title>
      <Shortcut>da1</Shortcut>
      <Description>Data Access Types</Description>
      <Author>Brennan Stehling</Author>
      <SnippetTypes>
        <SnippetType>Expansion</SnippetType>
      </SnippetTypes>
    </Header>
    <Snippet>
      <References>
        <Reference>
          <Assembly>Microsoft.Practices.EnterpriseLibrary.Common.dll</Assembly>
        </Reference>
        <Reference>
          <Assembly>Microsoft.Practices.EnterpriseLibrary.Data.dll</Assembly>
        </Reference>
        <Reference>
          <Assembly>Microsoft.Practices.ObjectBuilder.dll</Assembly>
        </Reference>
      </References>
      <Imports>
        <Import>
          <Namespace>System.Data</Namespace>
        </Import>
        <Import>
          <Namespace>System.Data.Common</Namespace>
        </Import>
        <Import>
          <Namespace>Microsoft.Practices.EnterpriseLibrary.Data</Namespace>
        </Import>
      </Imports>
      <Code Language="CSharp" Kind="type decl" Delimiter="$">
private Database db;

</Code>
    </Snippet>
  </CodeSnippet>
</CodeSnippets>
  </CodeSnippet>
</CodeSnippets>

```

The Data Access Types snippet simply declares a reference for the Database object that is used in the data access methods. This initial code snippet specifies the imports and references that are required for all the code snippets. Although C# does not automatically include the imports as in VB, I have included them with the expectation that a future release of Visual Studio will be able to use them.

Listing 2-2. Database Creation

```
<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>2 Database Creation</Title>
      <Shortcut>da2</Shortcut>
      <Description>Data Access Database Creation</Description>
      <Author>Brennan Stehling</Author>
    </Header>
    <Snippet>
      <Imports>
        <Import>
          <Namespace>Microsoft.Practices.EnterpriseLibrary.Data</Namespace>
        </Import>
      </Imports>
      <Declarations>
        <Literal Editable="true">
          <ID>connectionStringName</ID>
          <ToolTip>Connection String Name</ToolTip>
          <Default>db</Default>
          <Function>
            </Function>
          </Literal>
        </Declarations>
        <Code Language="CSharp" Kind="method body">
          <![CDATA[
            db = DatabaseFactory.CreateDatabase("$connectionStringName$");
          ]]></Code>
        </Snippet>
      </CodeSnippet>
    </CodeSnippets>
```

The Database Creation snippet creates the database instance. It should be placed in the constructor so that it needs to be initialized only once. The connection string is the only placeholder defined here. You should set this to the name of the connection string to use for this instance of the Database object.

Listing 2-3. *DataSet Method*

```

<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>3 Get DataSet Method</Title>
      <Shortcut>da3</Shortcut>
      <Description>Data Access DataSet Method</Description>
      <Author>Brennan Stehling</Author>
    </Header>
    <Snippet>
      <Imports>
        <Import>
          <Namespace>Microsoft.Practices.EnterpriseLibrary.Data</Namespace>
        </Import>
      </Imports>
      <Declarations>
        <Literal Editable="true">
          <ID>methodName</ID>
          <ToolTip>Method Name</ToolTip>
          <Default>GetDataSet</Default>
          <Function>
            </Function>
        </Literal>
        <Literal Editable="true">
          <ID>sproc</ID>
          <ToolTip>Stored Procedure</ToolTip>
          <Default>GetDataSet</Default>
          <Function>
            </Function>
        </Literal>
      </Declarations>
      <Code Language="CSharp" Kind="method decl">
        <![CDATA[

public DataSet $methodName$()
{
    DataSet ds = new DataSet();

    using (DbCommand dbCmd = db.GetStoredProcCommand("$sproc$"))
    {
        //db.AddInParameter(dbCmd, "@Parameter1", DbType.String, String.Empty);
        //db.AddOutParameter(dbCmd, "@Parameter2", DbType.String, 0);

        ds = db.ExecuteDataSet(dbCmd);
        //Object outputParameter =

```



```

        //db.GetParameterValue(dbCmd, "@OutputParameter");
    }

    //return the results
    return ds;
}]]></Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>

```

The `DataSet` method returns a `DataSet` after filling it with the results of a stored procedure call. The placeholders are for the name of the method and the name of the stored procedure. The snippet also includes a few lines for input and output parameters. These are in place to show how to add input and output parameters. These lines can be deleted if the stored procedure that is called does not take parameters. This snippet references the `DbCommand` variable with the `using` statement, which was introduced as a part of C# 2.0. It ensures that the `DbCommand` is disposed of at the end of the block. You will notice that it is not necessary to open and close the database connection here.

Listing 2-4. *DataReader Method*

```

<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>4 Get DataReader Method</Title>
      <Shortcut>da4</Shortcut>
      <Description>Data Access DataReader Method</Description>
      <Author>Brennan Stehling</Author>
    </Header>
    <Snippet>
      <Imports>
        <Import>
          <Namespace>Microsoft.Practices.EnterpriseLibrary.Data</Namespace>
        </Import>
      </Imports>
      <Declarations>
        <Literal Editable="true">
          <ID>methodName</ID>
          <ToolTip>Method Name</ToolTip>
          <Default>GetDataReader</Default>
          <Function>
            </Function>
          </Literal>
        <Literal Editable="true">
          <ID>sproc</ID>
          <ToolTip>Stored Procedure</ToolTip>

```

```

        <Default>GetDataReader</Default>
        <Function>
        </Function>
    </Literal>
</Declarations>
<Code Language="CSharp" Kind="method decl">
    <![CDATA[

public IDataReader $methodName$()
{
    IDataReader dr = null;

    using (DbCommand dbCmd = db.GetStoredProcCommand("$sproc$"))
    {
        //db.AddInParameter(dbCmd, "@Parameter", DbType.String, String.Empty);

        dr = db.ExecuteReader(dbCmd);
    }

    //return the results
    return dr;
}]]></Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>

```

The DataReader snippet does exactly the same work as the DataSet snippet, except that the DataReader snippet returns a DataReader object.

Listing 2-5. Nonquery Method

```

<?xml version="1.0" encoding="utf-8"?>
<CodeSnippets
  xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <CodeSnippet Format="1.0.0">
    <Header>
      <Title>5 Execute Nonquery</Title>
      <Shortcut>da5</Shortcut>
      <Description>Data Access Nonquery Method</Description>
      <Author>Brennan Stehling</Author>
    </Header>
    <Snippet>
      <Imports>
        <Import>
          <Namespace>Microsoft.Practices.EnterpriseLibrary.Data</Namespace>
        </Import>
      </Imports>
      <Declarations>

```

```

<Literal Editable="true">
  <ID>methodName</ID>
  <ToolTip>Method Name</ToolTip>
  <Default>SaveData</Default>
  <Function>
  </Function>
</Literal>
<Literal Editable="true">
  <ID>spc</ID>
  <ToolTip>Stored Procedure</ToolTip>
  <Default>SaveData</Default>
  <Function>
  </Function>
</Literal>
</Declarations>
<Code Language="CSharp" Kind="method decl">
  <![CDATA[

public void $methodName$(
{
    using (DbCommand dbCmd = db.GetStoredProcCommand("$spc$"))
    {
        //db.AddInParameter(dbCmd, "@Parameter", DbType.String, 0);
        //db.AddOutParameter(dbCmd, "@Parameter2", DbType.String, 0);

        db.ExecuteNonQuery (dbCmd);
        //Object outputParameter =
        //db.GetParameterValue(dbCmd, "@OutputParameter");
    }
}]]></Code>
</Snippet>
</CodeSnippet>
</CodeSnippets>

```

The Nonquery snippet does not return a `DataSet` or a `DataReader`. It can be used when doing an insert, an update, or a delete call into the database. It can also be used when simply pulling output parameters from the result of an executed command. In each of the three preceding method snippets (Listings 2-3, 2-4, and 2-5), the line after the execution line is a commented line showing how an output parameter value is pulled from the executed database command. Initially such a parameter is a generic object, but if the output value is `DbType.Int32`, the variable can be an `int` with the value cast as `int`. When running an insert command, it is common practice to return the primary key value of the newly inserted record.

To use code snippets, you must first add them to Visual Studio with the Code Snippet Manager. Place the preceding code snippets in a folder under `D:\Projects\Common\Templates\My Snippets` in a folder called `Data Access`. Then click **Tools ► Code Snippet Manager**. This brings up the Code Snippet Manager. Click the **Add** button and select the `My Snippets` folder. Then click **OK**. This makes these code snippets available to you in the editor.

With the code snippets in place, you can access them in three ways. You can right-click the editor, select Insert Snippet, and use the selection menu to get to the snippet you want to insert. Alternatively, you can use the hot keys Ctrl+K, Ctrl+X to pull up the menu. The fastest way is to use the shortcut specified by the snippet. The five snippets are marked as da1, da2, and so forth. Entering **da1** and pressing the Tab key twice will include that snippet. Figure 2-2 shows the selection menu for code snippets.

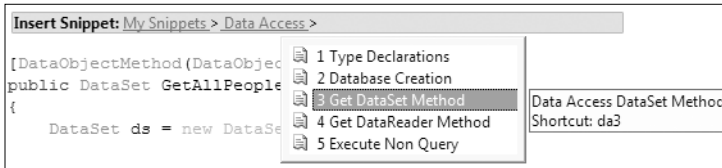


Figure 2-2. Code Snippet selection menu

After you select a snippet, the code is placed into the editor and the placeholders are highlighted. You can press Tab to move from placeholder to placeholder. There are default values in each placeholder. You can change the text in each marked placeholder and tab to the next value until you are finished. Then press Enter to accept the code snippet. In a matter of seconds, you can have a new method that returns a DataSet from a call to a stored procedure.

COMMON FOLDER ADDITIONS

These code snippets can be added to your Common folder in the My Snippets subfolder. (D:\Projects\Common\Templates\My Snippets). After they are in place, you can set up Visual Studio to reference them to be used in all your projects.

Sample Database

The examples shown through the rest of this chapter use a database holding two tables, Person and Location, as shown in Figure 2-3. Each record in the Person table references a record in the Location table with a foreign key constraint. A random and significantly sized set of data is then loaded into the Person table to allow for a distribution of values across the three columns: FirstName, LastName, and BirthDate. The foreign key reference is LocationId.

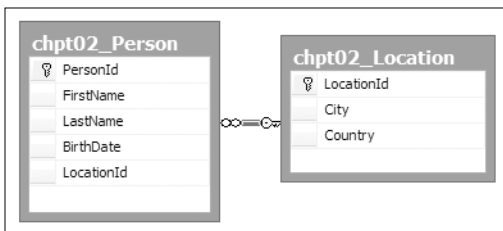


Figure 2-3. Person and Location tables

With a single table, it is trivial to drop a table from the Server Explorer onto a Typed DataSet Designer to generate the basic CRUD methods: Create, Read, Update, and Delete. By adding the secondary table, additional work is necessary to make the following examples work. A simple `SELECT * FROM Table1` will not be sufficient to bring two tables together. Instead, you can use a stored procedure to get all the desired data. The script in Listing 2-6 creates this stored procedure.

Listing 2-6. *chpt02_GetAllPeople.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'chpt02_GetAllPeople')
    BEGIN
        DROP Procedure chpt02_GetAllPeople
    END

GO

CREATE Procedure dbo.chpt02_GetAllPeople
AS

SELECT p.PersonId,p.FirstName,p.LastName,p.BirthDate,l.City,l.Country
FROM chpt02_Person AS p
JOIN chpt02_Location AS l on l.LocationId = p.LocationId

GO

GRANT EXEC ON chpt02_GetAllPeople TO PUBLIC
GO
```

Trivial Data Examples

It is amusing to call some data examples *trivial* in ASP.NET because they really are pulling off a complex task with little or no code. Simply dropping a table from the Server Explorer onto the design surface of a Web Form will automatically create a GridView and associate it with an SqlDataSource. The SqlDataSource is immediately configured with the SQL necessary to select, insert, update, and delete rows in that table. At all launch events for .NET 2.0, this sort of example was used to show the power of the ASP.NET 2.0 data model.

As you change properties on the GridView to allow for paging, sorting, selecting, and editing, it all just works. And more than that, it works without you writing any code. Listing 2-7 shows all the markup created after the Person table from the sample database is dragged onto a page.

Listing 2-7. *TrivialExample.aspx with SqlDataSource*

```
<%@ Page Language="C#" MasterPageFile="~/Site.master"
    AutoEventWireup="true" CodeFile="TrivialExample.aspx.cs"
    Inherits="TrivialExample" Title="Trivial Example" %>
<asp:Content
```

```

ID="Content1" ContentPlaceHolderID="ContentPlaceHolder1" Runat="Server">
  <asp:GridView
    ID="GridView1" runat="server" AllowPaging="True" AllowSorting="True"
    AutoGenerateColumns="False" DataSourceID="SqlDataSource1"
    EmptyDataText="There are no data records to display.">
    <EmptyDataTemplate>
      <strong>No Data</strong>
    </EmptyDataTemplate>
  </asp:GridView>
  <asp:SqlDataSource ID="SqlDataSource1" runat="server"
    ConnectionString="<%$ ConnectionStrings:chpt02 %>"
    ProviderName="<%$ ConnectionStrings:chpt02.ProviderName %>"
    DeleteCommand="DELETE FROM [chpt02_Person] WHERE [PersonId] = ➡
@PersonId"
    InsertCommand="INSERT INTO [chpt02_Person] ([FirstName], ➡
[LastName], [BirthDate], [LocationId]) VALUES (@FirstName, ➡
@LastName, @BirthDate, @LocationId)"
    SelectCommand="SELECT [PersonId], [FirstName], [LastName], ➡
[BirthDate], [LocationId] FROM [chpt02_Person]"
    UpdateCommand="UPDATE [chpt02_Person] SET [FirstName] = ➡
@FirstName, [LastName] = @LastName, [BirthDate] = @BirthDate, ➡
[LocationId] = @LocationId WHERE [PersonId] = @PersonId">
    <InsertParameters>
      <asp:Parameter Name="FirstName" Type="String" />
      <asp:Parameter Name="LastName" Type="String" />
      <asp:Parameter Name="BirthDate" Type="DateTime" />
      <asp:Parameter Name="LocationId" Type="Int64" />
    </InsertParameters>
    <UpdateParameters>
      <asp:Parameter Name="FirstName" Type="String" />
      <asp:Parameter Name="LastName" Type="String" />
      <asp:Parameter Name="BirthDate" Type="DateTime" />
      <asp:Parameter Name="LocationId" Type="Int64" />
      <asp:Parameter Name="PersonId" Type="Int64" />
    </UpdateParameters>
    <DeleteParameters>
      <asp:Parameter Name="PersonId" Type="Int64" />
    </DeleteParameters>
  </asp:SqlDataSource>
</asp:Content>

```

The first concern I always have is that instantly you have inline SQL, right in the application layer, directly in a page. And as easy as that SQL was to create, it is not that easy to update it for changes later, which will eventually trip you up unless the database never changes or you understand everything that has just been generated. It is easy to imagine that the Person table would eventually have more columns (for example, for the middle initial and gender). And if the table started with just a Name column and was split into First Name and Last Name, you

would need to make some changes to the SQL shown in Listing 2-7; otherwise, that SQL would be invalid.

The `SqlDataSource` in Listing 2-7 has a `Refresh Schema` command on the Smart Tag, but it really is not as capable as you might hope. Changing `Last Name` to `Surname` will cause an error when you refresh the schema. And adding a new column for the `Middle Initial` will not add the new column to the `Select` and other commands. To make those adjustments, you have to choose the other option on the Smart Tag, `Reconfigure Datasource`. And when this is done, there are consequences with the `GridView`, such as resetting the columns that you may have already customized considerably and will not want changed by the automatically generated changes. With all of this in mind, it is best to leave this quick-and-dirty way of data binding to prototyping new ideas and adjust to use a more maintainable and architecturally sound approach as a project moves past the early stages. Fortunately, the fast prototypes created with an inline query by using an `SqlDataSource` can be replaced with an `ObjectDataSource` later, as the prototype starts to become the actual application. The columns produced by the `ObjectDataSource` simply have to match the columns from the `SqlDataSource`, while your `GridView` can remain unchanged.

Nontrivial Data Examples

In real-world applications, the trivial examples get you only so far, and soon you are required to work with more than one table at a time. Adding data from more than one table to be shown as well as edited in a `GridView` or other databound control enters into the space of the nontrivial example.

In the sample `Person` database, the `Person` table has a relationship with the `Location` table through the `LocationId`. And in the `Location` table, the `City` and `Country` values define the location where the person lives. When displaying a `GridView` of all the people in the database, it will be much more intuitive to show the city and country instead of the `LocationId`. The `chpt02_GetAllPeople` stored procedure shown previously pulls these values together so they can be used as if they were in just a single table in the database from the application's point of view. Unfortunately, you cannot drag a stored procedure from the `Server Explorer` window onto the design surface of a Web Form as you can with the tables. Instead, you must drag a `GridView` from the `Toolbox` and configure it with a `datasource`. When configured as an `SqlDataSource`, the stored procedure can be selected, but the `Insert`, `Update`, and `Delete` commands will not be offered as options in the `datasource` wizard. It is still possible to get all the features of the single table example in the previous sections, but providing all the features that a `GridView` needs will require more than just this one stored procedure configured with a `Typed DataSet`. An example of how to do this is covered in detail in Chapter 3.

Typed DataSet

Starting with the basic stored procedure in Listing 2-6, you can build a customized `Typed DataSet`. You simply go to the class library project you have prepared to act as your data access layer and add a `DataSet` to it called `PersonDataSet.xsd`. Then add a `TableAdapter` and follow the steps provided by the wizard to add the existing stored procedure defined here. You then rename the `DataTable` and `TableAdapter` to more user-friendly names such as `People` and

PeopleTableAdapter in an attempt to make the code that is generated with the Typed DataSet better resemble the object it is meant to represent. Figure 2-4 shows what the People Typed DataSet looks like, and Figure 2-5 shows the properties from the FillAllPeople method.

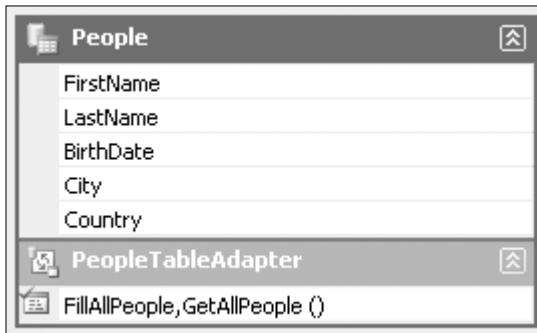


Figure 2-4. *People Typed DataSet*

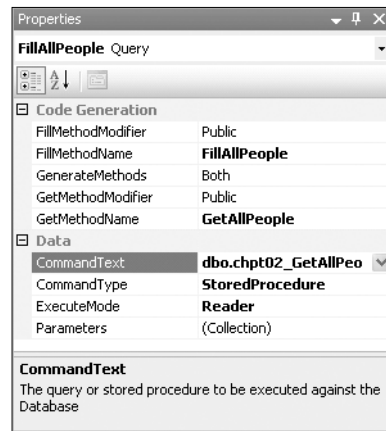


Figure 2-5. *FillAllPeople properties*

This Typed DataSet is defined as a part of a class library. With the website set to use this class library as a dependency, run the build for the website to ensure that the current assemblies used by the website have this newly created Typed DataSet. After the dependency is compiled, it is copied automatically into the bin directory of the website and is ready to be used in a Web Form. Simply drag a GridView control onto the design surface. Then use the Smart Tag to choose a datasource. Select a new ObjectDataSource. The available options will show the PeopleTableAdapter as it was named earlier. Set the select method to the GetAllPeople method. After the ObjectDataSource is set, you can return to the GridView and open the Smart Tag to enable paging and sorting. Now try out the page by right-clicking the page in the Solution Explorer and selecting View in Browser.

You may be pleasantly surprised that the page loads in a reasonable time and provides the sorting and paging functionality that you expect. The Typed DataSet also does a great job of cutting down the size of the ViewState. It is no wonder that the approach I've just described is strongly encouraged by MSDN documentation. However, this solution is not the only option or even the best option for all scenarios. Next you will explore a few alternatives.

RIGID TYPED DATASETS

When a Typed DataSet is put together on the XML Schema Definition (XSD) designer, it reads in all the schema information from the database and hard-codes the names of all the columns as well as the types and exact sizes. Changing a VARCHAR(10) to VARCHAR(11) can break the Typed DataSet. As soon as a record that is 11 characters long is inserted into the data and the Typed DataSet comes across this data, it will throw an exception due to the size constraint. If you are not careful, this problem could go unnoticed until it is pushed into a production environment and discovered at the worst time. To head off the problem, the updated column definition can be adjusted in the Typed DataSet manually in the Properties panel.

Nontyped DataSet

The Typed DataSet used in the previous section was built with an XSD file, which defines the various properties of the DataTables and TableAdapters. This is a fairly rigid model because the XSD file is edited primarily by using Visual Studio and the visual Typed DataSet Designer. It also locks in the database schema, which will most likely change over the life of a project. It can be helpful to use a more flexible option.

The stored procedure used by the Typed DataSet can be run directly to get the same columns that will generate a compatible DataSet at runtime. Instead of the columns redundantly being defined by the Typed DataSet and the stored procedure, they can be defined by just the stored procedure.

In the same class library, you create a class called `PersonDomain` and start filling in the code by using the code snippets defined earlier in this chapter. First you add the reference to the Database object and then add the initialization for that reference to the constructor (see Listing 2-8).

Listing 2-8. Database Initialization

```
/// <summary>
/// This is used as a global connection for database connectivity
/// </summary>
private Database db;

public PersonDomain()
{
    db = DatabaseFactory.CreateDatabase("chpt02");
}
```

Next you add the method in Listing 2-9, which calls the same stored procedure as the Typed DataSet.

Listing 2-9. GetAllPeopleDataSet

```
[DataObjectMethod(DataObjectMethodType.Select)]
public DataSet GetAllPeopleDataSet()
{
    DataSet ds = new DataSet();

    using (DbCommand dbCmd =
        db.GetStoredProcCommand("chpt02_GetAllPeople"))
    {
        ds = db.ExecuteDataSet(dbCmd);
    }

    //return the results
    return ds;
}
```

The simple method in Listing 2-9 calls the `chpt02_GetAllPeople` stored procedure and returns the generated `DataSet` with all the same columns as the `Typed DataSet`. The method also includes the `DataObjectMethod` attribute, which indicates that the associated method acts as a `Select` method. In addition to this method attribute, an attribute is also placed in the class declaration. For example:

```
[DataObject(true)]
public class PersonDomain
{
    ...
}
```

The attributes preceding the class and method declarations make these `Select` methods available to the `ObjectDataSource` configuration wizard used by the Web Form. Create a new Web Form and repeat the same steps to add the `GridView` to the page as done for the `Typed DataSet`. When configuring the `ObjectDataSource`, select the `PersonDomain` object and the `GetAllPeopleDataSet` method. Because the class is marked as a `DataObject` and the method is marked as a `Select` method, it is listed.

Also, instead of creating a new implementation from scratch, you can add a second `ObjectDataSource` to the same Web Form as the `Typed DataSet` by using the newly created class and method. Now you can use the Smart Tag on the `GridView` to select this new `ObjectDataSource`. Because the columns are completely compatible, you use either `datasource` interchangeably. Figure 2-6 shows a databound control with two configured `ObjectDataSources`.

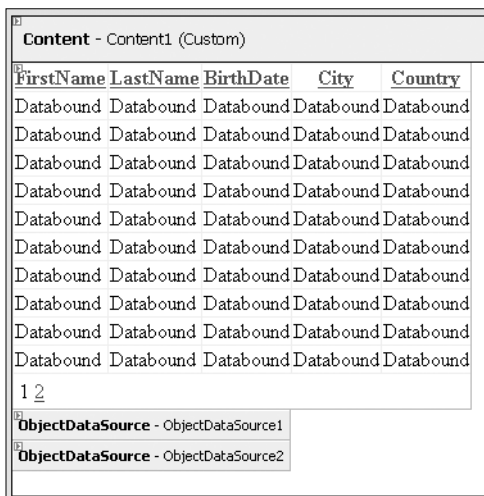


Figure 2-6. *Interchangeable datasources*

Both of the solutions I've described work sufficiently if there is not a great deal of data. But after there is a significant amount of data, the time necessary to transfer all that data between the database and application will become unreasonably long because the query is transferring all the data each time. The sheer size of the data being pulled from the database

will contribute to this sluggishness. Then shaping that data into the DataSet requires a certain amount of processing overhead. By using a DataReader, you can reduce that overhead.

DataReader

Unlike a DataSet, the DataReader does not offer functionality such as sorting, filtering, or even bidirectional movement on the result set. It has minimal features to iterate over the result rows and access the fields as needed. After you have passed a row, you cannot return to it with a DataKey or an index. Those seem like major limitations, but these limitations are meant to cut down on the overhead necessary with the DataSet, which maintains indexes on the data columns and holds the full result set in memory. In contrast, the DataReader is a lean and mean option with unique characteristics that make it ideal in some cases.

Listing 2-10 shows the code to add to PersonDomain, which again calls the same stored procedure but returns an IDataReader object as the result instead of a DataSet.

Listing 2-10. GetAllPeopleReader

```
[DataObjectMethod(DataObjectMethodType.Select)]
public IDataReader GetAllPeopleReader()
{
    IDataReader dr = null;

    using (DbCommand dbCmd = db.GetStoredProcCommand("chpt02_GetAllPeople"))
    {
        dr = db.ExecuteReader(dbCmd);
    }

    //return the results
    return dr;
}
```

Build the website, and this new Select method can be used with an ObjectDataSource. Either create a new Web Form to start from scratch, or add a new ObjectDataSource to the Web Form used for the Typed and Nontyped DataSet and associate it with the GridView. Naturally, you should select this new method. The new datasource shows the same data as the other two options, but unfortunately you will discover that the paging functionality does not work with the DataReader in its current form. Some changes must be made to allow for paging as well as sorting.

DataObject

These examples are all using the new DataObject introduced with .NET 2.0. With a Typed DataSet, the generated classes are marked as DataObjects and each of the methods are marked as one of the enumerations of DataObjectMethodType. One useful point about the DataObject and how databound controls work is that you can return a Typed DataSet, Nontyped DataSet, a DataReader, or a collection of Person objects as long as the column names or properties give the databound controls the values and types it is configured to use.

In fact, you can change the `PersonDomain` method returning a `DataSet` to return a collection of `Person` objects with properties called `FirstName`, `LastName`, `BirthDate`, `City`, and `Country` as they are defined by the stored procedure. The `GridView` will happily bind those properties just as easily as it would named columns on the `DataSet`.

What's the Downside?

Each of the approaches discussed in this chapter has a common shortcoming. Every time the page is displayed, the full set of data is pulled from the database—even if only ten rows are sent to the web client. When the database server is on the same machine as the web server, this is not a major issue. However, if the database server is on a different machine, especially when there is a very large amount of data, the time to transfer all that data will be a critical concern. Cutting down on the data moved from the database to the application is one of the optimizations we will explore in the next chapter.

Summary

This chapter covered the various choices available in the .NET framework for working with data. We looked at how each choice has its own unique advantages and disadvantages. As we go forward, we will leverage these options.



Database Management

Often the database is treated as an unchangeable resource as new application versions are released. Performance improvements are isolated to changes in the application layer instead of fully leveraging any improvements that could be made to the database. Database scripts can be managed as projects. This feature of Visual Studio is extremely useful but terribly underused. Every table, stored procedure, and database resource can be created and managed within these Database Projects. The solutions that hold your websites and other projects can be managed alongside your Database Projects. As each release is prepared, changes to the application and data layer can be adjusted as needed.

This chapter covers the following:

- Creating a Database Project
- Managing stored procedures
- Managing indexes and constraints
- Considering performance and stability
- Performing unit testing and continuous integration

Databases and the scripts tables and stored procedures are often not included within Visual Studio solutions along with the website and class library projects that we work with all the time.

By not including these scripts with a solution, they are disconnected and unmanaged. They are not even included in the same source-control system as the software that uses the databases. But Visual Studio Professional Edition supports Database Projects that you can include alongside your website and class library projects and organize with your source-control system.

Using Database Projects

Instead of only creating your tables and stored procedures in the database, you can extract those creation scripts and place them in Database Projects. Doing so aligns their changes directly with applications and manages them with the same source-control system. This allows you to deploy every application release with the necessary database changes.

Note To use Database Projects, you must have Visual Studio Professional Edition. The Standard and Express editions do not recognize this project type.

Visual Studio

In Visual Studio, the Database Projects are included in the other project types group when you add a new project to a solution. A file manifest keeps track of which files are in each of the folders, as with a class library project, and when you add a new table or stored procedure script to a folder, you are given a quick start template just as with Web Forms and classes. But the Database Project goes beyond the mundane work of holding your Structured Query Language (SQL) scripts. Each project can be associated with a database connection, with one being marked as the default. To run a stored procedure script on the default database, you simply right-click on the script and select Run On. The results of the script will be shown in the Output window. Figure 3-1 shows how to create a Database Project.

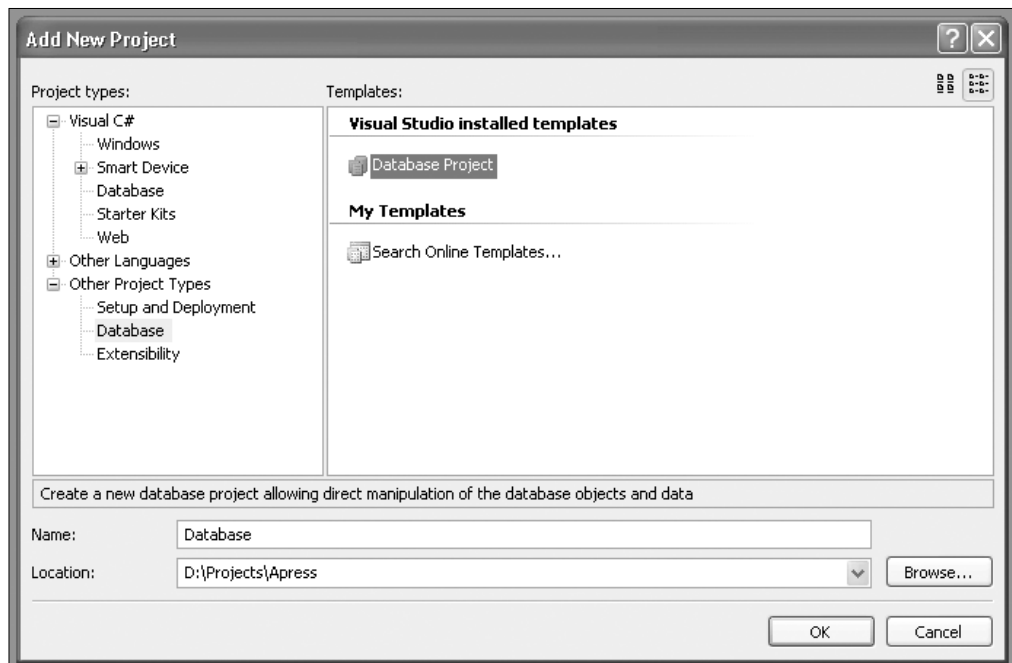


Figure 3-1. Database Projects

Database Projects are not used as often as they should be. All too often, tables and stored procedures are managed directly in the database, and changes to the schema or stored procedures are propagated either manually or with a third-party tool that generates scripts by analyzing the differences between these databases. By developing the scripts within Visual Studio, you will better understand the changes and have the control you need to plan for changes. And when the database scripts are managed within the solution, they will also be

version-controlled with your source-control system. At each release, you should be able to not only build your projects, but also return the database to a state that works with that release. Relying on manual changes or tools that adapt for unmanaged changes leads to confusion over the state of the database schema. And although these tools can eliminate the need to write change scripts yourself, they should not generate so much scripting that it is not possible to make sense of everything that is changing.

A silent feature built into the Run On process is dependency detection across table and stored procedure scripts. If you have several table scripts that must be run in a certain order, the dependency detection will determine the order when you highlight all the tables and click the Run On command. That is a real time-saver.

Visual Studio also has the Server Explorer, where you can view the tables and stored procedures deployed to the database. You can open a table to view and edit the contents of the table. You can even start a new query window within Visual Studio to query the databases listed in the Server Explorer. But for a little more power, many developers choose to use SQL Server Management Studio, which has a few additional features beyond what Visual Studio offers.

SQL Server Management Studio

SQL Server Management Studio does everything you need to manage table creation and modifications. It also works great with any sort of script. I have learned it is a valuable tool when it is paired properly with the Database Projects available in Visual Studio. I first create the stubs for the tables and stored procedure scripts in Visual Studio and then use Management Studio to build the tables and stored procedures. I do so with one table and one stored procedure at a time and test the changes at each step.

To create a table, I simply add a new table to the database with Management Studio and save it with the name I choose. Then I right-click the table in the Object Explorer and select Script Database As and then Create To and send it to the Clipboard. I move back to Visual Studio and paste the script in place and save the script. From that point, I can make adjustments to the script in Visual Studio, such as resizing the size of a VARCHAR or adding a new column. And to ensure that my script works after changes, I run it from Visual Studio against the development database. Right-click the script in the Solution Explorer and select Run On. The script will be run on the default database for the Database Project and report the result in the Output window.

Managing Stored Procedures

The process of creating stored procedures is very different from creating tables. In Visual Studio, you write your stored procedure script, run it against the database, and test it with any required parameters. As you refine the script, you will find that the process of deploying the stored procedure to the database to test it requires unnecessary overhead. Instead you can write the script in Management Studio and adjust it to work as a stored procedure.

I start by declaring the variables that the script will need and then set their values. Then I write the rest of the script, which makes use of those variables, as shown in Listing 3-1.

Listing 3-1. *Script to Select People by First and Last Name*

```
DECLARE @FirstName varchar(50)
DECLARE @LastName varchar(50)

SET @FirstName = 'John'
SET @LastName = 'Smith'

SELECT * FROM chpt03_Person
WHERE FirstName = @FirstName
AND LastName = @LastName
```

The preceding script shows how the variables are declared, set, and used within the script. After the script is working properly, it can be placed into the stored procedure template that was stubbed out in the Database Project (see Listing 3-2).

Listing 3-2. *Stored Procedure to Select People by First and Last Name*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND
    name = 'chpt03_GetPeopleByName')
    BEGIN
        DROP Procedure chpt03_GetPeopleByName
    END
GO

CREATE Procedure dbo.chpt03_GetPeopleByName
(
    @FirstName varchar(50),
    @LastName varchar(50)
}
AS

SELECT * FROM chpt03_Person
WHERE FirstName = @FirstName
AND LastName = @LastName

GO

GRANT EXEC ON chpt03_GetPeopleByName TO PUBLIC
GO
```

You can see how the declared variables are now used as parameters and the set commands are not necessary. The rest of the script follows unchanged. As your stored procedures grow more and more complex, this process will help simplify your work. A key difference is that Management Studio will be able to give you more descriptive and accurate warnings and errors when run as scripts instead of calls to a stored procedure.

CRUD PROCEDURES

CRUD is an acronym for *Create, Read, Update, and Delete*. The typical approach is to create stored procedures for each of these actions for every table in the database. With a fully normalized database structure, this means there would be many of these CRUD procedures that may not be very useful. By updating records in a piecemeal way, performance can suffer. By grouping CRUD functionality into planned-out stored procedures, it is possible to update multiple tables at a time without multiple trips to the database. As a result, there will not always be four stored procedures per database table.

Stored procedures do more than just return the results of a `Select` statement. A stored procedure may just set the value of an output parameter. In the case of saving data to the database, it may just return the key value of the saved record. Listing 3-3 shows how a person is saved.

Listing 3-3. *chpt03_SavePerson.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'chpt03_SavePerson')
    BEGIN
        DROP Procedure chpt03_SavePerson
    ENDGO

CREATE Procedure dbo.chpt03_SavePerson
(
    @FirstName varchar(50),
    @LastName varchar(50),
    @BirthDate datetime,
    @LocationId bigint,
    @PersonId bigint OUTPUT
)
AS
    INSERT into chpt03_Person
    (FirstName, LastName, BirthDate, LocationID)
    values (@FirstName, @LastName, @BirthDate, @LocationID)

    SET @PersonId = @@IDENTITY

GO

GRANT EXEC ON chpt03_SavePerson TO PUBLIC
GO
```

This stored procedure inserts a new person record and sets the output parameter for the `PersonId`. Notice that the `PersonId` parameter defined near the top has `OUTPUT` marking it as an output parameter. Having access to such values can become quite useful when you take advantage of them. The `Person` table has a relationship with the `Location` table and takes

LocationId as a parameter. An arbitrary value cannot be used, so a real value should be. Listing 3-4 shows the procedure of how to save a location.

Listing 3-4. *chpt03_SaveLocation.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND
    name = 'chpt03_SaveLocation')
    BEGIN
        DROP Procedure chpt03_SaveLocation
    END

GO

CREATE Procedure dbo.chpt03_SaveLocation
(
    @City [varchar](50),
    @Country [varchar](50),
    @LocationId bigint OUTPUT
)
AS

IF NOT EXISTS (
    SELECT * FROM chpt03_Location
    WHERE City = @City and Country = @Country
)
    BEGIN
        -- INSERT
        PRINT 'Inserting Location'
        INSERT into chpt03_Location
        (City,Country)
        values (@City, @Country)

        SET @LocationId = @@IDENTITY
    END
ELSE
    BEGIN
        -- get LocationId
        PRINT 'Location Exists'
        SET @LocationId = (
            SELECT LocationId FROM chpt03_Location
            WHERE City = @City and Country = @Country
        )
    END

GO

GRANT EXEC ON chpt03_SaveLocation TO PUBLIC
GO
```

This stored procedure includes an additional location to first check whether the city and country combination already exists. The procedure then either runs an INSERT or a SELECT command to get the value of the LocationId, which is the output parameter. When saving a Location and Person, this stored procedure can be called to get the LocationId.

Finally, a single stored procedure, shown in Listing 3-5, can be created to combine all this work into a single call to the database.

Listing 3-5. *chpt03_SavePersonWithLocation.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND
    name = 'chpt03_SavePersonWithLocation')
    BEGIN
        DROP Procedure chpt03_SavePersonWithLocation
    END

GO

CREATE Procedure dbo.chpt03_SavePersonWithLocation
(
    @FirstName varchar(50),
    @LastName varchar(50),
    @BirthDate datetime,
    @City [varchar](50),
    @Country [varchar](50),
    @LocationId bigint OUTPUT,
    @PersonId bigint OUTPUT
)
AS

EXEC chpt03_SaveLocation @City, @Country, @LocationId OUTPUT

EXEC chpt03_SavePerson @FirstName, @LastName, @BirthDate, @LocationId, ➡
    @PersonId OUTPUT

GO

GRANT EXEC ON chpt03_SavePersonWithLocation TO PUBLIC
GO
```

Fewer trips to the database makes for a more efficient application. It also gives you some flexibility with how the tables are organized in the database. Schema changes could be completely encapsulated behind the stored procedures so that the applications using the database require no changes.

Managing Indexes and Constraints

As you add tables that have foreign key constraints, you will start to run into some complications. You cannot drop a table if there are foreign key dependencies preventing that action. To get past this problem, I create a folder in each Database Project called Constraints, which has a script to add all the foreign key constraints and another to remove them. I also often have a script that will purge all data and fill in some initial sample data. When I change several table scripts, I can remove the constraints, run the table scripts, and repopulate the database with sample data so I can test the stored procedures.

There are no templates for Database Projects to check indexes and foreign keys, so you have to start from scratch. The best feature of table and stored procedure templates is the check for the existence of an item that needs to be dropped before it is re-created so that you can run the script without errors.

For the Person and Location tables, there are just three indexes. The scripts to manage the indexes are created by using Management Studio. The scripts were first added to the tables, and before saving the change I used the Generate Change Script button on the taskbar, which displays the script used to make an individual change. I copied and pasted that script into the script to add indexes. This script does not check whether the index already exists, so the query to check for these indexes must be included before the CREATE INDEX command to be run successfully each time. Listing 3-6 shows the script that removes indexing.

Listing 3-6. *Remove Indexing.sql*

```
BEGIN TRANSACTION
GO

IF EXISTS
    (SELECT * FROM sysindexes AS i
    JOIN sysobjects AS o on i.id = o.id
    WHERE o.name = 'chpt03_Location' and i.name =
'IX_chpt03_Location_City')
    BEGIN
        PRINT 'Dropping index IX_chpt03_Location_City'
        DROP INDEX IX_chpt03_Location_City ON dbo.chpt03_Location
    END
GO

IF EXISTS
    (SELECT * FROM sysindexes AS i
    JOIN sysobjects AS o on i.id = o.id
    WHERE o.name = 'chpt03_Location' and i.name =
'IX_chpt03_Location_Country')
    BEGIN
        PRINT 'Dropping index IX_chpt03_Location_Country'
        DROP INDEX IX_chpt03_Location_Country ON dbo.chpt03_Location
    END
GO
```

```

IF EXISTS
    (SELECT * FROM sysindexes AS i
    JOIN sysobjects AS o on i.id = o.id
    WHERE o.name = 'chpt03_Person' and i.name =
    'IX_chpt03_Person_BirthDate')
    BEGIN
        PRINT 'Dropping index IX_chpt03_Person_BirthDate'
        DROP INDEX IX_chpt03_Person_BirthDate ON dbo.chpt03_Person
    END
GO

```

```

COMMIT

```

The `sysindexes` and `sysobjects` tables provide the information needed to detect whether an index already exists so the `DROP` command can be run conditionally to avoid any errors. Removing indexing makes the process of adding lots of data to tables much faster because the indexes do not have to be checked and maintained for each insert when it is removed. After the index can be added back, the script in Listing 3-7 can be run. After the indexes are back in place, queries should run much faster when there is a lot of data.

Listing 3-7. *AddIndexing.sql*

```

BEGIN TRANSACTION
GO

IF NOT EXISTS
    (SELECT * FROM sysindexes AS i
    JOIN sysobjects AS o on i.id = o.id
    WHERE o.name = 'chpt03_Location' and i.name =
    'IX_chpt03_Location_City')
    BEGIN
        PRINT 'Adding index IX_chpt03_Location_City'
        CREATE NONCLUSTERED INDEX
        IX_chpt03_Location_City ON dbo.chpt03_Location
        (
            City
        ) WITH( STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
            ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
        ON [PRIMARY]
    END
GO

IF NOT EXISTS
    (SELECT * FROM sysindexes AS i
    JOIN sysobjects AS o on i.id = o.id
    WHERE o.name = 'chpt03_Location' and i.name =
    'IX_chpt03_Location_Country')
    BEGIN

```

```

        PRINT 'Adding index IX_chpt03_Location_Country'
        CREATE NONCLUSTERED INDEX
IX_chpt03_Location_Country ON dbo.chpt03_Location
    (
        Country
    ) WITH( STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
        ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
ON [PRIMARY]
    END
GO

IF NOT EXISTS
    (SELECT * FROM sysindexes AS i
    JOIN sysobjects AS o on i.id = o.id
    WHERE o.name = 'chpt03_Person' and i.name =
'IX_chpt03_Person_BirthDate')
    BEGIN
        PRINT 'Adding index IX_chpt03_Person_BirthDate'
        CREATE NONCLUSTERED INDEX IX_chpt03_Person_BirthDate
ON dbo.chpt03_Person
    (
        BirthDate
    ) WITH( STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
        ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON)
ON [PRIMARY]
    END
GO

COMMIT

```

After the indexes are easily managed, it is time to move on to the foreign key constraints. In the case of these constraints, there also is no starter template that checks for existing constraints before dropping them. The script in Listing 3-8 removes the single foreign key between the Person and Location tables.

DEFRAGGING INDEXES

Over time indexes can become fragmented. A fragmented index will take longer to scan than a freshly created index, so it is necessary to occasionally use the REORGANIZE command to clean up indexes. That command does not fully refresh an index but does improve performance. The REBUILD command can be used to fully refresh the index. These tasks are best left to your database administrator (DBA). As you start to notice performance degradation, you can consider this tuning technique.

Listing 3-8. *RemoveConstraints.sql*

```
IF EXISTS (SELECT * FROM dbo.sysobjects
WHERE id = object_id(N'[dbo].[FK_chpt03_Person_chpt03_Location]')
and OBJECTPROPERTY(id, N'IsForeignKey') = 1)
ALTER TABLE [dbo].[chpt03_Person]
DROP CONSTRAINT FK_chpt03_Person_chpt03_Location
```

And then to add the constraint back, the constraint in Listing 3-9 is run.

Listing 3-9. *AddConstraints.sql*

```
ALTER TABLE [dbo].[chpt03_Person] WITH CHECK ADD
CONSTRAINT [FK_chpt03_Person_chpt03_Location] FOREIGN KEY([LocationId])
REFERENCES [dbo].[chpt03_Location] ([LocationId])
```

Now that roadblock is easily removed whenever the tables need to be readjusted for a change to the tables. If you want to add a column, you can update the table script, remove the constraints, run the table script, and add the constraints back.

Performance Considerations

Before I get to the next example, there are some considerations to review. Where performance is concerned, there are many factors to consider. To tune a slow application using a database, there are a few basic steps you can take such as adding indexes to your tables and pulling only the necessary data. The indexing will help assemble the results more quickly, while pulling less data will reduce the amount of data transferred. In the case of very large result sets, leaving off some columns will add up to a measurable difference. There are many other changes that can be made to enhance performance, but these first two are the low-hanging fruit that will typically give the most benefit.

After you have pulled all data from the database into the web application to display on the Web Form, the databound values will be serialized to ViewState. With a very large amount of data, the result will be a very large file that the users of the website will have to download. For a GridView that is showing only 10 of 100 rows, the ViewState can still hold all the data for all 100 rows. And although the pages with such large sets of data will load quickly during development while you are working on the same computer as the web server, it will be painfully slow for a user accessing the website remotely. As a rule of thumb, the web page must be under 100 KB, if not under 50 KB.

I once worked on a website using a dynamic navigational menu that had ViewState and PostBack events covering several levels of the website hierarchy. It included a large number of links, which added to the size of the ViewState. The typical web page was nearly 300 KB, with over 90 percent of the page content being the ViewState. The original developer never knew about this problem as he worked on his laptop, and everything was always fast. He never checked the page size or how long it took to move from page to page from a remote server. This sort of problem should be caught early, when it is easier to fix.

You can tune your tables and queries all you like, but if the page is 300 KB and many of your users are still using dial-up or even low-end broadband speeds, they will find that your website is slow. In contrast, most major news sites and information portals are under 30 KB. In the following examples, the ViewState issue will be addressed and resolved with a simple strategy.

Note The time to transfer data is just one aspect of a slow web page. Large tables with many rows and cells add to the rendering time for a web page. If it takes less than a second to load the data for a page, it can still take a few more seconds for the web page to render a complex table.

It would be much more efficient to ask that the database return only the range of items that will be displayed, rather than have the database return all possible items. When only ten rows are shown, it is wasteful to transfer thousands of rows from the database to the web application. Fortunately, we can request just that subset, but we will need a new stored procedure that will work with our new goal. Listing 3-10 shows the script to create this special stored procedure.

Listing 3-10. *chpt03_GetPeopleSubSetSorted.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND
    name = 'chpt03_GetPeopleSubSetSorted')
    BEGIN
        DROP Procedure chpt03_GetPeopleSubSetSorted
    END

GO

CREATE Procedure dbo.chpt03_GetPeopleSubSetSorted
(
    @sortExpression    nvarchar(50),
    @startRowIndex     int,
    @maximumRows       int
)
AS

IF LEN(@sortExpression) = 0
    SET @sortExpression = 'PersonId'

-- reset to 1 based index
SET @startRowIndex = @startRowIndex + 1

-- build sql
DECLARE @sql nvarchar(4000)
SET @sql = 'SELECT PersonId,FirstName,LastName,BirthDate,City,Country
FROM
    (SELECT p.PersonId,p.FirstName,p.LastName,p.BirthDate,l.City,l.Country,
```



```

    ROW_NUMBER() over(ORDER BY ' + @sortExpression + ') AS RowNum
  FROM chpt03_Person AS p
  JOIN chpt03_Location AS l on l.LocationId = p.LocationId
) AS People
WHERE RowNum between ' + CONVERT(nvarchar(10), @startRowIndex) +
    ' and (' + CONVERT(nvarchar(10), @startRowIndex) + ' + '
    + CONVERT(nvarchar(10), @maximumRows) + ') - 1'

-- Execute the SQL query
EXEC sp_executesql @sql

GO

GRANT EXEC ON chpt03_GetPeopleSubSetSorted TO PUBLIC
GO

```

COMMON FOLDER ADDITIONS

The stored procedure in Listing 3-10 is a technique that you can apply many times to reduce load on the database while speeding up the application layer. This script can be placed in your Common folder in the Scripts subfolder to be referenced in future projects (D:\Projects\Common\Scripts\Database).

A lot is happening in this new stored procedure. It does not return every item in the Person table. Instead it uses input parameters @startRowIndex and @maximumRows to define a range of items to return. It also uses the @sortExpression input parameter to provide for sorting.

Introduced with SQL Server 2005 is the new Row_Number function. Given the order of a selection, the Row_Number function assigns a row number to each row. This row number is used to limit the scope of the returned items.

But the @startRowIndex and @maximumRows values have to come from somewhere. This information is given by the ObjectDataSource. When paging is enabled on the ObjectDataSource, the SelectMethod must reference a method that has these parameters. We must add such a method to the PersonDomain class. Listing 3-11 shows sample methods that use these additional parameters.

Listing 3-11. *GetPeopleSubSetSortedDataSet* Methods

```

[DataObjectMethod(DataObjectMethodType.Select)]
public DataSet GetPeopleSubSetSortedDataSet(int? startRowIndex, int? maximumRows)
{
    return GetPeopleSubSetSortedDataSet(null, startRowIndex, maximumRows);
}

[DataObjectMethod(DataObjectMethodType.Select)]
public DataSet GetPeopleSubSetSortedDataSet(

```

```

        string sortExpression, int? startRowIndex, int? maximumRows)
    {
        DataSet ds = new DataSet();
        if (String.IsNullOrEmpty(sortExpression))
        {
            sortExpression = "";
        }
        if (!startRowIndex.HasValue)
        {
            startRowIndex = 0;
        }
        if (!maximumRows.HasValue)
        {
            maximumRows = 0;
        }

        using (DbCommand dbCmd =
            db.GetStoredProcCommand("chpt03_GetPeopleSubSetSorted"))
        {
            db.AddInParameter(dbCmd, "@sortExpression", DbType.String, sortExpression);
            db.AddInParameter(dbCmd, "@startRowIndex", DbType.Int64, startRowIndex);
            db.AddInParameter(dbCmd, "@maximumRows", DbType.Int64, maximumRows);

            ds = db.ExecuteDataSet(dbCmd);
        }

        //return the results
        return ds;
    }

```

In Listing 3-11 there are two methods with the same name. The difference is the additional argument for `sortExpression` on the second method. You may recognize this technique as method overloading. Depending on whether the `ObjectDataSource` supports sorting, this parameter will be used. The first method simply calls the second method with a null value for the `sortExpression`, which leads to the `chpt03_GetPeopleSubSetSorted` stored procedure in Listing 3-10.

The `ObjectDataSource` has one other property beyond the `SelectMethod` property that we have been using so far and that we will use to make the solution work more efficiently. This property, called `SelectCountMethod`, is used to tell the databound control the total number of items the datasource returns. This value comes from another stored procedure. Listing 3-12 shows this stored procedure.

Listing 3-12. *chpt03_GetPeopleRowCount.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND ➡
    name = 'chpt03_GetPeopleRowCount')
BEGIN
    DROP Procedure chpt03_GetPeopleRowCount

```

```

END

GO

CREATE Procedure dbo.chpt03_GetPeopleRowCount
(
    @Count int OUTPUT
)
AS

SET @Count = (SELECT COUNT(*) AS [Count] FROM chpt03_Person)

GO

GRANT EXEC ON chpt03_GetPeopleRowCount TO PUBLIC
GO

```

This counts all records in the `Person` table and is used in the method shown in Listing 3-13.

Listing 3-13. *GetPeopleRowCount Method*

```

public long? GetPeopleRowCount()
{
    long? count = 0;

    using (DbCommand dbCmd = db.GetStoredProcCommand("chpt03_GetPeopleRowCount"))
    {
        db.AddOutParameter(dbCmd, "@Count", DbType.Int64, 0);

        db.ExecuteNonQuery(dbCmd);
        count = (long)db.GetParameterValue(dbCmd, "@Count");
    }

    return count;
}

```

Finally, the following `ObjectDataSource` in Listing 3-14 pulls it all together so it can be used by a databound control.

Listing 3-14. *ObjectDataSource1*

```

<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    TypeName="Chapter02.PersonDomain"
    SelectMethod="GetPeopleSubSetSortedDataSet"
    SelectCountMethod="GetPeopleRowCount"
    EnablePaging="True"
    SortParameterName="sortExpression">
</asp:ObjectDataSource>

```

With this `ObjectDataSource`, you can page through thousands of rows quickly and efficiently. It also reduces the amount of data moved from the database to the web application.

Stability Considerations

The link between the application and the database is often taken for granted. As a result, it is not given enough protection nor treated as a true integration point. As much as we would like a database query to be as reliable as looping over an array of objects in memory, it is not. Part of the disconnected nature of the database is the ability to change the application or database independently of the other, and despite our best efforts to keep them compatible, it is easy to change one without adjusting the other properly.

With a `Typed DataSet`, the schema is very strict. The slightest change on a single column will cause the application to break. Changing the name of a column will cause unwanted behavior on a `Nontyped DataSet`. When the relationship between the application and the database is treated as an integration point, the approach and assumptions change. And that change will allow for additional steps to be taken to ensure the stability of the integration. Later in this chapter, you will see what the options are for working with the data and what you can do to ensure that this important link stays reliable.

ISOLATING CHANGES

A primary goal of any software project should be to minimize the amount of maintenance. When a database used by several applications changes, it can generate a great deal of work to update all of the applications. But the changes could be isolated to a choke point, where all applications share a dependency. Consider placing your `Typed DataSets` and domain classes in a class library, which creates an assembly to be used by all the applications. When the database schema changes, the class library can be updated and deployed to each of the applications. In the case of a growing `VARCHAR`, it is easily absorbed into .NET because the `String` object is not concerned with the size and will not throw an exception if this class library suddenly starts returning longer `String` values. There may still be adjustments to be made, such as validation control properties, but this approach avoids updating a `Typed DataSet` reference in each application by isolating the change to a single point. And with this single point of access to the database, we have the perfect opportunity to test the class library to ensure that the changes in the database have been synchronized with the code.

Unit Tests for Data

Instead of manually testing every change each time a change is made, you can use unit tests, which can run a whole suite of tests automatically. These tests can confirm that each stored procedure and domain method do as they are requested and can give you an early warning when something does break. The early warning gives you the opportunity to correct the problem soon after the breaking change is made so you can more easily identify and fix it. If a problem were to go unnoticed until the application was being prepared for deployment, identifying the cause would be quite difficult, which would make it harder to make a correction.

The most popular unit-testing framework for .NET is NUnit. It is a simple framework used by placing attributes on classes and methods to indicate that they are part of a test suite. The

NUnit test runner will load an assembly, look for these attributes, and dynamically build a collection of tests to be run. In each test you will make calls into the database and check the return values with `Assert` statements, which will be either `succeed` or `fail`. When a test suite succeeds in all tests, the indicators are all green. But when a test fails, it is lit up in red.

After unit tests are in place and the test suite comprehensively covers your code base, each developer can run the tests prior to committing changes to the source-control system. Doing so will keep broken code out of the source-control system, which would spread to the other developers' systems as they pull down updates. But even when tests are run, the combination of changes from multiple developers can break the unit tests. To protect against this scenario, you can set up a build server to pull updates from source control, build the projects, and run the tests at regular intervals. This is commonly known as continuous integration.

Continuous Integration

The most popular continuous integration solution over the past few years has been CruiseControl.NET. It works with the unit-testing frameworks such as NUnit as well as a wide range of source-control systems. It can be configured to check the source-control system for updates every five minutes. When there is an update, CruiseControl.NET kicks off the build and testing process. The results of the build and tests are recorded with build reports, and the developer can get the results immediately by e-mail.

CruiseControl.NET also has a system tray application that stays in touch with the build server to report changes back to the developers. When a build is successful, the application pops up a notification bubble from the system tray. And when there is a failure, it displays a notification with the bad news. CruiseControl.NET also has a website that displays each of the configured projects with a list of all of the build reports. When a build fails, you can look at the latest build to see exactly what went wrong.

Not only does it show you the files that were changed for the latest build, but also the line numbers where the code broke. These details make it much easier to determine who broke the build and what was changed to cause the break. A side effect of having the automated build report failures is that it is much less personal, and the developer can simply volunteer to fix his problem right away. Many times I have seen the build fail because of a missing file that was not committed to source control with all the other changes. Committing that missing file quickly fixes the build, and everyone can get back to work. Because of such cases, it is best to commit your changes and request the build server to run the build immediately so you can ensure that the build is run successfully after your changes are included but before you head home for the day. That last thing you need is to be called back into work because you did not include a file with your latest changes.

The upcoming chapters cover specific examples for unit testing and continuous integration. They will become a regular part of your daily routine.

Summary

This chapter covered database management processes, from organizing the scripts in Database Projects to creating the scripts for managing stored procedures, indexes, and constraints. It concluded with a review of automated testing techniques that can be used to ensure that the application layer continues to work well with the database.



Databound Controls

There are many useful controls available in ASP.NET 2.0, which makes it very easy to work with data. Many of the examples provided by Microsoft show how you can quickly assemble a data-driven website with little or no code. But sometimes the real-world requirements push you beyond this safe space and demand a more complex solution. The solutions can still be elegant and work with very little code. You just need a deeper understanding of what you can do with the available controls and what you do with a more complex combination of these controls. And once you break up your approach into smaller components that work together, you can handle the more complicated requirements with manageable components.

This chapter covers the following:

- The DetailsView control
- The FormView control
- The GridView control
- Editing and validating fields
- Binding input parameters
- Embedding user controls
- Creating a databound control

By far the most commonly used databound control is the GridView control. It is a very powerful control that can show your data and provide add, update, and delete functionality you would expect from a rich data control. Beyond this one control, there are several other databound controls that provide the same display and editing features as GridView but do it in a different way. Let's look at those other controls first.

DetailsView

The DetailsView control shows a single record at a time. Like the GridView control, it features add, update, and delete functionality. It also has paging functionality. In contrast, the GridView shows the data in a tabular format so you can view multiple records at a time. The DetailsView is useful when there are more fields than you would reasonably want to show in a tabular view. This makes it better suited for editing.

Let's use the DetailsView control with the person and location data from the previous chapter. The first example is in Listing 4-1.

Listing 4-1. *DetailsView Example*

```

<asp:DetailsView ID="DetailsView1" runat="server"
    DataSourceID="ObjectDataSource1" AllowPaging="True"
    AutoGenerateRows="False" DataKeyNames="PersonId">
    <Fields>
        <asp:BoundField DataField="FirstName"
            HeaderText="First Name" SortExpression="FirstName" />
        <asp:BoundField DataField="LastName"
            HeaderText="Last Name" SortExpression="LastName" />
        <asp:BoundField DataField="BirthDate"
            HeaderText="Birth Date" SortExpression="BirthDate"
            DataFormatString="{0:MM/dd/yyyy}" HtmlEncode="False" />
        <asp:BoundField DataField="City"
            HeaderText="City" SortExpression="City" />
        <asp:BoundField DataField="Country"
            HeaderText="Country" SortExpression="Country" />
        <asp:CommandField ShowDeleteButton="True"
            ShowEditButton="True" ShowInsertButton="True" />
    </Fields>
</asp:DetailsView>

```

The columns selected here include First Name, Last Name, Birth Date, City, and Country. Each of these fields is automatically editable, and each record can be deleted. You can even add a new record. And with paging enabled, you can jump to each of the records. It is all as you would expect from such a data-editing control. I'll come back to the DetailsView control in the advanced examples throughout the rest of this chapter.

FormView

The FormView control starts out looking a lot like the DetailsView, but it is not as rigid. It may start out as a two-column layout, but it is a template control that can be changed completely. You may choose to group the First Name, Last Name, and Birth Date columns together in one block and the City and Country columns in another. When you view the code, you will see the various templates, including the ItemTemplate, which shows the data in read-only mode; you will also see all of the Label controls with the Bind method calls to fill each Label with the data from the DataSource. A sample FormView is shown in Listing 4-2.

Listing 4-2. *FormView Example*

```

<asp:FormView ID="FormView1" runat="server"
    AllowPaging="True" DataKeyNames="PersonId"
    DataSourceID="ObjectDataSource1">
    <EditItemTemplate>
        PersonId:
        <asp:Label ID="PersonIdLabel1" runat="server"
            Text='<%# Eval("PersonId") %>'></asp:Label><br />
        FirstName:
    
```



```

<asp:TextBox ID="FirstNameTextBox" runat="server"
    Text='<%# Bind("FirstName") %>'\>
</asp:TextBox><br />
LastName:
<asp:TextBox ID="LastNameTextBox" runat="server"
    Text='<%# Bind("LastName") %>'\>
</asp:TextBox><br />
BirthDate:
<asp:TextBox ID="BirthDateTextBox" runat="server"
    Text='<%# Bind("BirthDate") %>'\>
</asp:TextBox><br />
City:
<asp:TextBox ID="CityTextBox" runat="server"
    Text='<%# Bind("City") %>'\>
</asp:TextBox><br />
Country:
<asp:TextBox ID="CountryTextBox" runat="server"
    Text='<%# Bind("Country") %>'\>
</asp:TextBox><br />
<asp:LinkButton ID="UpdateButton" runat="server"
    CausesValidation="True" CommandName="Update" Text="Update">
</asp:LinkButton>
<asp:LinkButton ID="UpdateCancelButton" runat="server"
    CausesValidation="False" CommandName="Cancel" Text="Cancel">
</asp:LinkButton>
</EditItemTemplate>
<InsertItemTemplate>
    FirstName:
    <asp:TextBox ID="FirstNameTextBox" runat="server"
        Text='<%# Bind("FirstName") %>'\>
    </asp:TextBox><br />
    LastName:
    <asp:TextBox ID="LastNameTextBox" runat="server"
        Text='<%# Bind("LastName") %>'\>
    </asp:TextBox><br />
    BirthDate:
    <asp:TextBox ID="BirthDateTextBox" runat="server"
        Text='<%# Bind("BirthDate") %>'\>
    </asp:TextBox><br />
    City:
    <asp:TextBox ID="CityTextBox" runat="server"
        Text='<%# Bind("City") %>'\>
    </asp:TextBox><br />
    Country:
    <asp:TextBox ID="CountryTextBox" runat="server"
        Text='<%# Bind("Country") %>'\>
    </asp:TextBox><br />

```

```

        <asp:LinkButton ID="InsertButton" runat="server"
            CausesValidation="True" CommandName="Insert" Text="Insert">
        </asp:LinkButton>
        <asp:LinkButton ID="InsertCancelButton" runat="server"
            CausesValidation="False" CommandName="Cancel" Text="Cancel">
        </asp:LinkButton>
    </InsertItemTemplate>
</ItemTemplate>
    FirstName:
    <asp:Label ID="FirstNameLabel" runat="server"
        Text='<%# Bind("FirstName") %>'></asp:Label><br />
    LastName:
    <asp:Label ID="LastNameLabel" runat="server"
        Text='<%# Bind("LastName") %>'></asp:Label><br />
    BirthDate:
    <asp:Label ID="BirthDateLabel" runat="server"
        Text='<%# Bind("BirthDate") %>'></asp:Label><br />
    City:
    <asp:Label ID="CityLabel" runat="server"
        Text='<%# Bind("City") %>'></asp:Label><br />
    Country:
    <asp:Label ID="CountryLabel" runat="server"
        Text='<%# Bind("Country") %>'></asp:Label><br />
    <asp:LinkButton ID="EditButton" runat="server"
        CausesValidation="False" CommandName="Edit" Text="Edit">
    </asp:LinkButton>
    <asp:LinkButton ID="DeleteButton" runat="server"
        CausesValidation="False" CommandName="Delete" Text="Delete">
    </asp:LinkButton>
    <asp:LinkButton ID="NewButton" runat="server"
        CausesValidation="False" CommandName="New" Text="New">
    </asp:LinkButton>
</ItemTemplate>
</asp:FormView>

```

The free-form nature of the FormView allows for layouts that are very different from those possible with the DetailsView, yet still gives you the same features to add, edit, and delete data.

GridView

Now for the granddaddy of all databound controls: the GridView control. Whether it lists transactions from your checking account or lists contacts in your address book, this control is used nearly everywhere. And while it lists data in a tabular format and allows you to add, edit, and delete records, it also allows you to sort the data. This makes it a powerful and easy means of looking at data in a way that is most useful to you. Listing 4-3 shows an example of the GridView control.

Listing 4-3. GridView Example

```

<asp:GridView ID="GridView1" runat="server" AllowPaging="True" AllowSorting="True"
    AutoGenerateColumns="False" DataKeyNames="PersonId"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="FirstName"
            HeaderText="First Name" SortExpression="FirstName" />
        <asp:BoundField DataField="LastName"
            HeaderText="Last Name" SortExpression="LastName" />
        <asp:BoundField DataField="City"
            HeaderText="City" SortExpression="City" />
        <asp:BoundField DataField="BirthDate"
            HeaderText="Birth Date" HtmlEncode="False"
            SortExpression="BirthDate" />
        <asp:BoundField DataField="Country"
            HeaderText="Country" SortExpression="Country" />
        <asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
    </Columns>
</asp:GridView>

```

The editing functionality of the BoundField used by the GridView and the DetailsView is pretty limited. Fields will be edited with a TextBox control for strings as well as dates. And in the case of the City and Country fields, it will allow for any value. It would be an improvement to use an editor that recognized data types. This shortcoming will be addressed in the following section.

Editing and Validating Fields

While the preceding controls do allow you to modify the data they display, they do not offer a great deal of smart functionality for various data types. For example, the date-editing function will have you edit the date with a regular TextBox control. Entering a string that is not a date will cause the postback event to throw an exception, as shown in Figure 4-1.

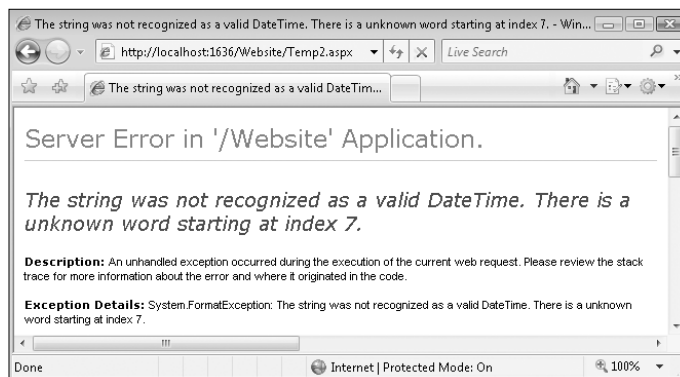


Figure 4-1. *FormatException when saving a date value*

You may even enter a value that looks like a valid date, such as June 31, but there is no 31st day of June, so it will also fail. It would be best to prevent that error from happening and prevent the exception. To do so, the BoundField can be replaced with a TemplateField. This can be done by converting the BoundField to a TemplateField in the Fields panel, shown in Figure 4-2, which is accessed by clicking the Edit Columns link on the Smart Tag for the GridView.

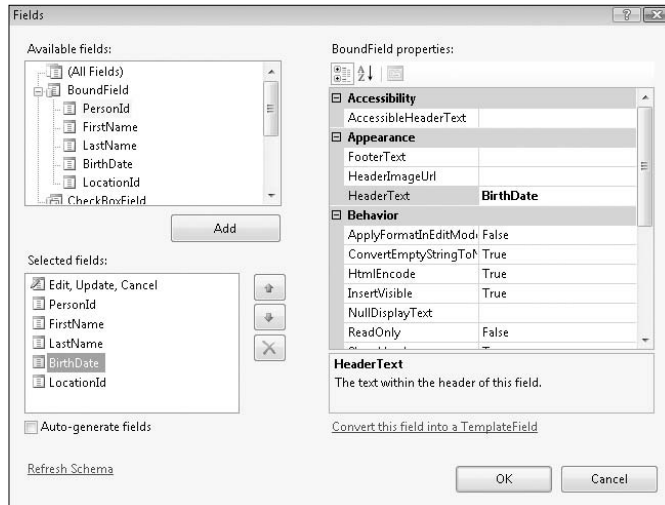


Figure 4-2. Fields editor

Once you have your TemplateField, it will start with a Label for read-only mode and a TextBox for edit mode and be placed in the ItemTemplate and EditTemplate. An example is shown in Listing 4-4.

Listing 4-4. *GridView with TemplateField*

```
<asp:GridView ID="GridView1" runat="server"
    AllowPaging="True" AllowSorting="True"
    AutoGenerateColumns="False" DataKeyNames="PersonId"
    DataSourceID="ObjectDataSource1">
    <Columns>
        <asp:BoundField DataField="FirstName"
            HeaderText="First Name" SortExpression="FirstName" />
        <asp:BoundField DataField="LastName"
            HeaderText="Last Name" SortExpression="LastName" />
        <asp:BoundField DataField="City" HeaderText="City" SortExpression="City" />
        <asp:TemplateField HeaderText="Birth Date" SortExpression="BirthDate">
            <EditItemTemplate>
                <asp:TextBox ID="TextBox1" runat="server"
                    Text='<%# Bind("BirthDate") %>'></asp:TextBox>
            </EditItemTemplate>
            <ItemTemplate>
                <asp:Label ID="Label1" runat="server"
```

```

        Text='<%# Bind("BirthDate") %>'></asp:Label>
    </ItemTemplate>
</asp:TemplateField>
<asp:BoundField DataField="Country"
    HeaderText="Country" SortExpression="Country" />
<asp:CommandField ShowDeleteButton="True" ShowEditButton="True" />
</Columns>
</asp:GridView>

```

The new `TemplateField` initially does not stop the problem with the invalid date but will allow you to add validation controls to check on the value. A `ValidationSummary` control can be placed above the `GridView` to show the `ErrorMessage` property of any validator that indicates an invalid value, but this would have to be repeated in every place that a date is edited. It also makes the `GridView` definition quite bulky. Instead, you will create a simple user control called `DateEditor`. This user control will be made up of a `TextBox` along with a `CustomValidator` and `RegularExpressionValidator`. The `DateEditor` is shown in Listing 4-5.

Listing 4-5. *DateEditor*

```

<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="DateEditor.ascx.cs" Inherits="DateEditor" %>
<asp:TextBox ID="TextBox1" runat="server"></asp:TextBox>
<asp:CustomValidator ID="cvDate" runat="server"
    ControlToValidate="TextBox1" Display="Dynamic"
    EnableClientScript="False" ErrorMessage="Date is invalid"
    OnServerValidate="cvDate_ServerValidate">*
</asp:CustomValidator>
<asp:RegularExpressionValidator ID="revDate" runat="server"
    ControlToValidate="TextBox1"
    Display="Dynamic" ErrorMessage="Date format is invalid. [MM/dd/yyyy]"
    ValidationExpression="^\d\d*\.\d\d*\.\d\d\d\d$" >*
</asp:RegularExpressionValidator>

```

In the code-behind, you add a little help to the `CustomValidator` to verify the value is truly a valid date and also make the value available through a property for the `TemplateField` that will hold the `DateEditor` (see Listing 4-6).

Listing 4-6. *DateEditor Code-Behind*

```

using System;
using System.Web.UI;
using System.Web.UI.WebControls;

public partial class DateEditor : UserControl
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}

```

```

public DateTime Date
{
    get
    {
        DateTime tmpDate = DateTime.MinValue;
        DateTime.TryParse(TextBox1.Text, out tmpDate);
        return tmpDate;
    }
    set
    {
        TextBox1.Text = value.ToString("MM/dd/yyyy");
    }
}

protected void cvDate_ServerValidate(object source,
ServerValidateEventArgs args)
{
    DateTime tmpDate;
    if (!DateTime.TryParse(TextBox1.Text, out tmpDate))
    {
        args.IsValid = false;
        return;
    }
    // these are the database constraints
    DateTime minDate = new DateTime(1753,1,1);
    DateTime maxDate = new DateTime(9999,12,31);
    if (tmpDate < minDate || tmpDate > maxDate)
    {
        args.IsValid = false;
        return;
    }
}
}

```

The code-behind carefully checks the value of the TextBox to ensure it never returns an invalid value. In the ServerValidate event handler for the CustomValidator, it uses the TryParse method of the DateTime object to check that the Text property from TextBox not only looks like a date, but also is a valid date. It will not allow June 31 through. And then in the Date property, the same check is done to prevent an invalid date from getting through. Back in the GridView, the DateEditor is placed in the EditTemplate with the Date property bound to the Birth Date field (see Listing 4-7).

Listing 4-7. *DateEditor*

```

<EditItemTemplate>
    <chpt04:DateEditor ID="DateEditor1" runat="server"
        Date='<%= Bind("BirthDate") %>' />
</EditItemTemplate>

```

Now when an invalid date is entered, a helpful error is shown instead of the exception. Better yet, the DateEditor can be used anywhere throughout the website, so a consistent validation check is used everywhere that will make for a better interface and less work.

Binding Input Parameters

Datasources can be configured with input parameters to filter data in multiple ways. Input parameters could come from controls like a TextBox or DropDownList as well as query string values. The wizard for configuring an ObjectDataSource and a SqlDataSource will detect when the datasource requires input parameters and will present these options. You can bind these parameters or choose to handle them programmatically.

Binding Input Parameters with a Control

The simplest example of working with input parameters is to bind them to a control such as a TextBox or a DropDownList. With the sample database, you will start by binding a list of last names to a DropDownList. Then a GridView will take the SelectedValue from the DropDownList and get all people where the last names are a match. This example can be done without writing any code in the code-behind file. It is just a matter of dragging and dropping the controls. Listing 4-8 shows the code.

Listing 4-8. *InputParameterExample.aspx*

```
<b>Select Last Name: </b>
<asp:DropDownList ID="DropDownList1" runat="server" AutoPostBack="True"
    DataSourceID="ObjectDataSource1" DataTextField="LastName">
</asp:DropDownList>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetLastNames" TypeName="Chapter04.PersonDomain">
</asp:ObjectDataSource>
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
    AllowSorting="True" DataSourceID="ObjectDataSource2">
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetPeopleByLastName"
    TypeName="Chapter04.PersonDomain">
    <SelectParameters>
        <asp:ControlParameter ControlID="DropDownList1"
            Name="lastName" PropertyName="SelectedValue"
            Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

The important part is near the bottom with the SelectParameters, which defines that the ControlParameter comes from the DropDownList1 and sets the lastName input parameter.

Binding Input Parameters Programmatically

Sometimes you cannot just get a value from a control. Either the datasource is not bound to the control yet or it needs to be carefully validated before setting the value as an input parameter to prevent an exception. In these cases, you can withhold the `ControlParameter` definition and instead leave the input parameter undefined except for the `Name` and `Type`. The `ObjectDataSource` has an event called `Selecting`, which is where you will want to set the input parameter programmatically. Listing 4-9 shows an example of declaring input parameters.

Listing 4-9. *InputParameterExample2.aspx*

```
<b>Select Last Name: </b>
<asp:DropDownList ID="DropDownList1" runat="server" AutoPostBack="True"
    DataSourceID="ObjectDataSource1" DataTextField="LastName">
</asp:DropDownList>
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
    AllowSorting="True" DataSourceID="ObjectDataSource2">
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetLastNames"
    TypeName="Chapter04.PersonDomain"></asp:ObjectDataSource>
<asp:ObjectDataSource ID="ObjectDataSource2" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetPeopleByLastName"
    TypeName="Chapter04.PersonDomain"
    OnSelecting="ObjectDataSource2_Selecting">
    <SelectParameters>
        <asp:Parameter Name="lastName" Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

Notice that the `OnSelecting` event handler is specified and the `SelectParameters` specifies the `lastName` parameter with no control association. The value for the parameter is set in the code-behind file. Listing 4-10 shows an event handler for the `Selecting` event.

Listing 4-10. *Selecting Event Handler*

```
protected void ObjectDataSource2_Selecting(object sender,
    ObjectDataSourceSelectingEventArgs e)
{
    e.InputParameters["lastName"] = DropDownList1.SelectedValue;
}
```

If this input parameter wanted a date and the value was coming from a `TextBox`, the value could be checked to ensure that it really is a valid date and only set the input parameter when it is valid. An input parameter may also be a more complex type, like a business object, that you assemble in the `Selecting` event handler. This simple handler gives you many options.

Binding a User Control

All too often I have seen very large pages with piles and piles of markup in the template side of each page and plenty more code in the code-behind because the pages were not broken up into user controls. A user control is just a fragment of a page that is very useful when used to encapsulate a section of the page. It will isolate not only a piece of the markup, but also events and behavior. User controls can also be used multiple times on one page as well as across many pages. It is a great way to introduce code reuse for a website.

The matter of binding to a user control can be tricky when you first attempt it. Should it try to read values from the query string or somehow traverse the control hierarchy to get to a control that defines the input parameter it needs to carry out its `DataBind` function? The best way to handle this is to treat the user control just like you already treat other controls such as the `TextBox` and set properties on the user control. First you need a user control, as shown in Listing 4-11.

Listing 4-11. *PersonListingControl.ascx*

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="PersonListingControl.ascx.cs"
    Inherits="Controls_PersonListingControl" %>
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
    AllowSorting="True" DataSourceID="ObjectDataSource1">
</asp:GridView>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetPeopleByLastName"
    TypeName="Chapter04.PersonDomain"
    OnSelecting="ObjectDataSource1_Selecting">
    <SelectParameters>
        <asp:Parameter Name="lastName" Type="String" />
    </SelectParameters>
</asp:ObjectDataSource>
```

This example includes the `GridView` as in the previous examples, yet it is held in a user control. It also has an `ObjectDataSource` with a `lastName` input parameter with the `Selecting` event handler specified. The parent page will need to pass in the value for the `lastName` parameter, and you will do it with a property declaration (see Listing 4-12).

Listing 4-12. *PersonListing.aspx.cs Code-Behind*

```
protected void ObjectDataSource1_Selecting(
    object sender, ObjectDataSourceSelectingEventArgs e)
{
    e.InputParameters["lastName"] = LastName;
}

private string _lastName = String.Empty;
public string LastName
{
```

```

    get
    {
        return _lastName;
    }
    set
    {
        _lastName = value;
        ObjectDataSource1.DataBind();
    }
}

```

The `LastName` property holds the value that is used on the `Selecting` event handler. And when the `LastName` property is set, you can see that it calls `DataBind` on `ObjectDataSource1` to immediately bind the data to the `GridView`. Without calling `DataBind`, the `GridView` may not be bound at all because it will not automatically get fired like it would if the `ObjectDataSource` was configured with a control property. It makes sense to call it right here when the value is set, but it can also be called on the `PreRender` event for the user control or even have the page holding this control call the `DataBind` method on the user control. Having the data bound immediately once the value is known ensures that the data is always bound.

In the case when there are multiple properties that are used to set multiple input parameters, it is better to defer the data binding to a later event like the `PreRender` event so all of the properties can be set before the `Selecting` event runs.

Embedding Databound Controls

For a more complex data hierarchy, you may find that you would like to embed a user control within another user control, just as the previous example embedded a user control within a page and bound data to it by passing a value to it through a property. Perhaps you are showing a report, and with each section there is a related subsection. Creating a user control to represent the section that holds another user control representing the subsection is a great way to break up the code into isolated pieces that are responsible for their own data binding.

The sample database has a `Location` table holding a `City` and a `Country` column. You will bind a list of countries to a `Repeater` control that displays the country with a `Label` control and then passes the country to a user control to show a list of cities in that country. You just need to design the user controls and wire them together to set the properties and fire the data binding in the right order.

The first user control, shown in Listing 4-13, draws a bulleted list with the countries.

Listing 4-13. *CountryListingControl.ascx*

```

<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="CountryListingControl.ascx.cs"
    Inherits="Controls_CountryListingControl" %>
<%@ Register Src="CityListingControl.ascx"
    TagName="CityListingControl" TagPrefix="uc1" %>
<asp:Repeater ID="Repeater1" runat="server"
    DataSourceID="ObjectDataSource1">

```

```

<HeaderTemplate>
    <ul>
</HeaderTemplate>
<ItemTemplate>
    <li>
        <asp:Label ID="Label1" runat="server"
            Text='<%# Bind("Country") %>'></asp:Label>
        <uc1:CityListingControl ID="CityListingControl1"
runat="server" Country='<%# Bind("Country") %>' />
    </li>
</ItemTemplate>
<FooterTemplate>
    </ul>
</FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetAllCountries"
    TypeName="Chapter04.PersonDomain"></asp:ObjectDataSource>

```

What is interesting here is that the Country property on the user control can be bound to the Country value just like the Label control, which is a standard ASP.NET control. When the data is bound to Repeater1, it automatically sets the Country property on CityListingControl1. And this all happens without any additional code in the code-behind. Listing 4-14 shows the city listing control.

Listing 4-14. *CityListingControl.ascx*

```

<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="CityListingControl.ascx.cs" Inherits="Controls_CityListingControl" %>
<asp:Repeater ID="Repeater1" runat="server" DataSourceID="ObjectDataSource1">
    <HeaderTemplate>
        <ul>
    </HeaderTemplate>
    <ItemTemplate>
        <li>
            <asp:Label ID="Label1" runat="server"
                Text='<%# Bind("City") %>'></asp:Label><br />
        </li>
    </ItemTemplate>
    <FooterTemplate>
        </ul>
    </FooterTemplate>
</asp:Repeater>
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetCitiesByCountry"
    TypeName="Chapter04.PersonDomain"

```

```
OnSelecting="ObjectDataSource1_Selecting">
<SelectParameters>
    <asp:Parameter Name="country" Type="String" />
</SelectParameters>
</asp:ObjectDataSource>
```

This user control also makes use of a Repeater to list all of the data in a bulleted list. In this case, it is all of the cities in the selected country. In the code-behind, shown in Listing 4-15, a little bit of code is put in place to wire together the parent user control to bind the data.

Listing 4-15. *CityListingControl.ascx.cs*

```
protected void ObjectDataSource1_Selecting(
    object sender, System.Web.UI.WebControls.ObjectDataSourceSelectingEventArgs e)
{
    e.InputParameters["country"] = Country;
}

private string _country;
public string Country
{
    get {
        return _country;
    }
    set {
        _country = value;
        ObjectDataSource1.DataBind();
    }
}
```

You could theoretically go many levels deeper, perhaps listing the people who live in each city by following this same pattern of passing properties into the user control to be bound as an input parameter. And doing so completely encapsulates the behavior of each user control.

BEWARE OF RECURSIVE DATA BINDING

Be sure to bind just the `ObjectDataSource` and not the entire user control when the properties are set. Rebinding the user control will cause an infinite recursive loop on the property as the user control repeatedly binds the data over and over.

ViewState and Databinding

When working with databinding, you will be using ViewState extensively as a part of the postback model. All of the data that is bound to a control, such as a GridView, is held in ViewState. By default, the data is serialized to a string, encrypted, and included in the page as a hidden input field. When a postback occurs, the value from the hidden field is decrypted and deserialized. The postback process then attempts to reset the control values with this data. By using the ViewState to redraw a databound control, you avoid going back to the database to get all of the necessary data. On a typical page, you may have several databound controls that will make use of this ViewState. In the case of a GridView, you could click one of the pager links to change the selected index, which will cause new data to be pulled from the database. While that happens, the other databound controls will simply use the ViewState data to redraw themselves. It is a pretty efficient model if you compare it to other solutions that would need to rebind all of the data from the database with each page request.

However, using ViewState does mean that when a page displays a large set of data, the ViewState data held in the hidden input field is also large. It is also a duplication of the displayed data held inside of an encrypted string to preserve the integrity of the data. So while you reduce the number of trips to the database during postbacks, you cause the page load time to increase, which may be just as much of a performance penalty for users who have a limited-bandwidth connection to your web application. Such users could be using a modem or mobile connection that does not offer high-speed access. They could also be sharing a high-speed connection with everyone in their office, which degrades the speed of their access. In any case, you want to make your page load times as fast as possible by reducing the size of the page. When ViewState becomes large, you will want to consider ways to reduce the size.

PAGE SIZE AND LOAD TIME

Jakob Nielsen, a leader in web usability, notes on his website that a user connected to the Internet with a modem will take ten seconds to load a page that is just 34 KB (<http://www.useit.com/alertbox/sizelimits.html>). For a Cable/DSL connection, it would take one second to load a 100KB page. It is easy for a data-heavy page to go well beyond 100 KB, so high-speed users may still wait two to five seconds for your pages to load. Cutting the ViewState could save a few seconds for those high-speed users and much more for any user on a slower connection. Of course these delays are extended by the time to prepare and start sending a response to a user.

Session and ViewState

You can change how ViewState is persisted to avoid placing all of this extra data in the page. ASP.NET 2.0 introduced the option to store ViewState in the Session instead of the hidden input field. You just have to override the PageStatePersister property to return either the HiddenFieldPagePersister, which is the default, or the SessionPageStatePersister. An example is shown in Listing 4-16.

Listing 4-16. *PageStatePersister Set to Use the Session*

```
protected override PageStatePersister PageStatePersister
{
    get
    {
        return new SessionPageStatePersister(Page);
    }
}
```

The implementation for `SessionPageStatePersister` places a `Guid` value into the hidden input field in place of all the encrypted data and places the actual data into the user's `Session`. When a postback occurs, the `Guid` value is used to look up that data. This data is limited to a rotating queue that holds only the last ten pages of data. And while this reduces the page size, the limitation of placing it in the `Session` presents a few problems.

The clear problem is that if the data is removed from the `Session`, a postback will throw an exception. This can happen if the user's `Session` expires after the default time-out of 20 minutes. When a user's `Session` expires, the `Session` is abandoned on the server along with the server-side `ViewState`. If a user steps away for that time period, returns, and then clicks a button causing a postback, that user will see the exception. You may consider bumping up the `Session` time-out to eight hours, but doing so will cause more memory to be used on the server. If you retain the `Session` state for all users who visit your website in that eight-hour period, you may struggle with the amount of memory that is required.

Users could also open two browser windows while using your web application. They click around in one window for a while and then move to the other window and click some more, causing postbacks as they do so. If they go beyond the ten-click limit and return to the other window and cause a postback, they will get the state exception. Fortunately, this feature can be enabled on a page-by-page basis. If your website has a limited set of data-intensive pages, you could set them to use the `SessionPageStatePersister` while the rest of the pages use the default functionality.

You could also implement your own custom `PageStatePersister` that works much like the `SessionPageStatePersister`, but instead of limiting the data to ten items, older data could be stored to the database in case it is needed. A scheduled job could purge this data nightly.

Paging

The size of the `ViewState` can also be reduced using paging. Consider a `GridView` that has 1,000 total items with the page size set to 10 items. Only the data for those 10 items will be persisted in `ViewState` instead of the data for all 1,000 items. When paging is combined with the subset selection technique covered in the last chapter, you can ensure that only the 10 rows that are shown are pulled from the database, which will reduce the load on the database. You will learn more about the benefits of paging later when you get into building a custom databound control in the section "Creating a Databound Control."

Disabling ViewState

Another way to reduce page size is to disable ViewState. You can do so with an entire page or at the control level with the `EnableViewState` property. If you disable ViewState for the page, that decision will have an impact on every control included on the page. Each control will not be able to override that setting, so the impact would be universal and possibly damaging for controls that require ViewState. The same is true for any control that holds other controls below it. The setting impacts all controls below it.

Due to the fact that you cannot enable ViewState for a control if the ViewState has been disabled for the entire page, you will want to identify individual controls that do not need it. A navigational control included on each page, perhaps as a part of the master page, would make for an ideal candidate. If it simply contains markup and a few `HyperLink` controls that have the `NavigateUrl` set declaratively, the value will be available during a postback when ViewState is disabled.

For a custom user control, you may also design its behavior to work with and without ViewState. Listing 4-17 shows how this can be done.

Listing 4-17. Working With and Without ViewState

```
if (! IsPostBack)
{
    GridView1.DataSource = GetDataSource();
    GridView1.DataBind();
}
else if (! IsViewStateEnabled)
{
    GridView1.DataSource = GetDataSource();
    GridView1.DataBind();
}
```

The datasource for the GridView is set on the first page load when it is not a postback and when it is a postback with ViewState disabled. Doing it this way will load the GridView data when the control needs it. However, this approach is pretty manual. It is better to simply use a declared `ObjectDataSource` reference. Doing so will allow the GridView to automatically detect when ViewState is disabled and bind using the `ObjectDataSource` when it needs data.

ControlState vs. ViewState

Because some pieces of data are critical to the functionality of a control, it is necessary to retain access to those details when ViewState is enabled. Controls such as the GridView are smart enough to work with and without ViewState, but they still need to retain the current state of the control related to position when paging is enabled. The data for each of the columns can be pulled from the database with each postback, but the database will not know that the fourth page of the GridView was selected. This value is retained by the `SelectedItem` property that is persisted with `ControlState`, a variation of ViewState.

`ControlState` was introduced with ASP.NET 2.0. It is not an automatic feature like ViewState. You must implement it yourself and register with the Page to persist the data. Listing 4-18 shows how to save `ControlState`.

Listing 4-18. *SaveControlState*

```
protected override object SaveControlState()
{
    Pair state = new Pair();
    state.First = base.SaveControlState();
    state.Second = _controlStateData;

    return state;
}
```

ControlState and ViewState are often persisted using the Pair and Triplet types from the System.Web.UI namespace. They are simply tiny arrays that hold onto your data. You could also use an array of objects that can be serialized, which means it is best to stick to types like string, int, and DateTime. You want to use the least amount of data to persist the necessary state. Using the Pair type is a good first layer that will hold the ControlState of the base class in the First property and the state of the current instance in the Second property. If you have many values to persist, the value stored in the Second property could be the array of many objects. This hierarchy of this state will then be unfolded when it is loaded, as shown in Listing 4-19.

Listing 4-19. *LoadControlState*

```
protected override void LoadControlState(object savedState)
{
    Pair pair = savedState as Pair;
    if (pair != null)
    {
        base.LoadControlState(pair.First);
        _controlStateData = (string)pair.Second;
    }
    else
    {
        base.LoadControlState(null);
    }
}
```

The state is cast as a Pair type and used to load the ControlState of the base class with the First property and then the state of the current instance with the Second property. With these values defined, it is necessary to tell the Page that ControlState is required for this control. This must be done during the Init event before the events to load ControlState are fired later in the event life cycle, as shown in Listing 4-20.

Listing 4-20. *Requiring ControlState*

```
protected void Page_Init(object sender, EventArgs e)
{
    Page.RegisterRequiresControlState(this);
}
```


The preceding examples for `ControlState` simply persist a single string variable. Many times you should not have much data to persist. The event ordering will cause `ControlState` to be persisted and loaded between the `Init` and `Load` events so that the persisted data will be available when your `Load` event is fired. Later the `PreRender` event will fire where most databound controls actually load the data. It is important to know when this data is available to your page or control.

Given the option to persist state in the `Session` or in `ControlState`, it should be noted that not all state has to be persisted. Some data can be lost safely because the input fields that were using them are already persisting the data, such as a `TextBox`. The value from the form will always be set in the `Text` property. However, the `TextChanged` event will now fire with each postback because the control cannot compare whether or not the value actually changed. If your control does handle this event, you will not be affected by `ViewState` being disabled. Other controls such as the `ListBox` and `RadioButtonList` may be populated declaratively instead of binding them to a `datasource`. These will work like the `TextBox` in that they can give you the currently selected value properly and redraw themselves with all of the available selections with the declared values. And again, the events triggered when a selection changes will not work as they would with `ViewState` enabled. In either situation, you can still get the value the user has set and take action as needed.

Creating a Databound Control

In order to deepen your understanding of how databound controls work, you will learn how to create a control that will be bound to an `ObjectDataSource`. You will be able to do more with this control than you can with standard controls such as the `GridView` and `DetailsView` because you will have the full source code, which you can modify and walk with the debugger. You can also see directly what the consequences are for those modifications.

This new control, called the `PersonListingControl`, will take in several fields related to people and locations and display them in one of two formats. It will also offer optional paging support that is coordinated with the `datasource`, which is defined declaratively with the `DataSourceID` property on the control. It will also work with and without `ViewState` enabled. This control will implement a great deal of functionality with very rich controls such as the `GridView`.

WHY CREATE A CUSTOM DATABOUND CONTROL?

The purpose of this databound control is purely as a learning tool. You would be hard pressed to come up with many scenarios where you cannot use the existing databound controls, from the `GridView` to the `Repeater`, to serve your needs. What you cannot do with those controls is walk them with the debugger to see exactly what is happening internally and experiment with various tests to see how you can improve on the process. Because this databound control also inherits from `CompositeDataBoundControl`, it will behave like the standard databound controls.

However, you could use this example as a basis for creating a practical solution that is closely customized to your application to leverage every advantage available to you. You may not see a great deal of improved performance just by converting a user control to a server control. With this example to get you started, you may be able to compare the two and measure the difference.

When you dig into the code for this control, you will start to see all of the facilities that are available to a control through the use of abstract classes. Just a few years ago, creating controls was quite difficult. In ASP.NET 2.0, it became much easier with the `CompositeControl` and `CompositeDataBoundControl`, which are base classes that the standard ASP.NET controls are built on. You can also use these base classes, which automatically provide you with rich Design Time support as well as extensive runtime functionality.

Both of these base classes include a method called `CreateChildControls`, which takes no parameters and does not return anything. The control hierarchy is created in this method, which is called automatically as a part of the event life cycle. In the `CompositeControl` you could override the behavior of this method, but in the `CompositeDataBoundControl` this method has already been implemented. Instead, there is an abstract method by the same name that takes a couple of parameters and has a result type. Because it is an abstract method, it must be implemented in the inheriting class. The example method signature is shown in Listing 4-21.

Listing 4-21. *CreateChildControls for CompositeDataBoundControl*

```
protected override int CreateChildControls(IEnumerable dataSource, bool dataBinding)
{
    // code
}
```

The `CreateChildControls` method is the core of the databound control. The overall control is created by initializing controls and adding them to the `Controls` collection. The `Controls` property is the accessor for this collection. When the `dataBinding` parameter is true, the `dataSource` value should be holding onto data that can be enumerated over while creating a collection of controls that are added to the `Controls` collection.

This may make you wonder why the `dataBinding` parameter would ever be false. You may also wonder why it returns an integer. When a postback occurs, the `ViewState` must be applied to the control hierarchy constructed when the control was first created. This means there must be the same number of rows, but because you are not going to have access to the actual data at this time, you have to create the skeleton of the previous hierarchy that will match the data in `ViewState`. There must be the same number of items in the `Controls` collection as there were before in order for the `ViewState` to be applied successfully.

For the `PersonListingControl`, these items are instances of the `PersonRow` control. This is equivalent to a `GridViewRow`. Each item pulled from the `IEnumerable` value is passed to the `PersonRow` through the constructor and added to the `Controls` collection. The `PersonRow` is then responsible for drawing that individual row.

There are a few additional classes that provide Design Time support that you will find in the downloadable sample code. The class library with these classes looks like Figure 4-3.

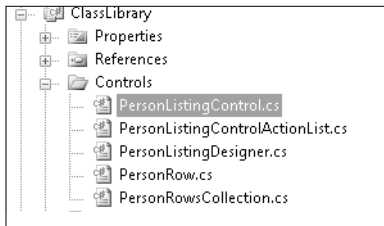


Figure 4-3. Files for the *PersonListingControl*

You will skip the *PersonRow* for the moment and look at the implementation of the *CreateChildControls* method. When the control is created for the initial page request, it will call *CreateChildControls* with the Boolean value set to true with a defined value for the data. You can iterate over that data, create the *PersonRow* instances, and add them to the *Controls* collection.

During a postback, before *ViewState* is applied, the *CreateChildControls* method must be called with the Boolean value set to false for the *dataBinding* parameter and a null value for the *dataSource* parameter. And somehow the number of items created in the previous control rendering must be known at this point so the correct number of items are re-created. Listing 4-22 shows the full implementation.

Listing 4-22. *CreateChildControls Implementation*

```
protected override int CreateChildControls(
    IEnumerable dataSource, bool dataBinding)
{
    Controls.Clear();
    int count = 0;

    if (dataBinding && dataSource != null)
    {
        IEnumerator e = dataSource.GetEnumerator();

        while (e.MoveNext())
        {
            object datarow = e.Current;
            PersonRow row = new PersonRow(count, datarow);
            Rows.Add(row);
            Controls.Add(row);
            count++;
        }
        _itemCount = count;
    }
    else
    {
        if (_itemCount > 0)
        {
```

```

        for (count = 0; count < _itemCount; count++)
        {
            PersonRow row = new PersonRow(count, null);
            Rows.Add(row);
            Controls.Add(row);
        }
    }

    CreatePagerControls();
    AttachStyle();

    ClearChildViewState();
    ChildControlsCreated = true;

    return count;
}

```

When there is data available, the first section of code moves over the enumerator, creates instances of `PersonRow`, and adds them to the `Rows` and `Controls` collections and increments the count. When there is no data, the member variable `_itemCount` is used to ensure the correct number of empty items is added to the `Controls` collection. As you can see, a null value is given to the `PersonRow` constructor. Finally, at the very end the count variable that was incremented on both execution branches is used as the return value.

Getting the Data

In order to get the data that is passed into `CreateChildControls`, a databound control will fire the `PerformSelect` method. The implementation details of the `CompositeDataBoundControl` take care of calling this method when it is needed so you do not have to call it explicitly with your own code. This is the method that raises the `OnDataBinding` and `OnDataBound` events before and after the data is requested.

The data is represented as a `DataSourceView`, which could be a `datasource` that was manually bound to the control or defined by the `DataSourceID` property. In any case, you can get an instance of it with the built-in `GetData` method, as shown in Listing 4-23.

Listing 4-23. *GetData Method*

```
DataSourceView dataSourceView = GetData();
```

This view has a method called `Select` that takes two parameters, `DataSourceSelectArguments` and `DataSourceViewSelectCallback`, which are used to give the view additional details that will be used when retrieving the data. In the case of an `ObjectDataSource` configured with a `DataObject` and `DataObjectMethod` with paging enabled, the `StartRowIndex` and `MaximumRows` must be defined. These values are passed as parameters to the `DataObjectMethod`. This is shown in Listing 4-24.

Listing 4-24. Calling the Select Method

```
DataSourceSelectArguments selectArguments = CreateDataSourceSelectArguments();
selectArguments.StartRowIndex = PageSize * PageIndex;
selectArguments.MaximumRows = PageSize;
dataSourceView.Select(selectArguments, callback);
```

The callback takes the data and carries out the data binding. Declare this callback inline with the PerformSelect method as a delegate. The callback delegate is shown in Listing 4-25.

Listing 4-25. DataSourceViewSelectCallback

```
DataSourceViewSelectCallback callback =
    delegate(IEnumerable data)
    {
        if (IsBoundUsingDataSourceID)
        {
            OnDataBinding(EventArgs.Empty);
        }
        PerformDataBinding(data);
    };
```

Both the PersonListingControl and the ObjectDataSource have a property named EnablePaging that controls how the Select method will work. If paging is enabled, it requires the StartRowIndex and MaximumRows values and passes them into the declared DataObjectMethod. If paging is not enabled, it uses a method that does not have those parameters. However, the ObjectDataSource and PersonListingControl do not have to have the same value for EnablePaging. If the ObjectDataSource has paging enabled and the PersonListingControl does not, it will still use the method that takes the paging parameters. When this mixed mode is the case, the values should be set to allow for all data to be returned to the control. The code to these values is shown in Listing 4-26.

Listing 4-26. Select Arguments in Mixed Mode

```
if (dataSourceView.CanPage)
{
    selectArguments.AddSupportedCapabilities(DataSourceCapabilities.None);
    selectArguments.StartRowIndex = 0;
    selectArguments.MaximumRows = Int16.MaxValue;
}
```

The MaximumRows property is set to Int16.MaxValue to allow for effectively unlimited results. This mixed mode is useful when more than one control is using the same ObjectDataSource.

Once the Select method has been called on the view, that data will be sent to the callback.

To finish up, the RequiresDataBinding value must be set to false to indicate that the data has been requested. Also, the MarkAsDataBound method, which updates the state of the control to indicate that data has been successfully bound to the control, must be called. The PerformSelect method is shown in Listing 4-27.

Listing 4-27. *PerformSelect Method*

```

protected override void PerformSelect()
{
    if (!IsBoundUsingDataSourceID)
    {
        OnDataBinding(EventArgs.Empty);
    }

    DataSourceView dataSourceView = GetData();

    DataSourceViewSelectCallback callback =
        delegate(IEnumerable data)
        {
            if (IsBoundUsingDataSourceID)
            {
                OnDataBinding(EventArgs.Empty);
            }
            PerformDataBinding(data);
        };

    if (EnablePaging && dataSourceView.CanPage)
    {
        DataSourceSelectArguments selectArguments =
            CreateDataSourceSelectArguments();
        selectArguments.StartRowIndex = PageSize * PageIndex;
        selectArguments.MaximumRows = PageSize;
        dataSourceView.Select(selectArguments, callback);
    }
    else
    {
        DataSourceSelectArguments selectArguments =
            CreateDataSourceSelectArguments();
        if (dataSourceView.CanPage)
        {
            selectArguments.AddSupportedCapabilities(DataSourceCapabilities.None);
            selectArguments.StartRowIndex = 0;
            selectArguments.MaximumRows = Int16.MaxValue;
        }
        dataSourceView.Select(selectArguments, callback);
    }

    RequiresDataBinding = false;
    MarkAsDataBound();

    OnDataBound(EventArgs.Empty);
}

```

Getting the Total Rows Count

When paging through data, it is necessary to know the total number of rows. Because the data returned from the selection is limited to the `MaximumRows` property, it does not represent the total. Instead, you must use the `Select` method on the `DataSourceView` to get the `TotalRowCount`, as is done in Listing 4-28.

Listing 4-28. *GetTotalRowCount*

```
private int GetTotalRowCount()
{
    int totalRowCount = 0;
    DataSourceView dataSourceView = GetData();
    if (dataSourceView.CanRetrieveTotalRowCount)
    {
        DataSourceSelectArguments selectArguments =
            CreateDataSourceSelectArguments();
        selectArguments.AddSupportedCapabilities(
            DataSourceCapabilities.RetrieveTotalRowCount);
        selectArguments.RetrieveTotalRowCount = true;
        DataSourceViewSelectCallback callback =
            delegate
            {
                totalRowCount = selectArguments.TotalRowCount;
            };
        dataSourceView.Select(selectArguments, callback);
    }
    return totalRowCount;
}
```

The `DataSourceSelectArguments` method is adjusted by adding the `RetrieveTotalRowCount` capability, which calls the `TotalRowCount` to be populated for the callback. This value is captured with the delegate and returned for use in other parts of the control. Specifically, this value is used by the `TotalRowCount` property shown in Listing 4-29.

Listing 4-29. *TotalRowCount Property*

```
private int _totalRowCount = 0;

[Browsable(false)]
public virtual int TotalRowCount
{
    get
    {
        if (_totalRowCount == 0)
        {
            _totalRowCount = GetTotalRowCount();
        }
        return _totalRowCount;
    }
}
```

The `TotalRowCount` property is used along with the `PageSize` property to calculate the value of the `MaxIndex` property, which is shown Listing 4-30.

Listing 4-30. *MaxIndex Property*

```
private int _maxIndex = 0;

[Browsable(false)]
public virtual int MaxIndex
{
    get
    {
        if (_maxIndex == 0)
        {
            double maxIndexDb1 = (TotalRowCount/PageSize) - 1;
            _maxIndex = (int)maxIndexDb1;
            if (maxIndexDb1 > _maxIndex)
            {
                _maxIndex++;
            }
        }
        return _maxIndex;
    }
}
```

To control paging, the `PersonListingControl` uses pager controls to move to the next and previous page using postback events. These pager controls are links created using `LinkButton` controls that have the `CommandArgument` value set to the targeted `PageIndex`.

Wiring the Pager Events

The pager controls fire postback events in order to set the new `PageIndex` value. Typically, attaching an event is fairly automatic. Because this is a custom databound control, it is necessary to initialize the control, attach the event, and then add it to the `Controls` collection so that when the postback events are fired, it is included. The sequence of events happens in this order: `Init`, `PagePreLoad`, `Load`, and `RaisePostBackEvent`. Between the `Init` and `PagePreLoad` events, the `ControlState` and `ViewState` are loaded. The pager controls must be initialized and added to the `Controls` collection prior to loading states and the `RaisePostBackEvent`. To ensure this is done, these controls can be initialized in `CreateChildControls` with the call to the `CreatePagerControls` method.

A control only needs to be initialized once, while each time `CreateChildControls` is run the controls need to be added back to the `Controls` collection. The `CreatePagerControls` method is shown in Listing 4-31.

Listing 4-31. *CreatePagerControls Method*

```
private void CreatePagerControls()
{
```



```

InitializePagerControls();

Controls.Add(_previousLinkButton);
Controls.Add(new LiteralControl(" "));
Controls.Add(_nextLinkButton);
Controls.Add(new LiteralControl(" "));
}

```

The work to set the properties on the controls and attach the events is handled in the `InitializePagerControls` method, which has a check in place to ensure that it is only run once. The code for this method is shown in Listing 4-32.

Listing 4-32. *InitializePagerControls*

```

private void InitializePagerControls()
{
    if (_controlsInitialized)
    {
        return;
    }
    if (_nextLinkButton == null)
    {
        _nextLinkButton = new LinkButton();
        _nextLinkButton.ID = "lbNext";
        _nextLinkButton.Text = "Next";
        _nextLinkButton.CssClass = "btn";
        _nextLinkButton.CommandName = SetPageIndexCommandName;
        _nextLinkButton.Click += new EventHandler(PagerButton_Click);
    }
    if (_previousLinkButton == null)
    {
        _previousLinkButton = new LinkButton();
        _previousLinkButton.ID = "lbPrev";
        _previousLinkButton.Text = "Previous";
        _previousLinkButton.CssClass = "btn";
        _previousLinkButton.CommandName = SetPageIndexCommandName;
        _previousLinkButton.Click += new EventHandler(PagerButton_Click);
    }
    _controlsInitialized = true;
}

```

These pager methods ensure that the events will fire during the postback process. And while moving from page to page, the next and previous links may or may not be available to the user. If the user is already on the last page of the controls, the next link should be hidden. Because the `RaisePostBackEvent` runs after the `Load` event, even the visibility of these pager links must be set in the `PreLoad` event shown in Listing 4-33.

Listing 4-33. *PreLoad Event*

```
protected override void OnPreRender(EventArgs e)
{
    base.OnPreRender(e);

    _previousLinkButton.Text = PreviousPageText;
    _nextLinkButton.Text = NextPageText;

    _previousLinkButton.CommandArgument = (PageIndex - 1).ToString();
    _previousLinkButton.Visible = EnablePaging && PageIndex > 0;

    _nextLinkButton.CommandArgument = (PageIndex + 1).ToString();
    _nextLinkButton.Visible = EnablePaging && PageIndex < MaxIndex;
}
```

When the page is first loaded, the previous link will not be visible. The next link will have the `CommandArgument` set to 1, which will be used to set the `PageIndex` when the pager event is handled by the `PagerButton_Click` event handler in Listing 4-34.

Listing 4-34. *PagerButton_Click Event Handler*

```
protected void PagerButton_Click(object sender, EventArgs e)
{
    LinkButton lb = sender as LinkButton;
    if (lb != null && SetPageIndexCommandName.Equals(lb.CommandName))
    {
        int pageIndex;
        int.TryParse(lb.CommandArgument, out pageIndex);
        PageIndex = pageIndex;
    }
}
```

Implicitly some data binding is happening. The `PageIndex` property is sensitive to change, as shown in Listing 4-35.

Listing 4-35. *PageIndex Property*

```
private int _pageIndex = 0;

[Browsable(false)]
public virtual int PageIndex
{
    get
    {
        EnsureChildControls();
        return _pageIndex;
    }
    set
```

```

    {
        EnsureChildControls();
        if (value < 0)
        {
            throw new ArgumentOutOfRangeException("value");
        }
        if (_pageIndex != value)
        {
            _pageIndex = value;
            if (Initialized)
            {
                RequiresDataBinding = true;
            }
        }
    }
}

```

The set operation of the `PageIndex` property will set the `RequiresDataBinding` value to true when the value for `PageIndex` changes, which will cause the data to be bound later in the event life cycle.

Creating PersonRow

The main content of the `PersonListingControl` is the collection of `PersonRow` items. This is a simple control that uses the data given to it to bind to the controls that it creates. The `PersonRow` inherits from `WebControl` and implements the `IDataItemContainer` and `INamingContainer` interfaces. Remember that the `INamingContainer` interface has no methods as it is just a marker interface used to manage the naming container hierarchy. The `IDataItemContainer` implements three read-only properties, shown in Listing 4-36.

Listing 4-36. *IDataItemContainer* Members

```
#region " IDataItemContainer Members "
```

```

public object DataItem
{
    get
    {
        return Data;
    }
}

public int DataItemIndex
{
    get
    {
        return _itemIndex;
    }
}

```

```

    }

    public int DisplayIndex
    {
        get
        {
            return _itemIndex;
        }
    }

    #endregion

```

When the `PersonRow` is constructed, these values are passed in as parameters. The data is stored in a member variable that is accessed with the `Data` property. The constructor and `Data` property are shown in Listing 4-37.

Listing 4-37. *PersonRow Constructor and Data Property*

```

public PersonRow(int itemIndex, object o)
    : base(HtmlTextWriterTag.Div)
{
    _data = o;
    _itemIndex = itemIndex;
}

public virtual object Data
{
    get
    {
        return _data;
    }
}

```

The data passed in here could be a null value, so when the `CreateChildControls` method is called, this should be handled properly. If there is data, the values can be loaded using the `DataBinder` from the data object with the code in Listing 4-38.

Listing 4-38. *Data Binding Code*

```

if (Data != null)
{
    DateTime dateValue;
    DateTime.TryParse(DataBinder.GetPropertyValue(Data, birthDateField, ""), ➡
    out dateValue);

    FirstName = DataBinder.GetPropertyValue(Data, firstNameField, null);
    LastName = DataBinder.GetPropertyValue(Data, lastNameField, null);
    BirthDate = dateValue;
    City = DataBinder.GetPropertyValue(Data, cityField, null);
}

```

```
Country = DataBinder.GetValue(Data, countryField, null);
}
```

You will notice that there is no code here to check if the data is a `DataSet`, an `IDataReader`, or a custom business object. The `DataBinder` uses reflection to get the value from the object and set it on the property. The `FirstName` property receives the value associated with the `firstNameField`, which is a string that should match a property on the data object. By default, the string value for `firstNameField` is `FirstName`, which matches up with one of the columns returned by the stored procedure that is used with this control. And like a good databound control, this string can be adjusted with a series of properties in case you cannot match up the field names with your datasource. The `FirstName` data field property is shown in Listing 4-39.

Listing 4-39. *FirstName Data Field Property*

```
private string _dataFirstNameField = "FirstName";
[Browsable(true), Category("Data"), Description("First Name Data Field"), ➡
DefaultValue("FirstName")]
public string DataFirstNameField
{
    get
    {
        EnsureChildControls();
        return _dataFirstNameField;
    }
    set
    {
        EnsureChildControls();
        _dataFirstNameField = value;
    }
}
```

The `DataFirstNameField` property is actually defined by the `PersonListingControl` to simplify the configuration of the control. When the `PersonRow` creates the child controls, a method called `CaptureSettings` will get the values from the `PersonListingControl`. The `CaptureSettings` method is shown in Listing 4-40.

Listing 4-40. *CaptureSettings Method*

```
/// <summary>
/// Gets from the parent if the parent is the PersonListingControl
/// </summary>
private void CaptureSettings()
{
    PersonListingControl parent = Parent as PersonListingControl;
    if (parent != null)
    {
        personFormat = parent.PersonFormat;
        firstNameField = parent.DataFirstNameField;
        lastNameField = parent.DataLastNameField;
    }
}
```

```

        birthDateField = parent.DataBirthDateField;
        birthDateFormat = parent.DataBirthDateFormat;
        cityField = parent.DataCityField;
        countryField = parent.DataCountryField;
    }
}

```

When the `DataBinder` is called, the data it extracts from the data object is first set on the `FirstName` property. This property is an accessor to a member variable instead of the `Text` property of a control. The controls used by the `PersonRow` are just a series of `LiteralControls` that are initially added to the `Controls` collection during the `CreateChildControls` method without the `Text` value defined. A simplified version of the `CreateChildControls` method is shown in Listing 4-41.

Listing 4-41. *Creating the Controls Collection*

```

ltFirstName = new LiteralControl();
ltLastName = new LiteralControl();
ltBirthDate = new LiteralControl();
ltCity = new LiteralControl();
ltCountry = new LiteralControl();

Controls.Add(new LiteralControl("\n<p>\n"));
Controls.Add(ltFirstName);
Controls.Add(new LiteralControl(" "));
Controls.Add(ltLastName);
Controls.Add(new LiteralControl(", "));
Controls.Add(ltBirthDate);
Controls.Add(new LiteralControl(", "));
Controls.Add(ltCity);
Controls.Add(new LiteralControl(", "));
Controls.Add(ltCountry);
Controls.Add(new LiteralControl("\n</p>\n"));

```

EFFICIENT MARKUP

The `PersonRow` uses the `LiteralControl` instead of a `Label` control specifically because it generates less markup code for the page. This reduces the page size and decreases the page load time for the user. A new control called `ListView` will be included with the .NET 3.5 release coming in early 2008. It will allow the developer to choose exactly what markup will be presented to the web browser, with a focus on offloading styling to an external stylesheet that is loaded a single time with each visit to a website. This reduced overall size will speed up the page requests and make your applications work more quickly.

These `LiteralControls` act as containers that are updated later during the `PreRender` event when the `Text` values are finally set, as shown in Listing 4-42.

Listing 4-42. *Setting the Text Properties in the OnPreRender Method*

```
protected override void OnPreRender(EventArgs e)
{
    base.OnPreRender(e);

    if (String.IsNullOrEmpty(CssClass))
    {
        CssClass = "personRow";
    }

    ltFirstName.Text = FirstName;
    ltLastName.Text = LastName;
    ltBirthDate.Text = BirthDate.ToString(birthDateFormat);
    ltCity.Text = City;
    ltCountry.Text = Country;
}
```

This separation is done to address some issues with ViewState that will be explained in the following section.

Persisting ViewState Manually

Custom controls may not persist state as you would like. To overcome this shortcoming, it is possible to manually handle the ViewState. This is done by overriding the SaveViewState and LoadViewState methods. The SaveViewState method is shown in Listing 4-43.

Listing 4-43. *SaveViewState Method*

```
protected override object SaveViewState()
{
    Pair state = new Pair();
    object baseState = base.SaveViewState();

    object[] objArray = new object[5];
    objArray[0] = _firstName;
    objArray[1] = _lastName;
    objArray[2] = _birthDate;
    objArray[3] = _city;
    objArray[4] = _country;

    state.First = baseState;
    state.Second = objArray;
    return state;
}
```

The structure of the state is completely arbitrary. It is virtually identical to how ControlState works, as shown earlier in this chapter. For a small amount of data, it may be reasonable to just use the Pair or Triplet types. Because there are five values that need to

be serialized here, I chose to use an array of objects set to the values of the member variables. Next the state must be reloaded into the page on the next page request. The LoadViewState method is shown in Listing 4-44.

Listing 4-44. *LoadViewState Method*

```
protected override void LoadViewState(object state)
{
    if (state != null && state is Pair)
    {
        Pair pair = (Pair)state;
        base.LoadViewState(pair.First);

        object[] objArray = (object[])pair.Second;

        if (objArray[0] != null)
        {
            _firstName = (string)objArray[0];
        }
        if (objArray[1] != null)
        {
            _lastName = (string)objArray[1];
        }
        if (objArray[2] != null)
        {
            _birthDate = (DateTime)objArray[2];
        }
        if (objArray[3] != null)
        {
            _city = (string)objArray[3];
        }
        if (objArray[4] != null)
        {
            _country = (string)objArray[4];
        }
    }
    else
    {
        base.LoadViewState(null);
    }
}
```

The LoadViewState method carries out the opposite action as SaveViewState by casting the object to a Pair to access the Second property, which is the array of objects set previously. With these methods in place, the five member variables will be safely stored to ViewState and reloaded before the Load event during the postback request.

This leads into the reason why the Text properties of the LiteralControls were not set in the CreateChildControls method and instead were set in the OnPreRender method. The ViewState used to persist the data for PersonRow will not be available when CreateChildControls is called. But these values are available when the OnPreRender method is run.

Working Without ViewState

To reduce page size, a developer may choose to disable the ViewState on a databound control. If the control does not handle this case, the control will fail to work as expected. Instead of displaying the data as it does on the first page load, it will show empty rows. It is important to note that this databound control may be one of several listed on a page, and a postback event could be caused by any control on the page. While creating this control, I used a test page that had a button that was not attached to an event handler (see Figure 4-4). It simply caused a postback.

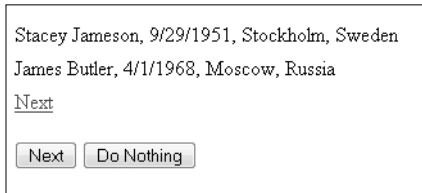


Figure 4-4. *Do Nothing button*

When I first turned off ViewState, it did what I expected because it was specifically using the ViewState accessor to hold onto the data. It was expected that it would fail. One option to ensure this data stays in place is to use ControlState, but doing so would simply move the problem to another place and not allow developers using the control to tune their page size by disabling ViewState. Instead, the problem has to be resolved in the PersonListingControl.

The PersonListingControl has access to the data through the DataSourceID so that it can always retrieve the data if it needs it. When ViewState is disabled, a set of values must be persisted as ControlState. These values include the PageIndex, TotalRowCount, ItemCount, and CommandArgument values for the pager links.

Earlier you looked at the CreateChildControls method for the PersonListingControl in Listing 4-22, which used the `_itemCount` member variable to re-create the right number of items when the `dataBinding` value was false. This is one of the values that will be persisted using ControlState. Listing 4-45 shows the SaveControlState method.

Listing 4-45. *SaveControlState Method*

```
protected override object SaveControlState()
{
    object[] state = new object[6];
    state[0] = base.SaveControlState();
    state[1] = _pageIndex;
    state[2] = _totalRowCount;
    state[3] = _itemCount;
    state[4] = _previousLinkButton.CommandArgument;
    state[5] = _nextLinkButton.CommandArgument;

    return state;
}
```

This limited set of values should be quite small when they are persisted. You may notice the value in the hidden input field where ViewState is placed normally is not empty even though you may have disabled ViewState for the entire page. The reason it is not empty is that ControlState is using the same space as ViewState with hopefully much less space. Loading the ControlState is shown in Listing 4-46.

Listing 4-46. *LoadControlState Method*

```
protected override void LoadControlState(object savedState)
{
    _pageIndex = 0;
    object[] objArray = savedState as object[];
    if (objArray != null)
    {
        InitializePagerControls();
        base.LoadControlState(objArray[0]);
        if (objArray[1] != null)
        {
            _pageIndex = (int) objArray[1];
        }
        if (objArray[2] != null)
        {
            _totalRowCount = (int)objArray[2];
        }
        if (objArray[3] != null)
        {
            _itemCount = (int)objArray[3];
        }
        if (objArray[4] != null)
        {
            _previousLinkButton.CommandArgument = (string)objArray[4];
        }
        if (objArray[5] != null)
        {
            _nextLinkButton.CommandArgument = (string)objArray[5];
        }
    }
    else
    {
        base.LoadControlState(null);
    }
}
```

When I implemented these methods, everything started to work as it did before ViewState was disabled. I knew that I had done absolutely nothing to persist the data for the PersonRow, yet I could click the pager links and the Do Nothing button and would always get the data I expected. It seems everything was handled for me automatically, and I wanted to know exactly how it was being done, so I fired up the debugger.

Walking the Debugger

To see exactly what was happening, I placed breakpoints at the start of each of the event-handler methods and the `CreateChildControl` methods for the `PersonListingControl` and `PersonRow` classes. I then started a debugging process on a page that had a single `PersonListingControl` on the page. Once the debugger was started, I added a few watch items such as `dataSource`, `dataBinding`, `_firstName`, and `IsViewStateEnabled` so I could see their values at each step.

Normally, with `ViewState` enabled, a postback will only cause the `CreateChildControls` method to be called once if a postback event does not specifically cause it to load fresh data, like when the `PageIndex` changes. With the `ViewState` disabled, I found that the `CreateChildControls` method was called a second time during the `PreRender` event with the `dataSource` parameter defined, which caused the `PersonRow` items to get the values they needed. As you can see in Figure 4-5, the `ViewState` is disabled and the `_firstName` member variable has been defined. This value was set during the second call to the `CreateChildControls` method.

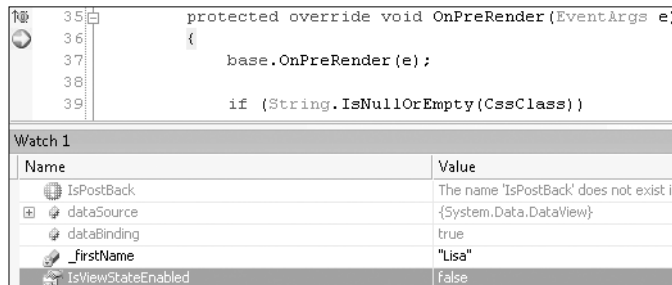


Figure 4-5. *Debugging without ViewState*

With this sample databound control, you can now try your own experiments and watch exactly what happens while walking through the code during a debugging session. You can see how the `ViewState` can impact performance, while moving `ViewState` to the `Session` can offer some benefits with a different set of risks. Using the right combination of databound control settings for your application will give you some performance improvements with minimal effort.

COMMON FOLDER ADDITIONS

This custom databound control is a useful code sample that you could place in your `Common` folder in the `Templates` subfolder (D:\Projects\Common\Templates\Databound Control). While you may not be building a custom databound control, you will occasionally want to create a control that uses `ViewState` more efficiently, perhaps even with `ViewState` disabled. This code can be downloaded with the rest of the sample code for the book.

Summary

In this chapter, you looked at all of the data binding options available to ASP.NET. You looked at several of the standard controls, combined them with a series of user controls, and finally created your own custom databound control. With these choices, you can now decide which databound controls will work well for your needs, and by choosing the right controls along with the optimal settings for ViewState, you can make a more efficient presentation layer.



SQL Providers

The flexibility of ASP.NET 2.0 is largely due to the pluggable architecture of the provider model, which beautifully connects SQL Server, the SQL implementations of the providers, and many ASP.NET controls. Not only are these providers and the controls that use them highly configurable, but they are also completely replaceable if you choose to build your own from scratch to suit unique requirements. It is quite easy to get your ASP.NET website to do exactly what you need it to do by using the available features, which give you a rich base of functionality to get you started with a full bag of tricks.

This chapter covers the following:

- The `SqlMembershipProvider`
- The `SqlRoleProvider`
- The `SqlProfileProvider`
- Building a SQL Photo Album provider
- Building a SQL SiteMap provider

The SQL providers in ASP.NET 2.0 give the platform a great deal of flexibility by pairing a domain model with user-interface components that can be pieced together with other domains and components in a very modular way. Most of the new controls introduced with ASP.NET 2.0 are backed by providers offering multiple implementations that go beyond SQL Server. And if you would like to implement your own provider, there are features in the framework to help you get started with minimal effort. You could even design your own SQL provider that is represented in a Web Form with your own controls. I have found the combination of SQL providers and controls to be a powerful combination when building modular websites.

The providers supplied with ASP.NET were created to address common tasks that were carried out over and over with custom code in classic ASP and ASP.NET 1.1. Such common tasks are managing users and roles and creating the controls to allow these users to create accounts, log in and out, change and retrieve their passwords, and customize the user accounts. As far back as classic ASP, it has been possible to use forms authentication and the database to create all the controls that are offered with ASP.NET 2.0, but you first had to design your database schema and then build all the controls to handle all the tasks users need to do with their accounts. Without a common implementation, developers were left to build their own from the ground up, which took time but also did not offer nearly as much of the rich, highly customized functionality that the ASP.NET 2.0 controls offer.

From the database to the user interface, there are ways to quickly deploy, customize, mix and match, and completely replace the implementations while still maintaining a level of compatibility with other standard components within the system. When you can master all the nuances available with these components, you will be able to produce a website that has all the expected features of a modern website, even if there are unique requirements that were not anticipated by the ASP.NET 2.0 design. It was created with extensibility in mind.

In this chapter, I will walk through the basics of setting up the three most used providers and will show you their flexibility. I will then show you how to implement your own provider, including one provider completely from the ground up that seamlessly integrates with the ASP.NET 2.0 providers and controls.

The SqlMembershipProvider

The most used SQL provider has to be SqlMembershipProvider, which manages users for your website. In the first chapter, you added membership services to the database by using the aspnet_resql.exe utility. With those tables, views, and stored procedures as well as configuration settings in place, you are able to use various controls that use the Membership API to manipulate the data held in your SQL Server. The ones of interest here are the Login controls shown in Table 5-1.

Table 5-1. *Login Controls*

Login	Displays the login form and handles authentication
LoginView	Displays content by using one of two templates for authenticated or anonymous users
LoginStatus	Displays a login link when the user is anonymous, and a logout link when the user is authenticated
LoginName	Displays the user's login name
PasswordRecovery	Displays a form to recover a lost password
CreateUserWizard	Displays a sequence of steps used to create a new user account
ChangePassword	Displays a form to allow a user to change her password

The controls in Table 5-1 are all configured with the settings in the `Web.config` file as well as properties on the controls themselves. The defaults are generally reasonable, but if you have requirements that differ from the defaults, it is not difficult to alter the behavior to meet your needs.

The SqlMembershipProvider is just one of two providers that Microsoft released with ASP.NET 2.0, with the other being the Active Directory implementation. SqlMembershipProvider conforms to the required interface of the base class of MembershipProvider and accurately carries out the equivalent work that the SQL implementation does. And as the names suggest, one uses a directory, while the other uses a database to manage users. This common interface is what the ASP.NET controls use to carry out the same behavior regardless of the underlying implementation.

INTERFACE OR ABSTRACT?

The base classes used by provider implementations are really abstract classes that first inherit from the `ProviderBase` class, which adds the base methods and properties common among all provider implementations. The methods and properties defined by a `MembershipProvider` that must be implemented are marked with the `abstract` attribute. Common functionality between providers could be implemented within the abstract base classes just as the `ProviderBase` sets the `Name` and `Description` properties in the `Initialize` method. You use this base class to define required methods but also to make it easier for a developer to implement his own implementation.

Although both the Active Directory and SQL implementations handle the required methods of the `MembershipProvider` base class, they have several unique configuration settings related to their implementations. For example, the Active Directory implementation has an attribute called `attributeMapEmail`, which is not used by the SQL implementation. The ability to use unique configuration settings will come in handy when you need to create a custom implementation that has requirements going well beyond what was anticipated with these two implementations.

Using XML Implementations

Another alternative to a SQL implementation is XML. You can serialize a `DataSet` to disk as an XML file and load it as you need it. When you create a custom SQL provider, you have to create the tables and stored procedures, which takes more than just programmatically building a `DataSet` in memory and serializing it out to disk as an XML file. In this way, you can build a quick prototype of a custom provider, and after you have controls using it through the provider interface, you can implement a SQL implementation for better scalability. Because the XML implementation is compatible as a provider implementation, it can still be used during development even after you have a SQL implementation. The XML version can come in handy when you are disconnected from the database, voluntarily or otherwise. Switching between the XML and SQL provider is just a matter of changing the `defaultProvider` attribute in the configuration.

Note The XML implementation of the `Membership` provider will not automatically have all the same users that are held by using the SQL implementation, but you will be able to switch between the XML and SQL configurations during development by creating a set of sample users to suit your needs. In Chapter 1, the example in Listing 1-19 shows how users and roles can also be created automatically when the application is started.

Setting the Database Connection

Connecting the `SqlMembershipProvider` to the database is done with the `connectionStringName` configuration attribute. Notice that this is a name and not a connection string. In .NET 1.1 it was common to set the connection string as just another setting in `appSettings`. With .NET 2.0

configuration files, the `connectionStrings` element represents database connections with a name and a `connectionString` attribute. The formalized construct separates connection string settings from other general settings for an application that makes it possible to reach these values programmatically. In the `SqlMembershipProvider` configuration, the `connectionStringName` refers to the name attribute of a connection string, binding the provider implementation to that database. You will discover that you can set multiple connection strings for different database catalogs in your database server and configure each of your provider implementations with a separate database. You could choose to isolate all the ASP.NET-generated resources in a database separate from your application tables for a clear separation and for ease of maintenance.

You may also find that as you are working on a website with these provider configurations, you will want to switch between different copies of databases on your workstation and remote databases on servers at the office. You can configure every one of these database connections and move the name used by the provider configurations to the one you want active. I typically work with a local copy of the database so that I can work without direct interference by someone making changes to a database on the server. You may maintain a common staging database that all developers use to test integration builds. As you complete a set of features on your local copy, you can deploy your database changes to that staging database, reconfigure your website for the staging database, and then run your tests to verify that your changes integrate properly with changes others may have made, before you check your changes in the source control.

Creating a Password Policy

When configuring the `SqlMembershipProvider` for a new website, you must choose how passwords will be handled. It is the first stumbling block to using the `MembershipProvider` because the default password policy is somewhat rigid. It requires a password to be at least seven characters long with at least one nonalphanumeric character. A password such as `aspnet9` would not be allowed, but `aspnet9!` would be allowed. During development I frequently create new accounts and log in to them, so simplifying this requirement, at least during development, is one of my first steps. Depending on the level of security your public site requires, you can ease the password policy and not require the nonalphanumeric character and possibly drop the required length down to six characters. Alternatively, you can configure a regular expression to be used to validate new passwords. Table 5-2 shows a few sample regular expressions you could use.

Table 5-2. *Regular Expressions for Passwords*

<code>^[7,10]\$</code>	The password can have any characters and a length of seven to ten characters.
<code>^[a-zA-Z]\w{7,14}\$</code>	The password must be just letters with a length of seven to fourteen characters.
<code>^(?=.**\d).[7,12]\$</code>	The password must include at least one numeric character and have a length of seven to twelve characters.
<code>^(?=.**\d)(?=.*[a-z])(?=.*[A-Z]).{7,15}\$</code>	The password must have at last one uppercase letter, one lowercase letter, and one digit. Its length must be seven to fifteen characters.

Naturally, you can get much more rigid with the regular expressions requirement but should balance that with what will work for your users. If it is just an online discussion forum about your fantasy football league, you can use a relaxed password policy. But if it is for an SSL-secured website managing bank accounts, you should lean toward the last example.

Note You can get more example regular expressions online at <http://regexlib.com/>. Search for *password*, and you will be given a list of regular expressions for passwords.

The SqlRoleProvider

The SqlRoleProvider is an implementation of the RoleProvider, which provides a way to group users. You are not required to enable the RoleProvider if you are using the SqlMembershipProvider. And if you do have both of them enabled, you can even choose different types of providers. You could configure your website to use the Active Directory Membership provider and the SqlRoleProvider, and it will work as it should through the provider interfaces.

The features of the SqlRoleProvider are pretty simple but very much necessary for most sites. Users are associated with roles, which can be used in various ways to grant or deny access as well as customize the behavior of the website.

Controlling Access by Role

After the website is using the SqlMembershipProvider and the SqlRoleProvider, you will probably want to secure it. To lock down a website completely, you deny access to all anonymous users. With ASP.NET, you do so in the `Web.config` file by using the `authorization` element. That element holds either `deny` or `allow` elements that designate your authentication requirements, whether it is by role or user as well as authenticated or anonymous. Listing 5-1 shows how to lock down a website so that only authenticated users can get in.

Listing 5-1. *Authenticated Users Only*

```
<authorization>
  <deny users="?" />
</authorization>
```

The question mark designates anonymous users. The default authentication is to allow all users regardless of their login status, as shown in Listing 5-2.

Listing 5-2. *Any User Allowed*

```
<authorization>
  <allow users="*" />
</authorization>
```

You can also combine the directives to first allow a user, perhaps by role, and then deny everyone else. Your access control rules can also be limited to a specific location. Listing 5-3 shows a configuration that must be placed outside the main `system.web` configuration section

because the location block cannot be contained by the `system.web` section. Typically, you will set your global access settings in the main `system.web` configuration and add additional, more restrictive rules immediately afterward.

Listing 5-3. *Allow Admin Users Only*

```
<location path="Admin">
  <system.web>
    <authorization>
      <allow roles="Admin"/>
      <deny users="*/>
    </authorization>
  </system.web>
</location>
```

CAREFUL, NOT EVERYTHING IS SECURED!

When adding the authorization requirement to prevent anonymous accounts from reaching your pages, you may not be securing everything. You are really securing only ASP.NET resources that are mapped to run under the .NET runtime. Files with extensions such as `.aspx`, `.asmx`, and `.ashx` are mapped to the .NET runtime, but images and text files are not. The files that are not managed with the .NET runtime are also not protected by this authorization mechanism. Normally, your critical data is on a web page, but keep this in mind if your ASP.NET application generates images or other documents to disk with critical data.

However, when you are working with Visual Studio 2005, you will find that all your resources are protected according to the configuration. This is because everything served up by the development web server is run through the .NET runtime, which respects the settings in `Web.config`. If you do wish to protect your files, you can either specifically map the extension for the file type to the .NET runtime or set the wildcard application maps for every file not specifically mapped in the application extensions section. The latter option may introduce more load on your server, because IIS is better equipped to serve up static image files, so if you are looking to secure generated Excel documents, it would be best to just map the `.xls` extension to the .NET runtime. You can copy the mapping of the `.aspx` extension. It references the .NET 2.0 installation folder ending with `aspnet_isapi.dll`.

There is one exception to the authorization mechanism, and that is for the login page. This page has to be allowed so that forms authentication can give the user access. By default the authentication mode is Windows, but to use the `SqlMembershipProvider` and the Login controls, you must change it to Forms mode. You can decide to use a different page, but `Login.aspx` is the default. You can also change the time-out from the default of 30 minutes to something like 2 weeks (20,160 minutes) and set it to use a sliding expiration so that as long as the user returns to the website within 2 weeks, she stays logged in. This configuration, shown in Listing 5-4, is useful for a portal or shopping website.

Listing 5-4. Login Page Configuration

```
<authentication mode="Forms">
  <forms loginUrl="Login.aspx"
    name=".ASPXFORMSAUTH"
    timeout="20160"
    slidingExpiration="true"/>
</authentication>
```

After the user successfully logs in to her account, she is assigned a token that is kept on the client side as a cookie. This cookie is known as the Forms Authentication token. In Listing 5-4, the cookie name is `.ASPXFORMSAUTH`, the default name. This token is the connection between the client and the back end for the membership system.

Controlling Behavior by Role

As the administrator of your website, you will likely add yourself to a role called `Admin`. In Listing 5-5, you can see how the `Admin` role is used to restrict access to the `Admin` section for users in the `Admin` role. When you log in to the website, you can display a link to the `Admin` section while the same link will not be visible to a regular user. You simply check whether the current user is in the `Admin` role and set the `Visible` property on the hyperlink.

Listing 5-5. Setting Visibility of the AdminHyperlink

```
protected void Page_Load(object sender, EventArgs e)
{
    AdminHyperLink.Visible = Roles.IsUserInRole(User.Identity.Name, "Admin");
}
```

You can take this even further. In the case of a commerce site, when a customer makes a first purchase, you may add him to a role of `Customer`, which will start showing content on his account page for order history and shipment tracking. And if your commerce site works with multiple vendors who supply the products for the website, you may add them to a `Vendor` role and grant them access to update inventory data and projected ship times.

Keep in mind that all the role information is stored in the database along with your application data. Although it is best to use the `RoleProvider` to interact with this data, you can use the underlying stored procedures to interact with the membership data. For a commerce site, you may want to run a monthly report to find all users who have purchased more than \$1,000 in merchandise in the last three months. You can then use the `aspnet_UsersInRoles_AddUsersToRoles` stored procedure to add them to a `PreferredCustomer` role that makes them eligible for discounts and special offers. And as orders come in, you will record the items, quantities, and prices for the order but leave the processing to be handled as a batch later in the day. At that point, you can verify that a user is in the `PreferredCustomer` role and give him his discount by using the `aspnet_UsersInRole_IsUserInRole` stored procedure. Back on the website, the user will start to see special offers on the home page that are not available to everyone else (hopefully winning the user's loyalty), which is all keyed on the user's association with the privileged role.

Another advantage of having these stored procedures accessible is that many platforms work with SQL Server. Your public website may run ASP.NET, but your fulfillment process may

be implemented with a different language or platform that cannot use the .NET framework directly. With all the talk about server-oriented architecture (SOA) as a means to integrate diverse platforms, I feel that the database is already an excellent integration point. When I was building a website a couple of years back, I had a few developers on the team who produced desktop applications with another software platform that handled vendor applications and the fulfillment process. I used the database as a way to safely pass data between platforms. With the stored procedures equally available to the other platform, I did not have to do any additional work for them to adapt to what I was building.

The SqlProfileProvider

The ASP.NET Profile provider adds additional properties to a user account. We touched on this in Chapter 1. What is interesting about the SqlProfileProvider is that it does not require you to also use an implementation of the MembershipProvider. That would seem to make little sense, if you could customize a user account but then could not log in to it and restore your preferences. But that is not exactly how ASP.NET authentication works. The Membership provider is actually just a layer above forms authentication, which has been around for several years. In fact, you can completely implement a membership system with forms authentication directly, which is compatible with the SqlProfileProvider. You can even store usernames and passwords in your Web.config file, as shown in Listing 5-6.

Listing 5-6. *Usernames and Passwords in Web.config*

```
<authentication mode="Forms">
  <forms loginUrl="Login.aspx"
    name=".ASPXFORMSAUTH"
    slidingExpiration="true">
    <credentials passwordFormat="Clear">
      <user name="admin" password="easy12guess"/>
      <user name="user" password="abc123"/>
    </credentials>
  </forms>
</authentication>
```

This all may explain why the Membership, Roles, and Profile interfaces seem mixed up, with so many places to gather the necessary information about your users. The MembershipUser object uses the User object, which has been around since .NET 1.0. But although you can use SqlProfileProvider without the MembershipProvider, there is little reason to do so.

After you are using the SqlProfileProvider, you have a choice to make. You can choose to allow anyone who comes to your website to maintain a profile anonymously or not. This is a common feature, but there are some concerns worth considering.

Note Remember that there are already default providers configured that you will need to clear when configuring your website. Otherwise, you will experience unexpected behavior. Although you can configure and use multiple implementations of a provider, it is not common. Starting your provider configuration with a clear directive should be the first step in configuring your website.

Why Anonymous Profiles?

An anonymous profile can store customization settings associated to a profile just as an authenticated profile can. These settings will stay with the user even after the web browser has been restarted. Perhaps you want to allow a new user to use your website before requiring her to create an account. A common feature on commerce sites allows a user to add products to her basket anonymously, and after she chooses to purchase the items, it creates an account for her during the checkout process so she can return later, log in, and access her order status or make a new order with the previous account information.

More and more websites are also offering small ways to customize their content to suit their users, but these minor preferences are not really worth the effort to create yet another account on another website. One example is the font-resizing buttons on the CNN website that allow you to increase or decrease the size of the text on the page (see Figure 5-1). The CNN site remembers your preference when you return days later. This single setting could be saved alone as a cookie, but when it is so easy to set properties on a profile, you can rely on the anonymous profile.

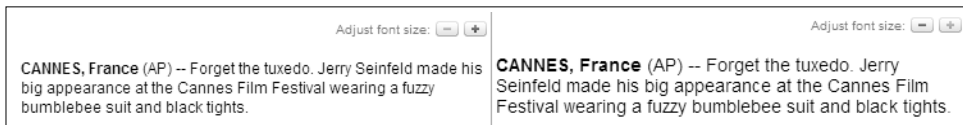


Figure 5-1. *CNN.com font resizing*

You can also use the anonymous profile to keep track of a user by recording the user's actions to the database. Each time the user returns to the website, it will log the visit. And each time the user digs into a section to get to an individual article, you can record that detail. Later you can analyze all this data to get a picture of the users coming to the website. Perhaps you later break down the users into seven groups, assign longtime users to one of those groups, and start to proactively adapt features, content, and advertising on the website to them.

Configuring Anonymous Profiles

Anonymous profiles are not enabled by default, so you will need to add that setting to the configuration (see Listing 5-7).

Listing 5-7. *Anonymous Profiles Enabled*

```
<anonymousIdentification
  enabled="true"
  cookieName=".ASPXANONYMOUS"
  cookieTimeout="144000"
  cookieSlidingExpiration="true"/>
```

This configuration gives the anonymous session a 100-day time-out with a sliding expiration. If you are going to allow for anonymous profiles, you will want to weigh the benefits and the consequences. Although it will be a benefit to have thousands of users with anonymous profiles using your commerce site to add products to your shopping basket system as they

mult over whether to purchase an item, it will also be costly to have all those anonymous profiles taking up space in your database.

Managing Anonymous Profiles

If you set the time-out to four weeks with a sliding expiration window, you can assume that an anonymous account that has not had any activity in four weeks is inactive as the user will not be able to reestablish a connection to that profile. After you can identify these inactive profiles, you can purge them from the system. You could purge some of them even sooner if they do not have significant data stored with the profile. You will find that search engines and other automated programs generate a significant number of anonymous sessions. This classification of anonymous profiles will make up a significant number of the profiles in your database. You could write a query identifying profiles in the database and write a script to purge those profiles, but fortunately the Membership API anticipated this need so you do not have to do all that work.

Listing 5-8 shows the event handler for a button on a Web Form. It can be placed in the code-behind of a page in the admin section of your website. When you want to purge the inactive profiles, you can have it execute the following lines of code.

Listing 5-8. *Delete Inactive Profiles with Button1*

```
protected void Button1_Click(object sender, EventArgs e)
{
    ProfileManager.DeleteInactiveProfiles(
        ProfileAuthenticationOption.Anonymous,
        DateTime.Now.AddMonths(-1));
}
```

This code deletes all anonymous profiles that have not had any activity in the past month. The `ProfileAuthenticationOption` enumeration also includes values of `Authenticated` or `All` to give you access to deleting any profile. This is still a manual process. It would be better to schedule a script to run nightly, to keep a rolling window going that keeps deleting all inactive profiles. When the `MembershipProvider` is backed by SQL Server, you can take advantage of the stored procedure that the `SqlMembershipProvider` calls, `aspnet_Profile_DeleteInactiveProfiles`. You have to pass in the application name, the profile option, and the date since the profile has been active. The profile option is 0 for anonymous, 1 for authenticated, and 2 for both profile types. Listing 5-9 shows the code that deletes the inactive profiles.

Listing 5-9. *Delete Inactive Profiles with T-SQL*

```
DECLARE @PurgeDate DATETIME
SET @PurgeDate = DATEADD(DAY, -60, GETDATE())
EXEC aspnet_Profile_DeleteInactiveProfiles 'appName', 0, @PurgeDate
```

With this script scheduled to run nightly, it should keep your database clear of unnecessary profile data. It is set to go back 60 days, but 14 days may be preferable.

Anonymous and Authenticated Profile Differences

The time-out for anonymous sessions can be different from the time-out for the authenticated sessions, but that is not the only difference. As discussed in the first chapter, custom properties can be allowed to require authenticated profiles. Keeping the anonymous profile properties to a minimum will help keep the size down for the anonymous profile data in the database. The custom properties to record the font size and profile group are shown in Listing 5-10.

Listing 5-10. *Custom Profile Properties*

```
<properties>
  <add name="FontSize" type="String" allowAnonymous="true" defaultValue="10" />
  <add name="ProfileGroup" type="String" allowAnonymous="false" />
</properties>
```

Migrating from Anonymous to Authenticated

When a user creates an account or logs in to your website, that user will carry along with him his anonymous profile. The cookie for it will still be set in his browser, and the profile values associated with the anonymous profile will still be in your database. What I typically want to do is remove that cookie and copy forward any values that make sense.

In the first chapter, I explained that I managed an association to a shopping basket with the Guid value from the authenticated or anonymous profile. But when a user did come to the site from a different computer, add items to an anonymous basket, and then finally log in while making purchases, I did not want to lose the items placed in the anonymous basket. To bring these basket items over to the authenticated account, I handled the `Profile_MigrateAnonymous` event, which is a global application event in `Global.asax`.

But I did not just blindly replace the authenticated basket with the anonymous basket. It was necessary to merge the baskets by adding items. If the user had logged in to that account from home the day before and added a couple of items to the basket and then while at work the next day added more items to a fresh anonymous basket, that user would effectively have two baskets. When logging in, the `Profile_MigrateAnonymous` event fires, and I simply add the items from the anonymous basket to the authenticated basket and display the merged basket. The alternative was to either completely forget the items in the anonymous basket or to replace the authenticated basket with the anonymous basket. Moving the items from the anonymous basket to the authenticated basket was the ideal choice.

Sometimes your profile may hold data that you cannot handle in this way. The simple example of font size preference shows how you may have selected one size when you first created your profile, and when you log in from another computer, you do not just want to override your authenticated profile value with the anonymous value. Perhaps the exception would be if the authenticated value is set as the default and the anonymous value is not the default. In that case, it may be helpful to take in the value to maintain some consistency as the user continues using the website. The code in Listing 5-11 handles the `Profile_MigrateAnonymous` in `Global.asax` to migrate the `FontSize` property and then removes all traces of the anonymous profile.

Listing 5-11. *Migrating an Anonymous Profile*

```

void Profile_MigrateAnonymous(object sender, ProfileMigrateEventArgs e)
{
    ProfileCommon anonymousProfile = Profile.GetProfile(e.AnonymousID);
    string defaultFontSize = "10";
    if (defaultFontSize.Equals(Profile.FontSize) &&
        !defaultFontSize.Equals(anonymousProfile.FontSize))
    {
        Profile.FontSize = anonymousProfile.FontSize;
    }

    // remove the anonymous user, profile, and cookie
    Membership.DeleteUser(e.AnonymousID, true);
    ProfileManager.DeleteProfile(e.AnonymousID);
    AnonymousIdentificationModule.ClearAnonymousIdentifier();
}

```

The last few lines of Listing 5-11 clean up the database so it is not cluttered with those anonymous accounts. I have seen how leaving the anonymous cookie identifier causes the `Profile_MigrateAnonymous` event to fire with each page request. So not only does removing the anonymous profile keep the database clean, it also prevents this event from firing unnecessarily.

Creating a User

The control used to create accounts with the Membership API is the `CreateUserWizard`. It is highly customizable. Beyond customization, you can even completely take over the process with your own Web Form and just work with the Membership API to create users. That involves more work, which you want to avoid with our goal of productivity in mind. Instead of starting from scratch with your own control, you will simply add a step to the `CreateUserWizard`.

For your profile, you want to allow the user to select a primary interest for the website and also the user's preferred font size (as defined in the custom properties as `ProfileGroup` and `FontSize` in Listing 5-10). The `ProfileGroup` is not allowed for an anonymous user. To customize the `CreateUserWizard`, you first add a step to the wizard (see Figure 5-2).



Figure 5-2. *Add a step to the wizard.*

For this customization, you want to place your wizard step at the start of the three-step sequence. Your step will ask for the user's preferences on the custom profile properties. The second step will ask for all the required profile information, and the last step will show the confirmation that the new user profile has been created. Use the markup in Listing 5-12 for the customized control.

Listing 5-12. *Customized CreateUserWizard*

```
<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="CreateCustomUserControl.ascx.cs"
    Inherits="Controls_CreateCustomUserControl" %>
<asp:CreateUserWizard ID="CreateUserWizard1" runat="server"
    OnCreatedUser="CreateUserWizard1_CreatedUser"
    ContinueDestinationPageUrl="~/Default.aspx">
<WizardSteps>
    <asp:WizardStep runat="server" Title="Primary Interest">

        <b>Select your primary interest:</b><br />
        <asp:RadioButtonList ID="rblPrimaryInterest" runat="server">
            <asp:ListItem Selected="True" Value="1">Sports</asp:ListItem>
            <asp:ListItem Value="2">Politics</asp:ListItem>
            <asp:ListItem Value="3">Weather</asp:ListItem>
            <asp:ListItem Value="4">Business</asp:ListItem>
            <asp:ListItem Value="5">Entertainment</asp:ListItem>
        </asp:RadioButtonList>

        <p>
        Please choose the font size below that you feel
        is most readable.
        </p>

        <table><tr><td>
            <b>Sample Sizes:</b>
            <div style="font-size: 10px;">Small Font Size</div>
            <div style="font-size: 12px;">Regular Font Size</div>
            <div style="font-size: 14px;">Large Font Size</div>

        </td><td>
            <asp:RadioButtonList ID="rblFontSize" runat="server">
                <asp:ListItem Value="10">Small</asp:ListItem>
                <asp:ListItem Selected="True" Value="12">Regular</asp:ListItem>
                <asp:ListItem Value="14">Large</asp:ListItem>
            </asp:RadioButtonList>

        </td></tr></table>

    </asp:WizardStep>
</asp:CreateUserWizardStep runat="server">
```

```
</asp:CreateUserWizardStep>
<asp:CompleteWizardStep runat="server">
</asp:CompleteWizardStep>
</WizardSteps>
</asp:CreateUserWizard>
```

The controls holding the values you want to use are in the radio button lists, `rblPrimaryInterest` and `rblFontSize`. You will take their values when the `CreatedUser` event is fired (see Listing 5-13).

Listing 5-13. *CreatedUser Event Handler*

```
protected void CreateUserWizard1_CreatedUser(object sender, EventArgs e)
{
    Profile.FontSize = rblFontSize.SelectedValue;
    Profile.ProfileGroup = rblPrimaryInterest.SelectedValue;
}
```

You can now use the `ProfileGroup` value to show the users content that serves them better and do so with their preferred font size. But you may want to get at this data directly. That can be tricky.

Dynamic Profiles and Profiles as BLOBs

It is tricky to get at profile data for two reasons. First, the data is serialized in a format that is not directly accessible via a simple database query. Second, you cannot create a command-line utility to read profile data that you can run as a scheduled job on a nightly basis to perform analysis tasks on your users because you need the ASP.NET dynamic compiler to generate the user profile.

In the case of the data serialization, the sample properties will be stored in the `aspnet_Profile` table in the `PropertyNames` and `PropertyValues` columns, as you can see in Table 5-3.

Table 5-3. *Serialized Profile Properties*

PropertyNames	FontSize:S:0:2:ProfileGroup:S:2:1:
PropertyValues	143

In `PropertyNames`, it lists the name, type, starting index, and length separated by a colon. In `PropertyValues`, you can use `PropertyNames` to derive that 14 is the `FontSize` and 3 is the `ProfileGroup`. This construct is designed for size optimization and for single-profile access and not for aggregate analysis. If you want to work with this data, you will need to write a rather complex stored procedure. The alternative is to rely on the .NET classes, which are already capable of deserializing this data.

That leads us to the second reason that working with this data is tricky. The ASP.NET dynamic compiler uses a build provider that reads the settings in the `Web.config` file and generates the `Profile` object in memory. This is how you can see IntelliSense support for your custom properties in Visual Studio. Currently this dynamic compiler does not work for class libraries. And if you place code in the `App_Code` folder of your website, you will find you cannot access these dynamic values either.

This leads us back to using the ASP.NET runtime to access these values through the profiles, which you can use from within a Web Form. To get usable data, you will simply go over every authenticated account and generate a comma-separated values (CSV) export file with all the profile data that you can parse and use, as people have been doing with CSV files for years. Listing 5-14 includes the code to export the profile data as a CSV download.

Listing 5-14. *Profiles Exporter*

```
using System;
using System.Web.Profile;
using System.Web.UI;

public partial class ProfilesExporter : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Response.AddHeader("content-disposition",
            "attachment; filename=CustomProfiles.csv");
        Response.ContentType = "text/csv";
        ProfileInfoCollection profiles =
            ProfileManager.GetAllProfiles(
                ProfileAuthenticationOption.Authenticated);
        foreach (ProfileInfo profile in profiles)
        {
            ProfileCommon pc = Profile.GetProfile(profile.UserName);
            Response.Write(String.Format("{0},{1},{2}\n",
                pc.UserName, pc.FontSize, pc.ProfileGroup));
        }
        Response.End();
    }
}
```

Notice that the code in Listing 5-14 adds the content-disposition header, which forces a download and defines a filename ending with the .csv extension. Without this header, the file will likely try to use ProfilesExporter.aspx as the filename. You could change the filename to include a timestamp so you can distinguish it from newer copies of the data from day to day. With this CSV data, you will be able to do your analysis more easily.

Instead of exporting to a CSV file, you could adjust this Web Form to insert the data back into the database with tables you have created for the purpose of data analysis. The approach used here is built to run quickly. In contrast, reading and writing data will slow down the database and extend the time to run the export.

The BLOB storage mechanism used by SqlProfileProvider caused many developers to create their own alternatives. My approach was to create my own set of tables and key them on the Guid used to identify the user. You can also use SqlTableProfileProvider created by Hao Kung. He is one of the engineers working at Microsoft on the ASP.NET team. He put together a provider implementation to resolve the BLOB problem. A quick search on Windows Live (<http://www.live.com>) for SqlTableProfileProvider will send you right to the download page.

Using the Provider-Powered ASP.NET Controls

Now you have configured your website with all the membership features your users will need to log in to your website. The Login control handles that easily. This process is rich with security, built on years of legacy technologies. Although the Membership API and provider model are new to ASP.NET, the mechanisms to authenticate and maintain a session with a user has been with ASP.NET from the beginning. When the user logs in to your website successfully, she is given a token that is stored with her web browser as a cookie. This token acts as a key that gets her into the protected and customized areas of your website.

The authentication token is an encrypted `FormsAuthenticationTicket`, which holds the name of the user, the session expiration time, and other data including custom user data. In the following scenario, I will show how this custom user data can be used to improve the security of an authenticated session and to demonstrate the flexibility of the authentication systems in ASP.NET.

For this example scenario, you are working with a banking website and you are checking on your balance while you are at work. You get called away briefly and do not think to lock your computer. A dishonest coworker sneaks onto your computer and copies the cookie used to authenticate your session with the bank. This cookie at least is not a plain-text value with your account number, which would allow access your account anytime. Instead it is an encrypted form of `FormsAuthenticationTicket`, which holds the expiration time and the date the token was issued. That means this dishonest coworker has a limited time to use that cookie.

What the coworker can do with a sliding expiration is place the cookie into a browser, which will keep updating the expiration time. What it does not change is the last login date, which is held on the server. If your banking website is really secure, the site may check your last login time prior to allowing you to do anything really important, such as scheduling a payment or transferring money to another person's account. Before the website allows you to do that, it could check that you had logged in within a few minutes of the transaction and request that you provide your password if you have not. You can log in even if you already are authenticated, and doing so will reset the last login time. This requirement will stop your coworker from completing those sorts of actions because he has only the cookie, which allows basic access to your account. But all it would take to grant the coworker access to make that transaction would be to wait for you to return to your computer and log in to the banking website again. What would be helpful would be to store your IP address in the token so the website can verify that you logged in recently and from that computer. This is where you will use the custom user data.

The login process can be adjusted to insert the remote address into the `FormsAuthenticationTicket` as user data. Simply drop the Login control onto the design surface of a Web Form. Then create a handler for the `Authenticate` event. The code in Listing 5-15 handles that event and inserts the remote address of the user into the ticket, which can be used to verify the user's location each time that user returns to the website.

Listing 5-15. *Login Authenticate Event Handler*

```
protected void Login1_Authenticate(object sender,
System.Web.UI.WebControls.AuthenticateEventArgs e)
{
```

```

if (Membership.ValidateUser(Login1.UserName, Login1.Password))
{
    e.Authenticated = true;
    string username = Login1.UserName;
    bool persist = Login1.RememberMeSet;
    int timeout = GetLoginTimeout();
    string userData = "remoteAddress=" + Request.UserHostAddress;

    FormsAuthenticationTicket ticket = new FormsAuthenticationTicket(1,
        username, DateTime.Now, DateTime.Now.AddMinutes(timeout),
        persist, userData);
    string encTicket = FormsAuthentication.Encrypt(ticket);
    HttpCookie cookie = new HttpCookie(
        FormsAuthentication.FormsCookieName, encTicket);
    cookie.Expires = ticket.Expiration;
    Response.Cookies.Add(cookie);
    Response.Redirect(
        FormsAuthentication.GetRedirectUrl(username, persist), true);
}
else
{
    e.Authenticated = false;
}
}

```

To get the configured time-out for the user session, I created a method that reads in the configured value from `Web.config`, as shown in Listing 5-16.

Listing 5-16. *GetLoginTimeout Method*

```

private int GetLoginTimeout()
{
    AuthenticationSection authSection =
        ConfigurationManager.GetSection("system.web/authentication")
            as AuthenticationSection;
    if (authSection != null && authSection.Forms != null)
    {
        return authSection.Forms.Timeout.Minutes;
    }
    // return the default
    return 30;
}

```

Now that the ticket has the value, you can check it whenever the user is about to do something more critical than the typical page request, such as making a bank transfer or changing a password. The method in Listing 5-17 checks the remote address.

Listing 5-17. *IsRemoteAddressMatched Method*

```
private bool IsRemoteAddressMatched()
{
    if (User.Identity.IsAuthenticated)
    {
        FormsIdentity identity = User.Identity as FormsIdentity;
        if (identity != null)
        {
            string prefix = "remoteAddress=";
            if (!String.IsNullOrEmpty(identity.Ticket.UserData) &&
                identity.Ticket.UserData.IndexOf(prefix) == 0)
            {
                string remoteAddress =
                    identity.Ticket.UserData.Substring(prefix.Length);
                return Request.UserHostAddress.Equals(remoteAddress);
            }
        }
    }
    return false;
}
```

The method in Listing 5-17 pulls the ticket from the user identity and then extracts the value of the remote address. Finally, it compares the stored value with the actual value from the page request. This added check helps enhance security.

However, you can take this a step further. Perhaps you would like to give the user an option to save his current location as a trusted location so that security can be relaxed when he comes from that location. Perhaps if he goes on a business trip and logs in from his hotel room, the login process will ask him to provide more than just his username and password. It could instead ask for the last four digits of his social security number and home ZIP code. By authorizing trusted locations, you will be able to log and analyze transactions that take place from newly added locations. This approach is enhanced as home broadband connections tend to maintain the same IP address for an extended period of time, so this remote address will not change very often.

PHYSICAL LOCATION BY IP ADDRESS

There are services known as GeoIP, which can tell you the physical location of an IP address. Allowing a user to take certain actions on her account from trusted physical locations instead of just an IP address can enhance the security but also make it easier for the user. If your user lives and works in Chicago, you can grant her a higher level of trust when you know she is logged in from that location. However, if she logs in at 1:00 p.m. from Chicago, and someone else logs in at 1:40 p.m. from Los Angeles, you can assume that something unexpected is happening, flag the account for review, and alert the user of the unexplained activity.

GeoIP attempts to identify every IP address with a physical location, but it is not always going to be accurate—just as you cannot always look up someone's phone number in the phone book. You can also use the blocks of IP addresses that are assigned by country. For example, Korea has IP addresses in the range from 210.220.128.0 to 210.223.255.255. Even when you cannot determine that the user is not in Chicago, you can identify whether that user is reaching the site from Korea, China, Russia, or another country.

USE A SHARED MACHINE KEY FOR WEB FARMS

When placing multiple web servers into a farm to distribute the load for a high-traffic website, it is necessary to use the machine key setting, which is typically unique to a server. This configuration setting includes the validation and decryption keys, which are used for hashing and decryption. The forms authentication token uses these values, and if the machine key is not identical across the servers, you will experience authentication problems as user sessions hop from server to server. This can be partially avoided with sticky sessions, but after a user leaves for an extended period, he may lose his sticky session with the load balancer and hit a different server with different machine key values that will be incompatible.

Other data is also protected by using the machine key, such as the ViewState, which is encrypted to prevent tampering. If a user were to leave a page idle for a long period and then later cause a PostBack, that user might experience an exception due to this cross-server incompatibility.

The machine key is configured in the `Web.config` file and must be generated. Although Microsoft does not provide a utility to generate these values, the MSDN documentation explains how it is done. Multiple utilities to do it can be found online on sites such as The Code Project, which has a utility aptly named the ASP.NET machineKey Generator.

Building a SQL Photo Album Provider

Beyond the standard providers, you can create your own custom provider to fill a specific need. And as you build your own provider, you can have it interact with the interfaces of the standard providers for a seamless integration. Creating your own custom provider allows for multiple implementations. More than one implementation may not be necessary for an internal application, at least not initially.

You could build a component that is used by multiple applications. If you build it by using the provider model, you allow for multiple implementations. This component could interact with a custom order-fulfillment system. Later your company might purchase an order-fulfillment system package that handles the same features. Instead of tightly coupling your applications to this new package, you could create a provider implementation for it. Migrating your applications to it will then just be a matter of adjusting the configuration.

The same scenario could be applied to internal releases of components. The first implementation of a custom provider might interface with a complex set of underlying dependencies. Maybe you have a mountain of legacy code that is difficult to maintain, but you and your team are in the process of cleaning it up to improve maintainability. Later, as the code base has been refactored, you could create a new implementation that works with the updated system. The applications built on top of the provider will not have to change.

Versioning the assemblies can also achieve a similar solution. The difference with providers is that you do not compile your applications directly against the implementation or version. You instead use the provider as an abstraction layer, which will use the configuration to instantiate the desired assembly. Doing so also avoids directly binding one assembly to a specific version of another assembly.

In this section, you will build a provider that manages a photo album via an implementation that uses the popular photo website Flickr.

Provider Requirements

A provider is made up of three essential pieces: the abstract provider, at least one provider implementation, and the configuration to load the implementation. For a custom provider built from the ground up, you will need additional code to load the configuration and give you access to the provider implementations. If you were implementing a custom implementation of an existing provider, such as the `MembershipProvider`, you would use the loading mechanism already in place and just create your provider implementation class.

The first two classes to implement when getting started with a new provider are the configuration and provider collection classes. To facilitate the loading of providers, there is a whole infrastructure to support the process. Using the following classes accelerates the process of creating a provider.

Configuration Section Class

If you were to look in the `Machine.config` file for references to the membership, you would see a section definition that references the `MembershipSection` type, which is the class that the `MembershipProvider` uses to read in its configuration. This class inherits the `ConfigurationSection` class, which you will use to create your own `PhotoAlbumSection` class, starting with Listing 5-18.

Listing 5-18. *PhotoAlbumSection.cs*

```
using System.Configuration;

namespace Chapter04.PhotoAlbumProvider
{
    public class PhotoAlbumSection : ConfigurationSection
    {
        [ConfigurationProperty("providers")]
        public ProviderSettingsCollection Providers
        {
            get { return (ProviderSettingsCollection)base["providers"]; }
        }

        [StringValidator(MinLength = 1)]
        [ConfigurationProperty("defaultProvider",
            DefaultValue = "SqlPhotoAlbumProvider")]
        public string DefaultProvider
        {
            get { return (string)base["defaultProvider"]; }
            set { base["defaultProvider"] = value; }
        }
    }
}
```


The `Providers` property loads the `providers` sub-element, while the `DefaultProvider` property loads the value of the `defaultProperty` attribute. There is really not much work necessary at this point, because the provider infrastructure handles a good deal of the work. Simply creating a property for the `ProviderSettingsCollection` type causes this `AlbumSection` class to function as a provider configuration section, just as with the standard providers, which hold the `add`, `remove`, and `clear` directives within a `providers` element in the configuration. The `Providers` property automatically handles these configuration elements.

Provider Collection Class

Because you can configure multiple instances of a provider, you will want to be able to hold a collection of them. To do so, you will create a class called `PhotoAlbumProviderCollection`, which will inherit the `ProviderCollection` and override the behavior to use the `PhotoAlbumProvider` type instead of the generic `Provider` type. This class is shown in Listing 5-19.

Listing 5-19. *PhotoAlbumProviderCollection.cs*

```
using System;
using System.Configuration.Provider;

namespace Chapter04.PhotoAlbumProvider
{
    public class PhotoAlbumProviderCollection : ProviderCollection
    {
        public new PhotoAlbumProvider this[string name]
        {
            get { return (PhotoAlbumProvider)base[name]; }
        }

        public override void Add(ProviderBase provider)
        {
            if (provider == null)
                throw new ArgumentNullException("provider");

            if (!(provider is PhotoAlbumProvider))
                throw new ArgumentException
                    ("Invalid provider type", "provider");

            base.Add(provider);
        }
    }
}
```

With your own customized provider infrastructure in place, you are ready to start defining the `PhotoAlbumProvider`.

Abstract Provider Class

The abstract provider is the class that directly inherits the `ProviderBase` class, which all providers are built on. This base class defines a baseline of support for all providers, with properties for `Name` and `Description` as well as the `Initialize` method, which is called when a provider is first loaded into an application. The properties and the method are all declared by using the `virtual` attribute so you can override them, but typically you just override the `Initialize` method.

In the abstract provider, you could leave the `Initialize` method untouched and leave it up to the provider implementation to do so, but you may want to add one or more universally available properties for all implementations of the provider, such as an `Enabled` property. You would do so in the abstract provider.

Now you want to start adding abstract property and method declarations to define your service contract, which all implementations must fulfill. For the `PhotoAlbumProvider`, these declarations include methods to insert, change, and delete data related to albums and photos. Listing 5-20 covers the `ProviderAlbumProvider` class.

Listing 5-20. *PhotoAlbumProvider.cs*

```
using System;
using System.Collections.Generic;
using System.Configuration.Provider;

namespace Chapter04.PhotoAlbumProvider
{
    /// <summary>
    /// Photo Album Provider
    /// </summary>
    public abstract class PhotoAlbumProvider : ProviderBase
    {

        #region " Abstract Methods "

        /// <summary>
        /// Gets albums for a user
        /// </summary>
        public abstract List<Album> GetAlbums(string userName);

        /// <summary>
        /// Gets photos for an album
        /// </summary>
        public abstract List<Photo> GetPhotosByAlbum(Album album);

        /// <summary>
        /// Creates an album
        /// </summary>
        public abstract Album AlbumInsert(string userName, string albumName,
            bool active, bool shared);
    }
}
```

```

    /// <summary>
    /// Creates a photo
    /// </summary>
    public abstract Photo PhotoInsert(Album album, string photoName,
        DateTime photoDate, String regularUrl, int regularWidth,
        int regularHeight, String thumbnailUrl, int thumbnailWidth,
        int thumbnailHeight, bool active, bool shared);

    /// <summary>
    /// Updates an album
    /// </summary>
    public abstract void AlbumUpdate(Album album);

    /// <summary>
    /// Updates a photo
    /// </summary>
    public abstract void PhotoUpdate(Photo photo);

    /// <summary>
    /// Deletes an album
    /// </summary>
    public abstract void AlbumDeletePermanent(Album album);

    /// <summary>
    /// Deletes album permanently
    /// </summary>
    public abstract void PhotoDeletePermanent(Photo photo);

    /// <summary>
    /// Moves an album
    /// </summary>
    public abstract void AlbumMove(Album album,
        string sourceUserName, string destinationUserName);

    /// <summary>
    /// Moves a photo
    /// </summary>
    public abstract void PhotoMove(Photo photo, Album sourceAlbum,
        Album destinationAlbum);

    #endregion
}
}

```

The abstract methods in `PhotoAlbumProvider` define the entire scope of what the provider will support.

The Provider Implementation

For the concrete implementation of the provider, you will likely want to read in more configuration settings. A SQL implementation will surely use a configuration attribute called `connectionStringName`, just as `SqlMembershipProvider` and `SqlProfileProvider` do. As you prepare to initialize the provider, you will want to validate the data that you are going to use. First check that the `config` parameter is not null and throw an `ArgumentNullException` if it is. Then make sure the core properties of `Name` and `Description` are in place so that `ProviderBase` can use them. The description may not be defined, so you can give it a default value when it is not. Then you want to call the `Initialize` method on the base class, which is the `PhotoAlbumProvider` in this case. It will eventually lead to the `Initialize` method in `ProviderBase`, which will load these configuration values for the `Name` and `Description` properties.

Next you will deal with just the configuration values that matter to this implementation, which is the `connectionStringName` attribute. Again, you want to ensure that the attribute is defined and that it is associated with a connection string in the connection string's configuration. If everything appears to be in place, you can use it to initialize the database connection that is used throughout the rest of the class. Finally, you will want to check that you have handled all the configured attributes and that no extras have been accidentally included. As you use each configuration value, you remove it from the collection so that in the end you should be left with none. If you do find that there are still attributes left, you can throw an exception complaining that an unrecognized attribute has been given. This may seem a bit rigid, but this may help you discover a misspelled attribute in your configuration that may otherwise go unnoticed, which would lead to unexplained behavior by the provider. Listing 5-21 shows how the SQL implementation is initialized.

Listing 5-21. *Initialize Method for `SqlPhotoAlbumProvider.cs`*

```
public override void Initialize(string name,
    NameValueCollection config)
{
    if (config == null)
    {
        throw new ArgumentNullException("config");
    }

    if (String.IsNullOrEmpty(name))
    {
        name = "SqlPhotoAlbumProvider";
    }

    if (String.IsNullOrEmpty(config["description"]))
    {
        config.Remove("description");
        config.Add("description", "SQL Photo Album Provider");
    }
}
```

```

base.Initialize(name, config);

if (config["connectionStringName"] == null)
{
    throw new ProviderException(
        "Required attribute missing: connectionStringName");
}

connStringName = config["connectionStringName"].ToString();
config.Remove("connectionStringName");

if (WebConfigurationManager.ConnectionStrings[connStringName] == null)
{
    throw new ProviderException("Missing connection string");
}

db = DatabaseFactory.CreateDatabase(connStringName);

if (config.Count > 0)
{
    string attr = config.GetKey(0);
    if (!String.IsNullOrEmpty(attr))
    {
        throw new ProviderException("Unrecognized attribute: " + attr);
    }
}
}
}

```

For a full listing of this code, please refer to Appendix A.

Provider Service Class

You can now take all these pieces and put them together. The provider service reads in the configuration section, loads the providers, and sets the default provider. This is the point where providers set themselves apart from simple assembly versioning. The class starts as a simple string in the configuration and is then brought to life as an instance of a class. It is also where you may be spending some time with breakpoints as you work through issues with your implementations and configurations and as you work out the details.

One important detail to remember here is that a provider should be initialized only once; otherwise, it will throw an exception. And you need to do so in a thread-safe way. To handle these requirements, you will make the `PhotoAlbumService` class a singleton—meaning the default constructor is private, and a static property called `Instance` returns the only instance. And in that property, the providers will be loaded while using the lock statement to ensure thread safety (see Listing 5-22).

Listing 5-22. *PhotoAlbumService.cs*

```
using System.Configuration.Provider;
using System.Web.Configuration;

namespace Chapter04.PhotoAlbumProvider
{
    public class PhotoAlbumService
    {
        private static PhotoAlbumProvider _defaultProvider = null;
        private static PhotoAlbumProviderCollection _providers = null;
        private static object _lock = new object();

        private PhotoAlbumService() {}

        public PhotoAlbumProvider DefaultProvider
        {
            get { return _defaultProvider; }
        }

        public PhotoAlbumProvider GetProvider(string name)
        {
            return _providers[name];
        }

        public static PhotoAlbumProvider Instance
        {
            get
            {
                LoadProviders();
                return _defaultProvider;
            }
        }

        private static void LoadProviders()
        {
            // Avoid claiming lock if providers are already loaded
            if (_defaultProvider == null)
            {
                lock (_lock)
                {
                    // Do this again to make sure _defaultProvider is still null
                    if (_defaultProvider == null)
                    {
                        // Get a reference to the < PhotoAlbumSection > section
```


For the PhotoAlbumProvider, I used unit tests right from the beginning to check each new method and implementation for proper functionality. Running the tests immediately identified breaking changes, whether due to logic error in the C# code or an incorrectly functioning stored procedure. And as I chose to completely rework the implementation, I was able to completely test my changes in a matter of moments by running the unit tests against the provider interface. If I had multiple provider implementations, I could run the same tests against all implementations to ensure compatibility as well.

Please refer to Appendix A for the complete listing of source code for the PhotoAlbumProvider, including the unit testing class as well as all the SQL scripts to create the tables and stored procedures.

The Finished Product

With the SqlPhotoAlbumProvider completed, you can now configure a website to use it and start showing those albums and photos. You just have to place the assemblies in the bin folder for the website and configure the Web.config file to hold the configuration. A sample configuration is shown in Listing 5-23.

Listing 5-23. *Web.config for PhotoAlbumProvider*

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="photoAlbumService"
      type="Chapter04.PhotoAlbumProvider.PhotoAlbumSection, ➡
Chapter04.PhotoAlbumProvider"/>
  </configSections>

  <!-- other configurations -->

  <photoAlbumService defaultProvider="SqlPhotoAlbumProvider">
    <providers>
      <clear/>
      <add name="SqlPhotoAlbumProvider"
        connectionStringName="chpt4"
        type="Chapter04.PhotoAlbumProvider.SqlPhotoAlbumProvider, ➡
Chapter04.PhotoAlbumProvider"/>
    </providers>
  </photoAlbumService>
</configuration>
```

With a few Web Forms wired up to the PhotoAlbumProvider, the photo gallery comes together quickly, as you can see in Figure 5-3. And to fill in some sample photos, I also created an import routine that pulls in photos by tag from Flickr to prefill a few sample albums. (See Appendix A for all source code.)



Figure 5-3. *The finished product*

Building a SQL SiteMap Provider

If you look closely at the finished product of the PhotoAlbumProvider in Figure 5-3, you will see something familiar at the top. It is a breadcrumb trail, which is a standard ASP.NET feature. Unfortunately, the only implementation of the breadcrumb navigation is XML. To make it work with the new photo album, you need a SQL implementation that will adapt instantly to changes in the photo album. Naturally, you want to be able to navigate around the different photo albums, so it makes sense to leverage the navigational controls in ASP.NET that work with the SiteMapProvider.

This time around, it will be a bit easier because you have a couple of details working in your favor. First, the SiteMapProvider already handles the configuration loading for you. And second, you can choose to extend one of the existing SiteMapProvider implementations or implement it from the abstract SiteMapProvider class. Because there was a great example on MSDN showing how to build an implementation from the base, I started with that and adjusted it to my requirements.

Note There is a great deal of content on MSDN for sample code and explanations of the architecture. It continues to improve all the time as Microsoft identifies areas that need more attention. I watched as the documentation was continually augmented as .NET 2.0 was prepared for the launch. And I find that every time I return to MSDN, I find something that was not there before. It now features a wiki so that developers like us can contribute content to MSDN to better enhance the overall document and the platform. If you look today, you will see most pages allow you to move to various versions of the frameworks such as .NET 1.1, 2.0, 3.0, and 3.5.

A site map is a hierarchical construct that must have a single parent node, and each node must have a unique URL. It is important to ensure that when the database is generated to hold the data about the website hierarchy, these rules are followed; otherwise, the site map will not work.

SiteMap Requirements

A SiteMap works like a tree and conforms to the following requirements. First, there must be a single root node. From there, each node must have a parent node that ultimately leads to the root node. Finally, each node must have a unique URL. The requirement for the unique URL can be tricky because you may allow a left node to appear in more than one area of the website, such as a product that is a leather chair. A page displaying leather furniture would point to the chair, and so would the page displaying chairs. The breadcrumb can point to only a single parent, so somehow the URL has to be unique even though it may conceptually exist under two separate parent nodes.

Implementing SiteMapProvider

With a working sample in place, I started to adjust it to fit my database requirements. They are very modest requirements that are easily handled by a single table, as shown in Table 5-4.

Table 5-4. *sm_SiteMapNodes Table*

ID	Bigint (Primary Key)
ParentID	Bigint
Url	Nvarchar(150)
Title	Nvarchar(50)
Depth	Int
Creation	Datetime
Modified	Datetime

To load the SiteMap, I just hard-coded the home page and the main gallery page and then read the table used by the PhotoAlbumProvider. The script used to populate the SiteMap is shown in Listing 5-24.

Listing 5-24. *sm_RepopulateSiteMapNodes.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'sm_RepopulateSiteMapNodes')
BEGIN
    DROP Procedure sm_RepopulateSiteMapNodes
END

GO

CREATE Procedure dbo.sm_RepopulateSiteMapNodes

AS

SET NOCOUNT ON

DECLARE @RootNodeID bigint
DECLARE @AlbumsNodeID bigint

-- reset the table
--TRUNCATE TABLE sm_SiteMapNodes
--DBCC CHECKIDENT (sm_SiteMapNodes, RESEED, 0)
DELETE FROM sm_SiteMapNodes

EXEC sm_InsertSiteMapNode -1, 'Default.aspx', 'Home', 0,
    @RootNodeID OUTPUT
EXEC sm_InsertSiteMapNode @RootNodeID, 'Albums/Default.aspx',
    'Albums', 1, @AlbumsNodeID OUTPUT

DECLARE @Albums TABLE
(
    ID int IDENTITY,
    AlbumID bigint,
    [Name] varchar(50),
    UserName nvarchar(256)
)

INSERT INTO @Albums (AlbumID, [Name], UserName)
SELECT ID, [Name], UserName
FROM pap_Albums
WHERE IsActive = 1

DECLARE @CurID int
DECLARE @MaxID int
DECLARE @AlbumID bigint
DECLARE @AlbumNodeID bigint
DECLARE @Name varchar(50)
DECLARE @UserName nvarchar(256)
DECLARE @Url nvarchar(150)

```

```

SET @MaxID = ( SELECT MAX(ID) FROM @Albums )
SET @CurID = 1

WHILE (@CurID <= @MaxID)
BEGIN
    SET @AlbumID = ( SELECT AlbumID FROM @Albums WHERE ID = @CurID )
    SET @Name = ( SELECT Name FROM @Albums WHERE ID = @CurID )
    SET @UserName = ( SELECT UserName FROM @Albums WHERE ID = @CurID )

    SET @Url = ( 'Albums/Album.aspx?AlbumID=' +
        CONVERT(varchar(10), @AlbumID) +
        '&UserName=' + @UserName )

    -- PRINT 'Name = ' + @Name
    -- PRINT 'UserName = ' + @UserName
    -- PRINT 'Url = ' + @Url

    EXEC sm_InsertSiteMapNode @AlbumsNodeID, @Url, @Name, 2,
        @AlbumNodeID OUTPUT

    SET @CurID = @CurID + 1
END

SET NOCOUNT OFF

GO

GRANT EXEC ON sm_RepopulateSiteMapNodes TO PUBLIC
GO

```

I have a few lines commented out so they are not included when the stored procedure is used normally, but when I need to make updates to it, I find it is helpful to re-enable those lines. As for the TRUNCATE command that is commented out, that requires increased privileges that you may not allow on a production system, so I have it set to use the DELETE command instead. The truncate alternative is useful because when it runs, it does not bother generating audit logs, which can save you time and disk space.

As the @CurID is used to loop over the table variable, the @Url is assembled and the new node is inserted into the sm_SiteMapNodes table by using sm_InsertSiteMapNode. The script to create this stored procedure is shown in Listing 5-25.

Listing 5-25. *sm_InsertSiteMapNode.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'sm_InsertSiteMapNode')
BEGIN
    DROP Procedure sm_InsertSiteMapNode
END

```

```

GO

CREATE Procedure dbo.sm_InsertSiteMapNode
(
    @ParentID bigint,
    @Url nvarchar(150),
    @Title nvarchar(50),
    @Depth int,
    @ID bigint OUTPUT
)
AS

IF NOT EXISTS (SELECT * FROM sm_SiteMapNodes WHERE Url = @Url)
BEGIN
    INSERT INTO sm_SiteMapNodes (ParentID, Url, Title, Depth, Creation, Modified)
    VALUES (
        @ParentID,
        @Url,
        @Title,
        @Depth,
        GETDATE(),
        GETDATE()
    )
    SET @ID = @@IDENTITY
END
ELSE
BEGIN
    SET @ID = ( SELECT ID FROM sm_SiteMapNodes WHERE Url = @Url )
END

GO

GRANT EXEC ON sm_InsertSiteMapNode TO PUBLIC
GO

```

Implementing the `SiteMapProvider` itself is a matter of overriding four abstract methods. There are several other methods on the `SiteMapProvider` abstract base class that orchestrate these four methods, but individually some of those methods can also be overridden, as shown in Listing 5-26.

Listing 5-26. *Abstract Methods on SiteMapProvider*

```

public abstract SiteMapNode FindSiteMapNode(string rawUrl);

public abstract SiteMapNodeCollection GetChildNodes(SiteMapNode node);

public abstract SiteMapNode GetParentNode(SiteMapNode node);

protected abstract SiteMapNode GetRootNodeCore();

```

There is nothing terribly difficult about implementing these methods. The real work is in loading the data from the database and then keeping it current. This work starts in the `Initialize` method in Listing 5-27.

Listing 5-27. *Initialize Method for `SqlSiteMapProvider`*

```
public override void Initialize(string name, NameValueCollection attributes)
{
    lock (this)
    {
        base.Initialize(name, attributes);

        connStringName = attributes["connectionStringName"].ToString();
        db = DatabaseFactory.CreateDatabase(connStringName);
        siteMapNodes = new List<DictionaryEntry>();
        childParentRelationship = new List<DictionaryEntry>();
        EnsureSiteMapLoaded();
    }
}
```

The initial infrastructure is prepared here and then the `EnsureSiteMapLoaded` method is called. This method is actually called at multiple points throughout the class in the event that the SiteMap data is updated and has to be reloaded. Let's take a look at this method in Listing 5-28.

Listing 5-28. *EnsureSiteMapLoaded Method in `SqlSiteMapProvider`*

```
private void EnsureSiteMapLoaded()
{
    if (rootNode == null)
    {
        // Build the site map in memory.
        LoadSiteMapFromDatabase();
    }
}
```

The `rootNode` is required for the SiteMap to work, so setting it to null causes the data to be reloaded the next time this method is called. I created a mechanism to automatically reset the `rootNode` back to null when the data is updated.

The SiteMap data is loaded from that single table by using a single query. Ultimately, this data is loaded from the database and placed into the cache mechanism with a one-hour window to live, with the `CachedItemRemovedCallback` attached. When it is removed from the cache, it clears the `rootNode`, forcing it to get fresh data on the next pass. There are many ways that the cache can be manipulated to invalidate a cached item. For the moment, this uses a very simplistic solution.

The first way the SiteMap data will be reset is via the simple one-hour time-out period, which is helpful to clear up memory when a cached item is not used. I also assumed that I am able to control the SiteMap when a change is made to the photo album database. What I created was a helper class for the `SqlSiteMapProvider`, which removes the item from the cache.

This raises the `CachedItemRemoved` event and clears the `rootNode`. In the code for the website, every time an album is added or removed, the code also calls this helper class to remove the cached item as well as repopulate the `SiteMap` with the new data.

This is not a very elegant approach, but it is functional. In the next chapter, you will learn techniques to manage cached data in far more efficient ways without resorting to manual calls to clear the data.

COMMON FOLDER ADDITIONS

The custom `SiteMap` provider will be useful for many websites because the standard XML provider is not typically sufficient. This sample could be placed in your `Common` folder under the `Templates` folder (`D:\Projects\Common\Templates\Provider Model`). The provider model will also be something that you will want to implement in future projects, so you may want to also include the `Photo Album` provider, which is available with the sample code download for the book (in the `Source Code` area of the Apress website, <http://www.apress.com>).

Summary

This chapter covered the three major ASP.NET SQL providers and how to adjust their behavior to suit your needs. You implemented a provider from the ground up and integrated it with a custom implementation of a standard SQL provider. The topics covered in this chapter showed how your application can use providers as pluggable components. These components are completely interchangeable by design, giving you the flexibility necessary to rework whatever part of your application that you determine needs to be reworked. This chapter also showed you how the interface layer will automatically work with multiple provider implementations if you simply adjust the configuration.



Caching

Perhaps the best way to speed up your applications is to implement a targeted caching strategy. As you profile an application for performance, you will typically find that the time to pull data from the database is the most significant delay during a page request cycle. And the less data you have to pull from the database, the faster your application will perform. The caching mechanisms built into ASP.NET give you a great deal of options when it comes to caching, building on a sound foundation with .NET 1.1 with additional features in .NET 2.0.

Caching is handled in ASP.NET with the `Cache` object, which is a part of the `System.Web.Caching` namespace. And despite the fact that it is in the `System.Web` assembly, it can easily be used in any .NET application including console and desktop applications. You can reference the assembly just like any other. And with an ASP.NET website, the `Cache` object can be used in a class library to create a reusable data access layer.

This chapter covers the following:

- Alternatives to caching
- Caching options
- Problems with caching
- Invalidating a cache
- Performance strategies

The focus of caching in this chapter is on ASP.NET largely due to the stateless nature of HTTP. Many strategies and technologies have been created to overcome the stateless nature of the request-and-response cycle such as cookies and sessions, but the fact remains that websites work in a detached mode. In contrast, a desktop application can load data and keep it in memory reliably with constant access to it. A desktop application also has a single user, which allows the application to take up all resources for that single user. These factors reduce the need for desktop applications to use the same caching techniques employed in web applications.

With a website, you could be servicing many sessions that hold data for each user, each session taking up a memory footprint. If you were to load data into each user's session that takes up 3 MB of space and had 1,000 concurrent sessions, you would already be at 3 GB of memory. By default, a session will be kept active until it times out after 20 minutes of inactivity. While it not uncommon for a server to have 4 GB of memory or more, it is not reasonable to take up that much space. It would be better to allow data to drop out of memory when space is needed, such as when a peak traffic period occurs, as they do with websites, and suddenly there are 5,000 active sessions.

For a typical visit, users may hit a few pages and be gone in less than 10 minutes. Perhaps they are checking your website for some prices on a product or the status of an order. They want to get at it quickly and then leave. Once they do leave, you want to have a way to clear up the memory they were using so it is available for other users. This is where caching fills a real need.

There is also the scenario where a particular set of data is used by many users simultaneously. Perhaps a popular product just went on sale. Each hit on the product page generates a series of queries to the database to display the page. The data does not really change all that often, but in a second you may have 20 users hit the page and run the same queries to load the page. This generates load on the database, causing more and more I/O delay as the database does a great deal of work in a short time frame. It also fills the network connection between the database server and the web server, which could be a bottleneck if the network does not have sufficient capacity to handle the bandwidth. If the product page or the queries were wrapped with a caching mechanism, even if it used just a five-second time-out, you could dramatically improve the performance of the page and reduce the load on the database server and network. This is another real issue that caching addresses.

WEB TO DATABASE SERVER COMMUNICATIONS

The communications between the web server and the database server are easily overlooked during development when a local database is on the same machine as the development web server started by Visual Studio. It can give you a false impression that the data access layer is going to be just as fast once it is deployed. Adjusting queries to just pull the necessary columns from a table instead of all columns is an example of easily reducing the bandwidth between the web server and the database server.

A sudden surge of queries on the database, even if they are the same query, could cause the performance of the database to drop dramatically. This can be caused by the fact that the query joins across multiple tables that happen to be located in different parts of the physical disk. SQL Server 2005 will attempt to compensate by keeping recently used data and indexes in memory, but these desired automatic optimizations are not guaranteed. It is best to save the results of those queries on the web server to allow the database to service other queries.

Alternatives to Caching

Using the ASP.NET cache is not always the right solution to boost performance. The more data you place in the cache, the more complex it can become. And more data in the cache may push out items you want to have in there to enhance performance. Some data can be stored by other means that require less overhead, such as using application state, Session, or even ViewState.

While the techniques described in the following text are sometimes appropriate solutions, they do also have some problems. Either they hold onto objects without an automatic way to release them when memory is in short supply or they do not hold onto the data long enough to be useful. Even so, you can combine the techniques with each other and with caching to provide a more comprehensive solution that gives you all of the benefits with a minimal downside.

Application State

Application state is a global resource that every user and request will be able to access. You can use it to hold onto objects you will want to access quickly. You can place items into this collection with the `Application_Start` event in `Global.asax` so items are available immediately when the application first starts up. Listing 6-1 shows how data can be stored in the application state using the `Global.asax` file in the root folder of the website. The rest of the code can be placed in a utility class in the `App_Code` folder. Listing 6-2 shows the `GetApplicationState` method, which is called by the methods in `Global.asax`.

Listing 6-1. *Loading Data into Application State in Global.asax*

```
void Application_Start(object sender, EventArgs e)
{
    HttpContext.Current.Application["ApplicationState"] = GetApplicationState();
}

void Application_End(object sender, EventArgs e)
{
    HttpContext.Current.Application["ApplicationState"] = null;
}
```

Listing 6-2. *GetApplicationState Method in App_Code\Utility.cs*

```
public static DataSet GetApplicationState()
{
    return HttpContext.Current.Application["Global Data"] as DataSet;
}
```

Without any additional work, the data placed in application state will not be updated. If an external event requires this data to be updated, a handler could be used to reload the data. Listing 6-3 shows a handler (`.ashx`) that can be called to reload the application state.

Listing 6-3. *ApplicationStateHandler.ashx*

```
<%@ WebHandler Language="C#" Class="ApplicationStateHandler" %>

using System.Web;
using Chapter06.ClassLibrary;

public class ApplicationStateHandler : IHttpHandler {

    public void ProcessRequest (HttpContext context) {
        context.Response.ContentType = "text/plain";

        string action = context.Request.QueryString["action"];

        if ("ReloadApplicationState".Equals(action))
        {
```

```

        context.Response.Write("Reloading Application State\n");
        Domain.LoadApplicationState();
        context.Response.Write("Done.");
    }
    else
    {
        context.Response.Write("Action unknown: " + action);
    }
}

}

public bool IsReusable {
    get {
        return false;
    }
}
}

```

Application state can hold multiple items like any collection, so a diverse set of data could be stored for use anytime a request would need it.

Session

A Session holds onto data for an individual user with a sliding 20-minute window. In `Global.asax`, you can use the `Session_Start` and `Session_End` event handlers to manage the life cycle of a Session, which is another collection that can hold any type of object. You can add data to it over time as the user uses the website. A good place to load up a Session is right after a user logs in to a website. It could hold all data relevant to that user that you expect the user will eventually request from the database during the visit.

ViewState

ViewState further limits the scope of the data that it holds to just the current page, for as long as the page maintains a postback cycle. When data is bound to a GridView, it will not have to hit the database on a postback, as it uses the ViewState to draw the GridView. You can also place additional items into the ViewState, which you can access on the postback, that are not attached to page elements. However, this is limited to objects that can be serialized for use in ViewState.

Current Context

You can also hold data in the current context, which is limited to just the current request. You can store data that is then shared by any control through the request process. Consider that if the `Page_Load` in your code-behind populates objects in the context, all user controls included in that page will then be able to use it instead of going to the database to get the data. In the case of a product detail page, you may break up sections of the page into user controls to simplify maintenance, but with a page holding five user controls, you could be hitting the

database to pull product details five separate times for one request. Using the current context would dramatically reduce your database load. Listing 6-4 shows how you can load data into the current context. Listing 6-5 shows how every user control, or any control, on the page can access the data placed in the current context.

Listing 6-4. *Loading the Product into the Current Context in the Page_Load Method*

```
using System;
using System.Web.UI;

public partial class _Default : Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Context.Items["CurrentProduct"] = GetProduct();
    }

    private Product GetProduct()
    {
        Product product = new Product("Product 1");
        return product;
    }
}
```

Listing 6-5. *Using the Current Context to Access the Product in a User Control*

```
using System;
using System.Web.UI;

public partial class Controls_ProductControl : UserControl
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Product currentProduct = Context.Items["CurrentProduct"] as Product;
        if (currentProduct != null)
        {
            Label1.Text = currentProduct.Name;
        }
        else
        {
            // hide the control when there is nothing to display
            Visible = false;
        }
    }
}
```

YOUR OWN WORST ENEMY

I once worked with a developer who hit the database hundreds of times for a single page request without realizing it. These queries caused the page to take a very long time to load. He did not see how calls into certain methods within loops in his code were continually reloading data he already requested previously. By simply holding onto data from early in the same request to be used later in the request, you can speed up your application without making use of the ASP.NET caching system.

One of the side effects of hitting the database hundreds of times for a single page request is an unusually slower response for each query, which adds up quickly. It is like a traffic jam during rush hour, but it all happens in a matter of seconds for a database, which is noticeable when you expect a page to load in under a second. Using the data you recently requested from the database will space out your queries and reduce the frequency of query traffic jams.

Event order is very important when you are loading data and making use of it later. When used with a master page, you may expect the Load event on the master page to run first, but in fact the Load event on the page will fire first, then the master page, and then all of the user controls on the page. I have used this technique along with a custom SiteMap to load the current objects for the page into the current context using the `Url` association to these objects. The `Url` may not be a product, but when it is, I can populate that item, and all user controls can check for it and use it when it is available. To centralize all of this logic, you could place the code to populate these context items in the master page in the `Init` event handler so it precedes the Load event on the page.

SHOULD YOU SKIP PERFORMANCE TUNING?

You should not prematurely optimize a system, because early on you will not know what will need to be tuned. It is important to first measure before you tune, and then again after so you can show the difference. After several iterations, you will start to see patterns in your code emerge that will show you common performance problems that hopefully have a common solution you can apply throughout your code in a consistent way. Attempting to tune an application before it is completed can introduce problems that delay the completion of the project without adding measurable value.

There are also features deep in .NET that address commonly inefficient coding techniques that kick in to automatically speed up your application. By using your normal coding techniques, you will find what may appear to be inefficient will actually perform quite well. But if you attempt to pretune the application, you will not realize that automatic benefit and will put in much more time and energy into the project.

PERFORMANCE TUNING STRING CONCATENATION

A common hallway conversation on tuning is the debate over string concatenation. Should you add strings together or use a `StringBuilder` and append strings together? The conflict is that it is easier and faster to just add strings together with the `+` operator instead of using a `StringBuilder`, but a `StringBuilder` performs better. What quickly ends the debate is someone pointing out that once the code is compiled either way, it is optimized to use a `StringBuilder` anyway. And this is just one of many performance-enhancing mechanisms built into .NET that you can use without changing your current coding practices. A good rule of thumb to remember is that programming code is for people and machine code is for machines. Let the compiler do its job.

As you complete the components that make up your application, you will be able to profile the actual performance and identify bottlenecks. Sometimes tuning the performance will be a simple one-line adjustment to a stored procedure or a few lines of code in your data access layer. Using regular techniques to build your application allows you to be as productive as possible while letting the inefficiencies appear as you near completion. Gradually, as you gain a level of comfort, you will start to fall into the patterns that result in more performant applications right from the start.

Caching Options

There is a dizzying array of options available when it comes to caching. The `Cache` object does not simply hold onto a collection of objects so they are available for fast access. The entire caching system includes a rich set of features that can be used in different scenarios. Primarily caching in ASP.NET breaks down into two groupings: output caching and data caching.

Output Caching

With output caching, you are working at the user interface level. You are caching the output of either a page or a user control. (Caching the output of a user control is known as *fragment caching*. Think of a user control as generating part of, or *a fragment of*, a page.) A page or user control configured for output caching will run normally for the first request and every subsequent request until it expires from the cache, and it will return the same output as the request that placed the content into the cache. Pulling the content from the cache prevents server-side processing in the code-behind, which includes all data binding and database queries.

While there are some major benefits to output caching, it is a bit of a brute-force approach as opposed to a targeted solution, which can take into account the nature of the data that is being consumed. A page or user control can cache many pieces of data that all have unique dependencies, which raises many questions when the data is grouped together for display. With the right combination of the techniques covered here, it should be possible to minimize the downside.

Page Caching

At the highest level you will cache the output of an entire page, including all controls held within the page. The declarative way to enable such caching is to place a directive in the page with the desired settings.

The example in Listing 6-6 will cache the output of the page for a Duration of 60 seconds and will be unique for any parameter. The `VaryByParam` attribute will cause the output cache to produce a fresh output if any of the query string parameters are different when the wildcard (*) is used. You can also specify a specific query string or strings with a semicolon-delimited list of names.

Listing 6-6. Enabled Output Caching

```
<%@ OutputCache Duration="60" VaryByParam="*" %>
```

One of the appealing features about output caching is that not only does it not have to pull data from the database when it returns a cached response, it also does not have to fire any page events because the output from the first request that recorded the output to the cache is used. So not only does it reduce database load, it also reduces processor usage.

The output cache can also be set programmatically in the code-behind, as shown in Listing 6-7, although this approach is not as elegant as the declarative approach. It does give you the option to use logic to set your expiration time, which could come in handy if you want to adapt the cache based on server load.

Listing 6-7. Programmatically Setting a Page for Output Cache

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Cache.SetCacheability(HttpCacheability.Public);
    if (IsPeakLoadTime())
    {
        // cache for ten minutes during peak load time
        Response.Cache.SetExpires(DateTime.Now.AddMinutes(10));
    }
    else
    {
        // cache for two minutes during normal traffic periods
        Response.Cache.SetExpires(DateTime.Now.AddMinutes(2));
    }
    Response.Cache.SetValidUntilExpires(true);

    Product currentProduct = Context.Items["CurrentProduct"] as Product;
    if (currentProduct != null)
    {
        Label1.Text = currentProduct.Name;
    }
    else
    {
        // hide the control when there is nothing to display
    }
}
```



```
Visible = false;  
}  
}
```

Problems with Output Caching

A website deployed with precompiled assemblies has already achieved a great deal of performance optimization from a request-handling standpoint. The slower activities in an application will still be pulling data from the database. By using efficient cache dependencies, you can ensure the data in your cache is always current, but the output cache declarations do not have all of the richness and detail necessary to know when the cache dependencies change. All the efforts to tune your data access layer to ensure the data that is displayed is current may be lost if the output cache holds the cached output for too long. As a result, you may find that output caching is not often a realistic option.

Fragment Caching

Instead of caching an entire page, you may have various sections, some of which need to be updated more often than others. Using output caching in a user control is known as fragment caching. Each user control placed on a page can have distinct output cache settings. Consider a product detail page showing a photo of the product along with various details about the product. I once used fragment caching to display several sections of a product page with five or six user controls. Some of the user controls were cached and others were not. Certain details would not change all that often, such as the product photo, description, dimensions, colors, and vendor association. But other details such as the pricing and availability status could change more regularly. Each user control declared the output cache settings specific to the content being displayed. The product page was a composite of these user controls, and it performed very well during busy traffic periods.

You can combine page caching and fragment caching, but obviously the `Duration` set on the page must be shorter than the user controls with output caching enabled. You could set the page to a value of 10 seconds, while a user control that holds data that rarely changes is cached at 3,600 seconds.

Postcache Substitution

Another output-caching feature is postcache substitution, which allows a page to use output caching while allowing a piece of the page to be updated with each request. This can be done declaratively with the Substitution control. This technique has a limitation: the Substitution control cannot be used in a user control or master page that has output caching enabled, so you cannot combine this technique with fragment caching, at least not directly. You have to instead cache the entire page and use the Substitution control where you need each request to get fresh content.

The Substitution control works differently from a typical ASP.NET control. A typical control like a `Label` lets you set the `Text` property, which the control uses for display along with other properties such as `Font-Bold` and `CssClass`. With a Substitution control, you instead provide a string that is displayed directly like a `Literal` control. This is done with a `MethodName` property that refers to a static method in the code-behind, which takes an `HttpContext` object as the only parameter and returns the string. This string is displayed directly without manipulation or decoration due to other properties on the control.

In the case of a product detail page, you would set the entire page to use the output cache, while each user control used to assemble the page is not. In place of certain details that need to be updated, you would use a Substitution control.

In Listings 6-8 and 6-9, the Label controls will only be updated when the output cache for the page expires, while the Substitution control's values are current with each request.

Listing 6-8. *Substitution Control Markup*

```
<asp:Label ID="lblProductName" runat="server" Text=""></asp:Label><br />
<asp:Label ID="lblProductNumber" runat="server" Text=""></asp:Label><br />
<asp:Substitution ID="subPrice" runat="server"
    MethodName="GetPrice" /><br />
<asp:Substitution ID="subAvailability" runat="server"
    MethodName="GetAvailability" /><br />
```

Listing 6-9. *Substitution Control Code-Behind*

```
private static string GetPrice(HttpContext context)
{
    Product product = GetProduct(context);
    return product.Price.ToString("C");
}

private static string GetAvailability(HttpContext context)
{
    Product product = GetProduct(context);
    return product.Availability;
}
```

If all you are doing is returning a small bit of text, this would be sufficient, but if you instead want to return anything more, you will have to somehow generate all of the content to be returned from this method. Creating a string full of markup is what you try to avoid with ASP.NET development because of all of the helpful visual tools. The way the Substitution control works does not give you a direct way to stay in that visual mode. But this can be done indirectly.

Fragment Caching with Postcache Substitution

With the page set for output caching, you can set a user control to hold a Substitution control, which then loads another user control that holds the content to return through the Substitution method. To simplify this process, I created the SubstitutionFragment class (see Listing 6-10) and placed it in the App_Code folder.

Listing 6-10. *SubstitutionFragment Class*

```
using System.IO;
using System.Text;
using System.Web;
using System.Web.UI;
```

```

public abstract class SubstitutionFragment : UserControl
{
    public abstract void BindToContext();

    private HttpContext _currentContext;
    public HttpContext CurrentContext
    {
        get
        {
            return _currentContext;
        }
        set
        {
            _currentContext = value;
        }
    }

    public virtual string RenderToString(HttpContext context)
    {
        CurrentContext = context;
        BindToContext();
        DataBind();

        StringBuilder sb = new StringBuilder();
        StringWriter sw = new StringWriter(sb);
        HtmlTextWriter tw = new HtmlTextWriter(sw);
        RenderControl(tw);

        return sb.ToString();
    }
}

```

What you should notice first is the fact that the `SubstitutionFragment` class inherits from `UserControl` as the base class and then declares the `BindToContext` method as abstract. Any user control that will be used as a fragment in a substitution will change the inherited class from `UserControl` to `SubstitutionFragment` and implement the `BindToContext` method (see Listing 6-11). The rest of the work is handled by the `RenderToString` method, which converts the user control into a string that can be used with the Substitution control.

Listing 6-11. *User Control As a SubstitutionFragment*

```

using System.Web;

public partial class Controls_ProductDetailSF : SubstitutionFragment
{
    public override void BindToContext()
    {

```

```

        Product product = GetProduct(CurrentContext);
        if (product != null)
        {
            lblProductName.Text = product.Name;
            lblProductNumber.Text = product.ProductNumber;
            lblPrice.Text = product.Price.ToString("C");
            lblAvailability.Text = product.Availability;
        }
    }

    private static Product GetProduct(HttpContext context)
    {
        return Utility.GetProduct(context);
    }
}

```

The context makes it possible to load the product using the query string value that I have placed in a class in App_Code called Utility. This method also works with any other page or user control, which can simply pass in the Context property that is normally populated with the page or user control class (see Listing 6-12).

Listing 6-12. *GetProduct Method in the Utility Class*

```

public static Product GetProduct(HttpContext context)
{
    if (context == null)
    {
        return null;
    }

    Product product = context.Items["CurrentProduct"] as Product;
    if (product == null)
    {
        // use the context.Request to load the Product
        string productIdStr = context.Request.QueryString["ProductID"];
        int productId = -1;
        int.TryParse(productIdStr, out productId);
        Domain domain = new Domain();
        DataSet productDs = domain.GetProductByID(productId);
        if (productDs != null && productDs.Tables.Count > 0 &&
            productDs.Tables[0].Rows.Count > 0)
        {
            DataRow row = productDs.Tables[0].Rows[0];
            product = new Product((int) row["ProductID"]);
            product.Name = (string) row["Name"];
            product.ProductNumber = (string) row["ProductNumber"];
            product.Price = (decimal) row["ListPrice"];
            product.Availability = (string) row["Availability"];
        }
    }
}

```

```

        product.Data = row;
    }
    context.Items["CurrentProduct"] = product;
}
return product;
}

```

With the user control ready to be used as a `SubstitutionFragment`, it can be referenced by a `Substitution` control with the `Substitution` method (see Listings 6-13 and 6-14).

Listing 6-13. User Control Markup

```

<asp:Substitution ID="subProductDetail" runat="server"
    MethodName="GetProductDetail" />

```

Listing 6-14. User Control Code-Behind

```

using System.Web;
using System.Web.UI;

public partial class Controls_ProductDetailSFBridge : UserControl
{
    private static string GetProductDetail(HttpContext context)
    {
        SubstitutionFragment substitutionFragment = (new Page()).LoadControl(
            "~/Controls/ProductDetailSF.ascx") as SubstitutionFragment;
        if (substitutionFragment != null)
        {
            return substitutionFragment.RenderToString(context);
        }
        else
        {
            return "Unable to load control: control is null";
        }
    }
}

```

The abstraction provided by the `SubstitutionFragment` class saves you from having to resort to casting or reflection techniques and also encapsulates some common behavior for all user controls that will be used with this technique.

Data Caching

Data caching allows you to place any object into the cache along with a key to uniquely identify it and several other parameters to customize how the cache manages your data.

Cache Methods

There are a few methods you can use to place items into the cache and then remove them. The Cache object directly includes three methods: Add, Insert, and Remove. These are the only methods you will use to directly manipulate the cache. Beyond these methods, you will use the various types of dependencies you send into the cache as you work with your data.

Add Method

The Add method takes a key and an object along with other parameters to place objects into the cache. If an item is already in the cache, it does not add the new value; instead, it leaves the existing item untouched. This behavior is the reason it is more common to use the Insert method. It is important to realize this key difference between the Add and Insert methods.

You can test the behavior I've just described by adding an item to the cache with a key of `myKey` and a value of 1. Then add another item with the same key with a value of 2. When you pull the value for `myKey` from the cache, it will be 1, as the second add did not replace the first value. This behavior can present a problem, but it can also be used to your advantage in the right scenario. Generally though, you will not attempt to add an item to the cache when one already exists. The normal data-caching pattern is to first check whether an item exists in the cache. If it exists, you use it. If it does not exist, you create it, add it to the cache, and then use it.

Insert Method

The Insert method also takes a key and an object with several parameters to place items into the cache. However, if an item with the same key is already in the cache, it will replace the item and use all of the values provided by the parameters. While the Add method only has one method signature, there are four method signatures for the Insert method.

Remove Method

The Remove method takes the key used to place an item into the cache and removes it from the cache. It is not required that an item marked with that key still exist in the cache for it to be removed, but if it does, the Remove method will return a reference to this object as it is removed. Otherwise, it returns a null reference.

The Cache Index

The cache can also be used as an index. Listings 6-15 and 6-16 show how the index is used to add and get data from the cache. Adding a value in this way uses the Add method and does not give you the option to set the other parameters, so it will use the default cache settings. Getting data from the cache is normally done through the index, as you do not need any parameters to get the value. You just need the key used to place the item into the cache.

Listing 6-15. Adding an Item with the Cache Index

```
Cache["Product-129"] = product;
```

Listing 6-16. Getting an Item with the Cache Index

```
Product product = Cache["Product-129"] as Product;
```

Enumerating Over the Cache

You can loop over the cache entries with the `Cache` object. For monitoring purposes, you may want to log the number of items in the cache broken down by type so you can better fine-tune the performance that the cache is providing. You could also purge all items in the cache by a certain type, as demonstrated in Listing 6-17.

Listing 6-17. *Purging Cached Items by Type*

```
private static int PurgeCacheItemsByType(Type type)
{
    Cache cache = HttpRuntime.Cache;
    List<String> keys = new List<string>();
    foreach (DictionaryEntry entry in cache)
    {
        if (entry.Value != null &&
            entry.Value.GetType().Equals(type))
        {
            keys.Add((string)entry.Key);
        }
    }
    foreach (string key in keys)
    {
        cache.Remove(key);
    }
    return keys.Count;
}
```

While the example in Listing 6-17 could prove useful in the right situation, it is generally better to allow the cache to manage the data based on the parameters provided when you insert items.

Parameters

The cache makes use of several parameters that control the behavior for the cache, specifically for the item being added. The parameters used to place items into the cache form a policy for how the data should be handled. Some data can be assigned a higher priority, which will increase the likelihood that the data will be available when it is requested. But you may also have data that you are caching for the simple convenience of reducing database load. You can give it a lower priority. The following text discusses these parameters and explains how they can be used to influence how the data you place in the cache is handled.

CacheItemPriority

The cache uses the priority to determine which items it can remove when it needs to clear space to free up memory or there are other reasons for automatic removal. Table 6-1 shows the various values for `CacheItemPriority`.

Table 6-1. *CacheItemPriority Values*

Value	Description
AboveNormal	The value just above normal, meaning it is less likely to be removed when freeing up space for memory, etc.
BelowNormal	The value just below normal, meaning it is more likely to be removed when freeing up space for memory, etc.
Default	The same as Normal
High	A value that causes the associated item to be the least likely to be removed before other items
Low	A value that causes the associated item to be the most likely to be removed before other items
Normal	The middle priority, which is used as the default
NotRemovable	A value that prevents the cache from removing the item when automatically purging items to free up space

CacheItemRemovedCallback

The `CacheItemRemovedCallback` property references a method that will be called whenever the item is removed from the cache. This method provides the key used to place the item into the cache, the object itself, along with a `CacheItemRemovedReason`, which indicates why the item was removed (see Table 6-2). Generally, an item is removed because the time-out has been reached and the item just expires, but there are other reasons an item is removed.

Table 6-2. *CacheItemRemovedReason Values*

Value	Description
DependencyChanged	A cached dependency associated with the item has changed.
Expired	The end of the time-out period has been reached.
Removed	The item was removed by request by either the <code>Remove</code> or <code>Insert</code> method.
Underused	The item was removed to clear space.

You may find that items are being removed due to the `Underused` reason, even though you keep placing them back into the cache. This can be frustrating when you know you have only a few small items in the cache and it keeps kicking them out. It is tempting to raise the priority or to just set the priority to `NotRemovable`. But if everything in the cache is set to `NotRemovable`, you lose the advantage that the priority gives you. Instead, you can try adjusting your cache settings.

For the cache settings in the `Web.config` file, there are two values that will matter the most when it comes to removing items from the cache with the `Underused` reason: `privateBytesLimit` and `percentagePhysicalMemoryUsedLimit`. By default, there is no limit for `privateBytesLimit`, and `percentagePhysicalMemoryUsedLimit` will be at 89 percent of total physical memory. Knowing that 89 percent is already a very high value, you may want to simply add more physical memory to the server to improve caching. But if your server is shared by multiple applications, you could set the `privateBytesLimit` to a restrictive setting for some of the applications to free up more space for the ones that need it most.

Finding the right cache settings can be tricky. The cache is managed using your custom settings along with a set of heuristics that are hard to predict. To get a better idea of how your settings will perform, you can test your configuration settings with a simulation.


Cache Simulation

It can be hard to understand how the cache will work with the various parameters described in the previous section. To visualize what it is actually doing, I have created a simulation. Because the cache system can be used outside of a web application, you can create a simple console application that reads in data from the database and places it in the cache. To simulate the load of an active web server, I have set it to run a thread that adds several new items into the cache every few seconds. As activity occurs, it is reported immediately to the console with coloring to indicate the type of activity, such as adding an item being added to or removed from the cache. I set the simulation to use two variables: the time-out between new additions and the number of additions to make at each interval.

The simulations I have run use the sample database from Microsoft called AdventureWorks, which is an example of a commerce database full of products and ordering data. (The database can be downloaded from the Microsoft website.) The simulation starts by reading in a list of all of the products and then placing all of the keys into a collection to be used for randomly adding items to the cache at each interval.

This simulation also behaves differently based on the cache settings. When I set the `privateBytesLimit` to an unreasonably low value, I find that items are removed more frequently due to the `Underused` reason. But as I raise that value or return it to the default, I start to see more items reach the time-out. That causes more to be removed from the cache with the `Expired` reason, which is preferable to the `Underused` reason.

You can rework this simulation for each of your applications as you work to find an optimal compromise between memory and performance. This simulation software reports the results at the end, along with the current cache settings and the duration so you can document the results (see Figure 6-1). This documentation will help justify the purchase of additional memory if that is what the test shows. See the code download for this chapter for the complete source code for the simulation.



```

C:\WINDOWS\system32\cmd.exe
o <Already in cache> Product-367
+ Product-301, System.Data.DataSet
o <Already in cache> Product-4
+ Product-86, System.Data.DataSet
o <Already in cache> Product-459

<q to quit> :
q
Configuration:
0      Private Bytes Limit
89     Percentage Physical Memory Used Limit

Totals:
89     Already in Cache
113    Expired
0      Underused
0      Removed
0      Dependency Changed

Duration: 00:01:07.8075024
Press any key to continue . . .

```

Figure 6-1. Cache simulation

COMMON FOLDER ADDITIONS

The cache simulation can be adjusted to scenarios matching your application. It can be a helpful tool to add to your Common folder in the Tools subfolder (D:\Projects\Common\Tools\Cache Simulation). The full source for the simulator can be downloaded with the sample code for this book.

Invalidating Cached Data

Cached data can be invalidated in multiple ways. Primarily, a time-out is set to force a cached item to expire after a specified point in time. Alternatively, you can set a sliding window for a moving expiration. After a period of inactivity, the sliding window will allow the cached item to be removed from the cache. The settings for absolute expiration and sliding expiration are mutually exclusive. Also in this section, I talk about cache dependencies, which you can use to invalidate items when dependent items change, and about manually removing items from the cache.

Absolute Expiration

My preference has always been to set an absolute time-out on items in the cache. I know that just having an item cached for ten seconds will help improve performance during peak load periods. But setting the time-out to five minutes or more at least ensures that if I were to change values in the database, the content displayed on the website would be updated within the five-minute period. Listing 6-18 shows an example using absolute expiration.

Listing 6-18. *Absolute Expiration*

```
DataSet data = GetItem(3491);
string cacheKey = "Item-3491";
DateTime expiration = DateTime.Now.AddMinutes(5);
Cache.Insert(cacheKey, data, null, expiration,
    Cache.NoSlidingExpiration, CacheItemPriority.Normal, null);
```

Sliding Expiration

Sliding expiration causes the item to stay in the cache as long as the item is accessed within the specified window. For data that does not change often but does get hit frequently, this is a good choice. But theoretically, there is no limit to how long an item could be in the cache if the item is frequently accessed. This could represent a problem that first arises through some unexplained behavior when data does not appear to change after it has been updated in the database. It is generally a good practice to set up a cache dependency with an item when sliding expiration is used. Listing 6-19 shows an example using sliding expiration.

Listing 6-19. *Sliding Expiration*

```
DataSet data = GetItem(3491);
string cacheKey = "Item-3491";
TimeSpan slidingWindow = TimeSpan.FromSeconds(30);
Cache.Insert(cacheKey, data, null, Cache.NoAbsoluteExpiration,
    slidingWindow, CacheItemPriority.Normal, null);
```

Cache Dependency

To quickly remove an item from the cache before the absolute expiration or when the period for the sliding expiration has passed, you can make use of a cache dependency. There are multiple implementations of cache dependencies. You could have an item in the cache bound to a file, and when the file is updated the cached item would be removed from the cache. You can also implement your own cache dependency by creating a class that inherits the `CacheDependency` class. A `SqlCacheDependency` is one of these implementations that can alert your application when data has changed in the database. This implementation will be covered shortly.

Manual Removal

If you keep track of the keys you use to place items into the cache, you can use them to call the `Remove` method on the cache to manually remove the items. And when you call the `Insert` method when an item already exists by the same key, you actually cause the existing item to be removed as you add the new item. Perhaps as users log into your website you load up their data and place it into the cache using a key such as `UserData-JSmith`, and when their session expires or when they log out, you proactively ask the cache to remove such keys as you will not be using them again until they return. The data does not have to be in the cache when you request it to be removed, but because you would know the username, it will be possible to assemble the key used to place the data into the cache. If the data stored for this user is significant, this approach may help make space for other items to remain in the cache longer, improving performance.

SQL Cache Dependencies

Data can be automatically removed from the cache when the data in the database changes by attaching dependencies to the cached items. A dependency will override the expiration or sliding window for the cached items immediately once a database change has been detected.

Using the `SqlDependency` and `SqlCacheDependency`

There are a couple of ways to tie a dependency to a cached item to inform you when the data has changed. The `SqlDependency` is the baseline mechanism that communicates with SQL Server to monitor the database for changes. (Note that `SqlDependency` is not the same as `SqlCacheDependency`.) Table 6-3 shows the members of the `SqlDependency` object.

Table 6-3. *SqlDependency Members*

Name	Type	Description
Id	Guid	Read-only property
HasChanges	Boolean	Read-only property
OnChange	Event	Event

The `Id` and `HasChanges` properties are the only properties on the `SqlDependency` object, and they are read-only. The `OnChange` event is raised when a dependency change occurs that gives you access to the `Id` and `HasChanges` properties on the `SqlDependency` object. Because the `Id` property is created with a random `Guid` value, it is not terribly useful. Listing 6-20 shows an `OnChange` event handler. Because you cannot use the `Id` property to identify the key used to place the changed item into the cache or the item itself, you may choose to just use the `SqlCacheDependency` object instead, which does provide these useful details.

Listing 6-20. *The OnChange Handler Method*

```
void OnChangeHandler(object sender, SqlNotificationEventArgs e)
{
    SqlDependency sqlDependency = (SqlDependency) sender;
    // sqlDependency.Id is a random Guid
}
```

Why Use `SqlDependency`?

There are clear limitations of `SqlDependency` when compared to `SqlCacheDependency`, which manages items in the cache for you. But if you want to ensure that items are not ejected from the cache, you can manage your own collection and use the `OnChange` event to remove items from the collection to keep it current. You will have to understand that items placed in your managed collection will take up memory until you remove those items. You already know the cache will manage memory usage and eject items from the cache when it determines memory space needs to be cleared for new items. Depending on your needs, you may decide to make this trade-off.

To enhance the use of the `SqlDependency`, it will be necessary to get the parameters used in the originating query into the `OnChange` event handler. This can be done with an anonymous method that is created as a delegate. The inline block of code will have access to the parameters of the enclosing method, so it will not be necessary to make use of the `Guid` value on the `SqlDependency`. In Listing 6-21, a method is given an integer value named `productId`, which is used as a parameter passed to a stored procedure that retrieves a product from the database. This integer value is used as the index value for the managed collection.

Listing 6-21. *Anonymous Method with a Delegate*

```
SqlDependency sqlDependency = new SqlDependency(sqlCmd);
OnChangeEventHandler onChangeHandler =
    delegate(object sender, SqlNotificationEventArgs e)
```

```

{
    dataItems.Remove(productId);
};
sqlDependency.OnChange += onChangeHandler;

```

Notice how the signature of the delegate matches the required signature of the `OnChange` event. The block of code also directly references the `productId` even though this event may not fire until long after the enclosing method has passed. This is a powerful technique that can be used for just a single query parameter or many.

To construct the `SqlDependency` requires a standard `SqlCommand` object. Because you are using the Data Access Application Block, you have a `DbCommand` instead. Conveniently, it can be cast as a `SqlCommand` and used to construct the `SqlDependency`.

You can also add more `SqlCommand` objects to the `SqlDependency` with the `AddCommandDependency` method. Listing 6-22 shows how many commands are added to the same `SqlDependency`.

Listing 6-22. Multiple SqlCommand Dependencies

```

SqlDependency sqlDependency = new SqlDependency();
sqlDependency.AddCommandDependency(productSqlCommand);
sqlDependency.AddCommandDependency(pricingSqlCommand);
sqlDependency.AddCommandDependency(inventorySqlCommand);
OnChangeEventHandler onChangeHandler =
    delegate(object sender, SqlNotificationEventArgs e)
    {
        dataItems.Remove(productId);
    };
sqlDependency.OnChange += onChangeHandler;

```

If the data among the three commands in Listing 6-22 actually do invalidate each other, it does make sense to bind them together in this way. However, you may never have a reason to do this. If the pricing data changes, it most likely will not require you to get updated product and inventory data, but the option is there if you need it.

Using the SqlCacheDependency

The `SqlCacheDependency` makes use of the `SqlDependency` to remove data from the cache when a dependency changes. It derives from the `CacheDependency` base class, which is one of the parameters used when inserting an item into the cache. Table 6-4 shows the `SqlCacheDependency` constructors.

Table 6-4. *SqlCacheDependency Constructors*

Parameters	Callback Mechanism
<code>String dbName, String tableName</code>	Polling
<code>SqlCommand sqlCommand</code>	Notifications

Polling

When a `SqlCacheDependency` is created with a database and table name, it will monitor the table for changes. Monitoring the table is done with the polling features that are available with SQL Server 2000 and 2005. The way polling works is by attaching a trigger to the table to be monitored that is fired with each insert, update, and delete statement. The trigger increments a number in a status table for this source table to indicate that the table has been changed. When a table is being monitored, this status table is polled for that number to check whether it has changed.

Enabling Polling for a Table

Polling is enabled using the `aspnet_regsql.exe` utility. I use the script in Listing 6-23 to enable the `Production.Product` table in the AdventureWorks sample database. First the services to manage dependencies must be enabled and then the specific table must be enabled.

Listing 6-23. *Add SQL Cache Dependencies.cmd*

```
@echo off

set REGSQL="%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_regsql.exe"
set DATABASE=AdventureWorks
set DSN="Data Source=.\SQLEXPRESS;Initial Catalog=%DATABASE%;
    Integrated Security=True"

echo Registering dependencies
%REGSQL% -C %DSN% -ed

echo Registering Production.Product table
%REGSQL% -C %DSN% -et -t Production.Product

Pause
```

Unfortunately, the script in Listing 6-23 breaks on the first run because the `Product` table is in the `Production` schema instead of the default `dbo` schema. This can be fixed by adjusting the stored procedure used by the utility to make it recognize the fact that the target table does not exist in the `dbo` schema. Once the database has been enabled for dependencies, a stored procedure named `AspNet_SqlCacheRegisterTableStoredProcedure` is created. This stored procedure must be updated to allow for the `Production.Product` table to be monitored with polling (see Listing 6-24). (Thanks go to Chris Benard [<http://chrisbenard.net/>], who published this updated script on his blog.)

Listing 6-24. *Updated AspNet_SqlCacheRegisterTableStoredProcedure*

```
ALTER PROCEDURE [dbo].[AspNet_SqlCacheRegisterTableStoredProcedure]
    @tableName NVARCHAR(450)
AS
BEGIN
```

```

DECLARE @triggerName AS NVARCHAR(3000)
DECLARE @fullTriggerName AS NVARCHAR(3000)
DECLARE @canonTableName NVARCHAR(3000)
DECLARE @quotedTableName NVARCHAR(3000)
DECLARE @schemaName NVARCHAR(3000)

/* Detect the schema name */
IF CHARINDEX('.', @tableName) <> 0 AND CHARINDEX('[', @tableName) = 0
    OR CHARINDEX('[', @tableName) > 1
    SET @schemaName = SUBSTRING(@tableName, 1, CHARINDEX('.', @tableName) - 1)
ELSE
    SET @schemaName = 'dbo'

/* Create the trigger name */
IF @schemaName <> 'dbo'
    SET @triggerName = SUBSTRING(@tableName,
        CHARINDEX('.', @tableName) + 1, LEN(@tableName) -
        CHARINDEX('.', @tableName))
ELSE
    SET @triggerName = @tableName
SET @triggerName = REPLACE(@triggerName, '[', '__o__')
SET @triggerName = REPLACE(@triggerName, ']', '__c__')
SET @triggerName = @triggerName + '_AspNet_SqlCacheNotification_Trigger'
SET @fullTriggerName = @schemaName + '[' + @triggerName + ']'

/* Create the canonicalized table name for trigger creation */
/* Do not touch it if the name contains other delimiters */
IF (CHARINDEX('.', @tableName) <> 0 OR
    CHARINDEX('[', @tableName) <> 0 OR
    CHARINDEX(']', @tableName) <> 0)
    SET @canonTableName = @tableName
ELSE
    SET @canonTableName = '[' + @tableName + ']'

/* First make sure the table exists */
IF (SELECT OBJECT_ID(@tableName, 'U')) IS NULL
BEGIN
    RAISERROR ('00000001', 16, 1)
    RETURN
END

BEGIN TRAN
/* Insert the value into the notification table */
IF NOT EXISTS (SELECT tableName FROM
    dbo.AspNet_SqlCacheTablesForChangeNotification
    WITH (NOLOCK) WHERE tableName = @tableName)
    IF NOT EXISTS (SELECT tableName FROM

```

```

dbo.AspNet_SqlCacheTablesForChangeNotification
WITH (TABLOCKX) WHERE tableName = @tableName)
INSERT dbo.AspNet_SqlCacheTablesForChangeNotification
VALUES (@tableName, GETDATE(), 0)

/* Create the trigger */
SET @quotedTableName = QUOTENAME(@tableName, '')
IF NOT EXISTS (SELECT name FROM sysobjects WITH (NOLOCK)
WHERE name = @triggerName AND type = 'TR')
IF NOT EXISTS (SELECT name FROM sysobjects WITH (TABLOCKX)
WHERE name = @triggerName AND type = 'TR')
EXEC('CREATE TRIGGER ' + @fullTriggerName + ' ON ' + @canonTableName + '
FOR INSERT, UPDATE, DELETE AS BEGIN
SET NOCOUNT ON
EXEC dbo.AspNet_SqlCacheUpdateChangeIdStoredProcedure N' +
@quotedTableName + '
END
')
COMMIT TRAN
END

```

The Add SQL Cache Dependencies.cmd script can be adjusted to update the stored procedure inline so that it does not fail using the OSQL command-line utility, which is a part of SQL Server 2005 (see Listing 6-25).

Listing 6-25. *Add SQL Cache Dependencies.cmd (Revised)*

```

@echo off

set REGSQL="%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_regsql.exe"
set OSQL="C:\Program Files\Microsoft SQL Server\90\Tools\Binn\osql.exe"
set SCRIPTS_DIR="AdventureWorksDatabase\AspNet Scripts"
set UPDATE_SCRIPT=AspNet_SqlCacheRegisterTableStoredProcedure.sql
set DATABASE=AdventureWorks
set DSN="Data Source=.\SQLEXPRESS;Initial Catalog=%DATABASE%;
Integrated Security=True"

echo Registering dependencies
%REGSQL% -C %DSN% -ed

echo Updating AspNet Stored Procedure for Schema Support
%OSQL% -S .\SQLEXPRESS -E -d %DATABASE% -i ➡
%SCRIPTS_DIR%\%UPDATE_SCRIPT%

echo Registering Production.Product table
%REGSQL% -C %DSN% -et -t Production.Product

pause

```


Once this stored procedure is updated, the script to enable dependency polling on this table will run without failing. And to again clear the resources added to the database, you can run the script in Listing 6-26.

Listing 6-26. *Remove SQL Cache Dependencies.cmd*

```
@echo off

set REGSQL="%windir%\Microsoft.NET\Framework\v2.0.50727\aspnet_regsql.exe"
set DATABASE=AdventureWorks
set DSN="Data Source=.\SQLEXPRESS;Initial Catalog=%DATABASE%; ➡
Integrated Security=True"

%REGSQL% -C %DSN% -dd

pause
```

Configuring the SqlCacheDependency for Polling

Once the database is prepared for polling, you must also configure the application to make use of it. The configuration block shown in Listing 6-27 belongs within the `system.web` block and creates a mapping to the AdventureWorks database, which is designated with the `aw` values as the connection string name as well as the name of the cache dependency.

Listing 6-27. *Configuration SqlCacheDependency*

```
<casting>
  <sqlCacheDependency enabled="true">
    <databases>
      <add name="aw" connectionStringName="aw" pollTime="15000"/>
    </databases>
  </sqlCacheDependency>
</casting>
```

The `pollTime` attribute is optional. It controls how often the status table is checked. This value is in milliseconds, but I would highly discourage the value ever dropping below one second (1,000 ms). In fact, I would make the poll time much higher, such as ten seconds or a full minute. And if you are using absolute expiration, you would naturally want the poll time to be much shorter than the caching window, while a sliding window may never invalidate cached items unless polling indicates to the cache that a dependency has changed.

Now that polling is enabled and configured, the code snippet in Listing 6-28 will create a polling cache dependency and insert the item into the cache.

Listing 6-28. *Creating a Polling Cache Dependency*

```
Cache cache = HttpRuntime.Cache;
CacheDependency cacheDependency =
    new SqlCacheDependency("aw", "Production.Product");
cache.Insert(cacheKey, dataSet, cacheDependency,
```

```
DateTime.Now.AddSeconds(120), Cache.NoSlidingExpiration,  
CacheItemPriority.Normal, removedCallback);
```

Notice the `aw` parameter references the configured cache dependency, and the table fully qualifies the `Production.Product` table. The cached item is given an absolute time-out of 120 seconds, while the cache dependency will poll the status table for changes every 15 seconds. If the data changes immediately after the query pulls the data, it will not get a change until at least 15 seconds later.

Problems with Polling

While polling is a simple and reliable way to invalidate data held in the cache, it can be a bit resource intensive if the poll time is very low. And because the poll time is configured for the entire database and not unique for each monitored table, it is likely you will set the poll time to the lowest number necessary for the table that needs to remain the most current. A shorter poll time means the database will be hit more often.

Polling is also not a fine-grained approach. The cache dependency will change for any insert, update, or delete in the monitored table regardless of whether the modified data is actually held in the cache. If you have a table with 50,000 rows and 10,000 of those rows are currently held in the cache, an update to a single record will cause all of those 10,000 rows to be removed from the cache.

Query Notifications

The other option to remove items from the cache automatically prior to their expiration is query notifications. Instead of constantly checking whether the status of a table has changed, query notifications register a query with the notifications system and request to be notified if the resources referenced in the query change. This is an immediate notification without delay and also specific to the scope of the query, which makes it ideal when the targeted data is time sensitive. It also means it is more fine-grained than the polling approach, which cannot distinguish changes of one row from another in the same table.

Contrasting Polling and Notifications

Choosing between polling and notifications will be a critical decision when you approach caching. While one or the other may be a perfect solution for your scenario, there are some details to consider. What is appealing about polling is the fact that it is so easy to set up, but as the examples in the previous section show, the change dependency is not known by the application until the poll time is reached. And each change is triggered by any change made to the table.

Notifications do alert you immediately to a changed dependency with greater specificity of the rows you are concerned about, but the valid queries are limiting. Notifications can also become quite intense on a system where the data changes frequently. Where the choice of polling or notifications alone does not sufficiently solve your performance needs, you can consider a hybrid or custom approach.

The polling solution works by placing a trigger on each table, which is monitored for each insert, update, and delete. Each time a change is made, a counter is incremented on a status table. Then the polling mechanism monitors the counters for changes. A custom solution

could make use of stored procedures that handle all inserts, updates, and deletes to your data, and a more fine-grained status table can be updated as changes are made so that a single row does not invalidate all of the cached items from that table. Perhaps you have products organized into categories, and when a single category changes, you just invalidate the cached items from the affected category.

With your own efficient status table, you can use notifications to monitor changes and remove cached items as necessary. Variations of this custom solution will allow you to adapt a high-performance solution to your application.

Enabling the Service Broker

To get query notifications set up, you must take several steps to prepare your database. This can be a tricky process, but the following steps should get you up and running shortly. Query notifications work on top of a notifications system that uses the Service Broker, which is a feature of the SQL Server database. This feature goes well beyond just monitoring queries, and for security reasons it is not enabled by default when you first install your database. This is true for SQL Server 2005 and SQL Express. To enable it, you would use a utility called SQL Server Surface Area Configuration for Features, which has a link at the bottom called Surface Area Configuration for Features. After a fresh installation, you will see that when you dig into the Service Broker node under the Database Engine node, you cannot enable the Service Broker because an endpoint has not been created yet. The necessary endpoint requires the *master key* to be defined. This can all be created with the script in Listing 6-29.

Listing 6-29. *Create Service Broker Endpoint.sql*

```
CREATE MASTER KEY ENCRYPTION BY password = 'CHANGE_ME';
GO

use Master;
GO

CREATE CERTIFICATE [SERVER_NAME]
    with subject = N'SERVER_NAME';
GO

CREATE ENDPOINT [ServiceBroker]
    state = started as tcp (listener_port = 4022) for service_broker
    (authentication = certificate [SERVER_NAME]);
GO
```

Naturally, you should change the password used to create the master key and also set it to use your computer name in place of `SERVER_NAME`. With the endpoint in place, you can restart the Surface Area Configuration for Features utility and check the status of the Service Broker. With the endpoint defined, you will now see the endpoint that was created in this script, and the state will either be started or stopped. It may be necessary to restart your instance of SQL Server for this change to take effect.

You are now ready to enable the Service Broker. When you first create a new database, the Service Broker will already be enabled. For the AdventureWorks database, the Service Broker is not enabled. It is enabled with the script shown in Listing 6-30.

Listing 6-30. *Enable Service Broker.sql*

```
ALTER DATABASE AdventureWorks SET ENABLE_BROKER
```

If also goes well, this command should run fairly quickly, but you may find that it takes an extremely long time to complete due to a deadlock. To force the command to run, you can add `WITH ROLLBACK IMMEDIATE` to the end of the command and rerun the script. To check whether the Service Broker is enabled, you can run the query in Listing 6-31.

Listing 6-31. *Service Broker Query.sql*

```
SELECT name, is_broker_enabled FROM sys.databases
```

This query will list all of your databases with a 1 or 0, the former indicating the Service Broker is enabled and the latter indicating it is disabled.

Granting Permissions

In order to make use of the notifications system, it will be necessary to grant the user the ability to create procedures, queues, and services in the database (see Listing 6-32). Several other permissions are also necessary.

Listing 6-32. *Grant Permissions.sql*

```
GRANT CREATE PROCEDURE to [USERNAME]
GRANT CREATE QUEUE to [USERNAME]
GRANT CREATE SERVICE to [USERNAME]

GRANT REFERENCES on
CONTRACT::[http://schemas.microsoft.com/SQL/Notifications/PostQueryNotification]
to [USERNAME]
--(note that the schema is case sensitive)

CREATE SERVICE SqlQueryNotificationService ON QUEUE ServiceBrokerQueue

GRANT VIEW DEFINITION to [USERNAME]

EXEC sp_addrole 'sql_dependency_subscriber'

GRANT SUBSCRIBE QUERY NOTIFICATIONS TO [USERNAME]

GRANT REFERENCES on
CONTRACT::[http://schemas.microsoft.com/SQL/Notifications/PostQueryNotification]
to [USERNAME]
```

```
EXEC sp_addrolemember 'sql_dependency_subscriber', 'USERNAME'
```

```
GRANT SELECT TO [USERNAME]
```

```
GRANT SUBSCRIBE QUERY NOTIFICATIONS TO [USERNAME]
```

```
GRANT SEND ON SERVICE::SqlQueryNotificationService TO [USERNAME]
```

```
GRANT RECEIVE ON QueryNotificationErrorsQueue TO [USERNAME]
```

Simply change USERNAME to the username specified in the connection string. The user will be granted the necessary permissions to create the resources required for notifications.

COMMON FOLDER ADDITIONS

You will need to run the scripts to enable SQL notifications each time you want to make use of this feature of SQL Server. You can place these scripts into your Common folder under the Scripts subfolder (D:\Projects\Common\Scripts\SQL Notifications).

Requirements for Queries

Query notifications will not work for all queries. First, a query must reference the tables qualified with a two-part name, such as in Listing 6-33.

Listing 6-33. *Valid Query for Notifications*

```
SELECT
    ProductID, [Name], ProductNumber, Color, ListPrice,
    SellStartDate, SellEndDate, DiscontinuedDate
FROM Production.Product
WHERE ProductID = @ProductID
```

There is also a long list of operators and expressions that will not be allowed with a query notification request. The following are the commonly used functions, operators, and expressions that cannot be used with query notifications.

- A count(*) aggregate
- AVG, MAX, MIN
- The TOP clause
- The DISTINCT keyword
- The UNION operator
- Subqueries
- A SUM function that references a nullable expression
- An INTO clause

The rule of thumb to follow is to create a query that has a limited scope. It makes perfectly good sense that you cannot include the average `ListPrice` from the `Production.Product` table in the AdventureWorks database because it would require monitoring all rows in the table.

You will also want to consider that monitoring a table with query notifications will degrade the performance of table updates. And if the data that is being monitored is updated frequently by multiple applications, it may be best to choose another way to ensure data in the cache is kept current, as the notifications will negate the value of caching the data. You may not be able to cache the frequently updated data, but instead limit the caching to the data that does not change frequently. Doing so will at least reduce the overall load on the system, as less data is being pushed around.

Due to all of the special cases that are not supported, it is necessary to verify that each query is valid. If a query is not valid, a notification will immediately fire the `OnChange` event, which will remove the item from the cache.

Troubleshooting Query Notifications

Notifications can be a black box, much like the cache itself. Knowing what is happening and how changes to the code base will affect the actual performance of the cache and notifications should be carefully monitored to ensure these black boxes are functioning as expected. There are tools that you can use to take a peek into the black boxes to monitor activity.

Troubleshooting with the SQL Server Profiler

The SQL Server Profiler for SQL Server 2005 can be used to see what is happening as queries are sent into the system and how the notifications system responds by tracing the internals of the system. Unfortunately, SQL Express Management Studio does not include the profiler, so you will need to use the full version. Most of the examples so far use SQL Express because I personally prefer to just run SQL Express on my development system and only install SQL Server 2005 on another system that I use as needed. I have found that SQL Express does not tie up resources as much as SQL Server 2005. When it comes to tracing, I simply deploy my changes to the other computer (Virtual PC) and run a trace to get the information I need. Normally, I only need to run traces when I cannot diagnose a problem by other means, such as watching the monitor utility covered earlier in the chapter.

To capture information specific to notifications, it is necessary to create a custom template in the profile. Start the SQL Server Profiler and click the **New Template** icon. Then click the **Events Selection** tab to get a listing of all of the available events (see Figure 6-2). For notifications, you are interested in the **Broker** and **Query Notifications** events. Click the check box in the leftmost column to enable all of the events for those two event groupings. Return to the **General** table and enter **Notifications** as the name of the new template and save it.

Now start a new trace by clicking the **New Trace** icon. Select the newly created **Notifications** template and click the **Run** button (see Figure 6-3).

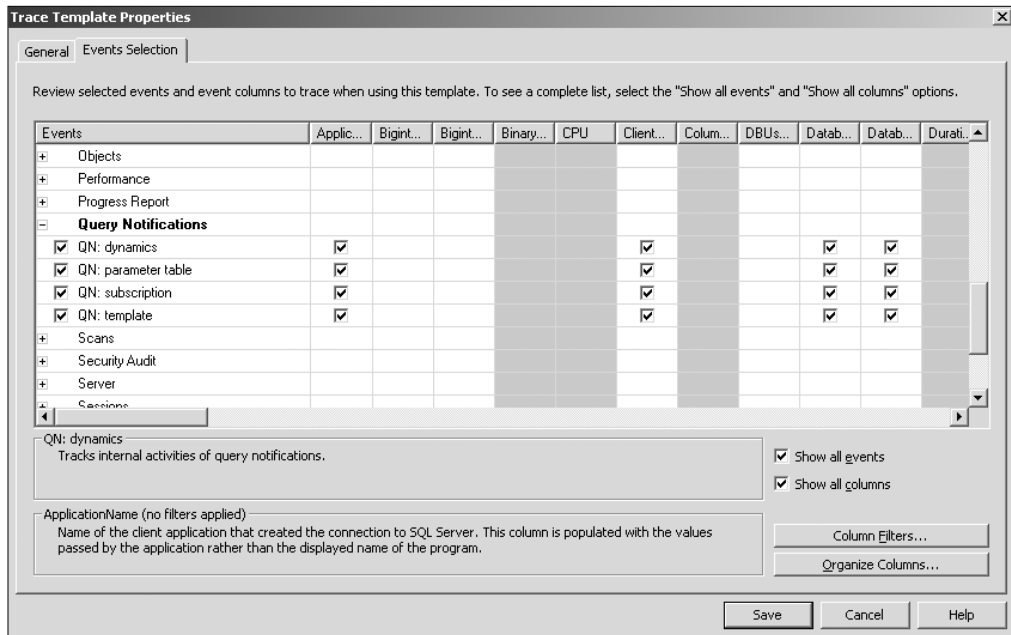


Figure 6-2. Custom tracing template for notifications

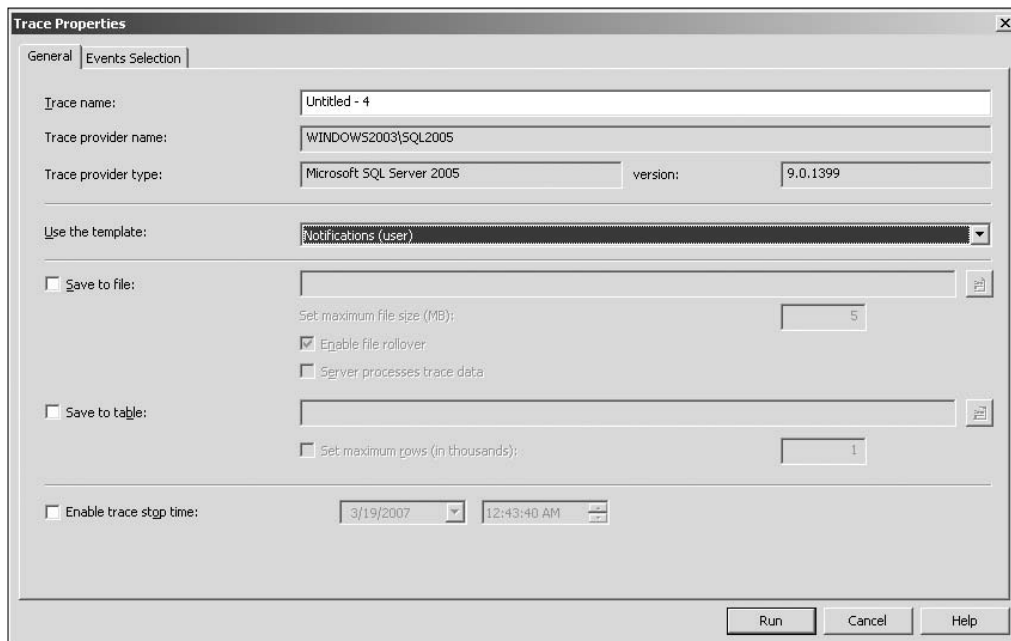


Figure 6-3. Running the notifications trace

Once the trace has started, you can launch your application and watch the events as they happen. Figure 6-4 shows a trace of a functioning notification system. The broker and query notification systems are showing the right sequence of events in the trace, and the application is behaving as it should. If it were failing, you may see the broker starting and ending a “conversation” immediately. This will indicate that the query notification rejected your query. Perhaps it had an aggregate function or another SQL feature it could not support. You can update the query you are testing by trimming it down and restarting this sequence to identify which part of the query is being rejected.

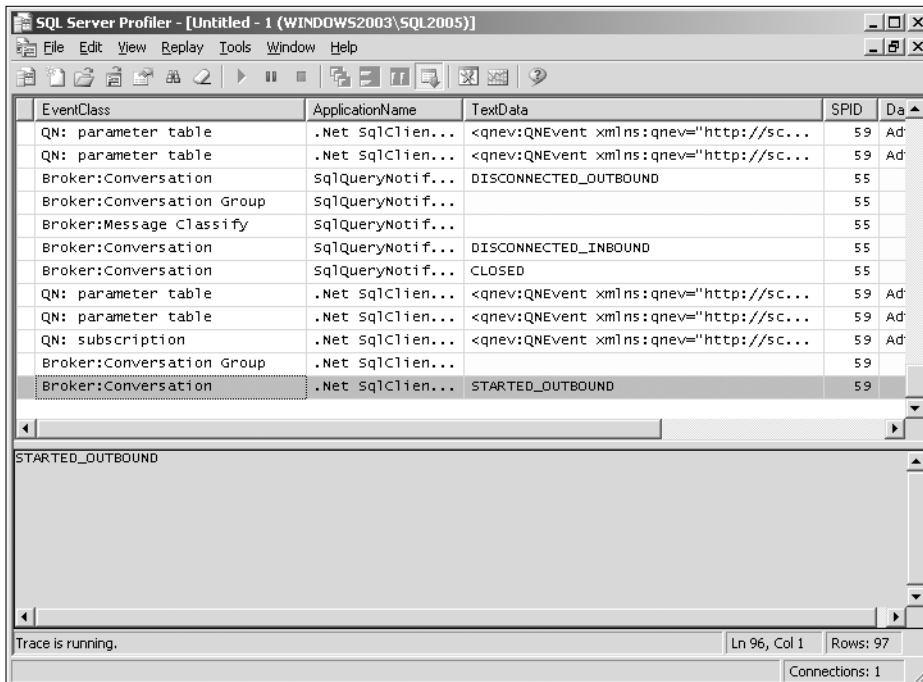


Figure 6-4. *Watching the trace*

You do not have to stop the trace, but it is helpful to pause it and click the Clear button, which is directly to the left of the Play button. Click play before launching your application again to resume watching the trace.

Once you have your application working as you like, you can save the trace to a file to use as a comparison later if you ever need to compare behavior. Simply save the trace from the File menu. Later, you can open the saved trace file in the profiler alongside another freshly created trace.

Troubleshooting via Unit Tests

For a more automated approach to testing notifications and the caching system in general, you can make use of unit tests. If you expect that a notification will force a dependency change, you can check that a change in the data does raise the event to remove the cached data from the cache.

A popular unit-testing framework is NUnit (<http://www.nunit.org/>), which is open source and available freely. You can also get UnitRun from JetBrains (<http://www.jetbrains.com/unitrun/>), which is a free tool that runs as an add-in to Visual Studio 2005 to run and debug tests created with NUnit (see Figure 6-5).

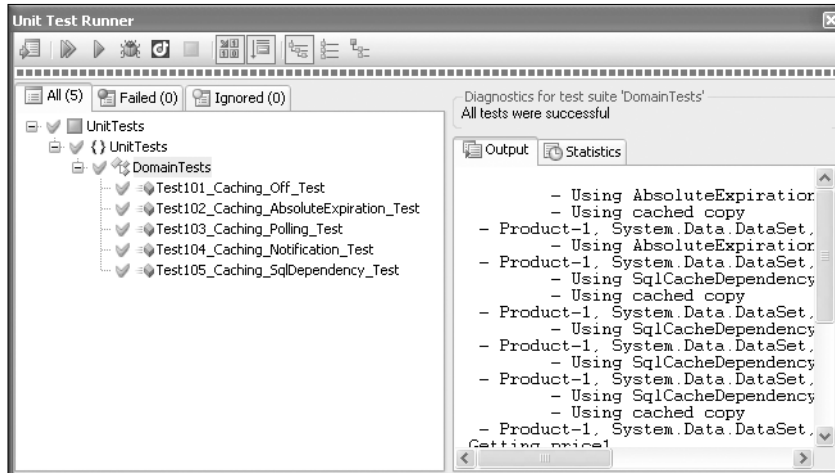


Figure 6-5. *UnitRun with NUnit tests*

Each of the tests shown in Figure 6-5 check the various caching scenarios supported by the data access layer created for use in this chapter. These scenarios include no caching (Off), absolute expiration, polling, notifications with `SqlCacheDependency`, and notifications with `SqlDependency`.

These tests are not strictly unit tests. They cross the line to integration tests because they are doing much more than a single unit of work and interact with more than a single object or resource. This distinction is critically important to some developers who feel strongly about test-driven development. For the purposes of testing here, these tests are integration tests; however, attempts are made to isolate each test so that one test does not affect another.

Part of that effort includes declaring the `StartUp` and `TearDown` methods (see Listing 6-34).

Listing 6-34. *DomainTests.cs*

```
using System;
using System.Data;
using System.Threading;
using Chapter06.ClassLibrary;
using NUnit.Framework;

namespace UnitTests
{
    [TestFixture]
    public class DomainTests
    {

```

```

#region " Shared Methods "

private Domain domain;
private int productId;
private decimal originalPrice;

[SetUp]
public void SetUp()
{
    productId = 1;
    domain = new Domain();
    DataSet productDs = domain.GetProductByID(productId, CachingMode.Off);
    originalPrice = GetPrice(productDs);
}

[TearDown]
public void TearDown()
{
    domain.SetListPrice(originalPrice, productId);
    domain.ClearCache();
    domain = null;
}

private decimal GetPrice(DataSet ds)
{
    // verify there is data
    Assert.IsTrue(ds.Tables.Count > 0, "DataSet must be populated");
    Assert.IsNotNull(ds.Tables[0], "Table must be populated");
    Assert.IsTrue(ds.Tables[0].Rows.Count > 0, "Row must be populated");

    DataRow row = ds.Tables[0].Rows[0];
    decimal price = (decimal)row["ListPrice"];
    return price;
}

#endregion

// Test Methods //

}

}

```

You can see the `SetUp` method creates the domain object and gets the original price. The `TearDown` method restores the price using the original price. For every test that is run, the `SetUp` method is run first, and after the test has completed, the `TearDown` method is run. One test does not affect the next.

With each test, a specific scenario is checked to conform to expectations. Each of the assumptions is checked with an `Assert` call, which will be counted as a pass or fail for the overall test suite. The tests in Listing 6-35 check the proper behavior when caching is not turned on.

Listing 6-35. *Test101_Caching_Off_Test Method*

```
/// <summary>
/// Tests that the system works properly with caching off
/// </summary>
[Test]
public void Test101_Caching_Off_Test()
{
    CachingMode mode = CachingMode.Off;
    domain.PrepareCachingMode(mode);

    // get the first copy of the product
    DataSet productDs1 = domain.GetProductByID(productId, mode);

    decimal oldPrice = GetPrice(productDs1);
    decimal newPrice = oldPrice + 0.01m;
    domain.SetListPrice(newPrice, productId);

    // get the second copy of the product
    DataSet productDs2 = domain.GetProductByID(productId, mode);
    decimal updatedPrice = GetPrice(productDs2);

    Assert.AreNotEqual(oldPrice, updatedPrice);
    Assert.AreEqual(newPrice, updatedPrice);

    domain.CompleteCachingMode(mode);
}
```

If caching is off, the old and new price should not match, and the updated prices should match the new price. In Listing 6-36, the test tries caching with the absolute expiration mode.

Listing 6-36. *Test102_Caching_AbsoluteExpiration_Test Method*

```
/// <summary>
/// Tests that the system works properly with absolute expiration
/// </summary>
[Test]
public void Test102_Caching_AbsoluteExpiration_Test()
{
    CachingMode mode = CachingMode.AbsoluteExpiration;
    domain.PrepareCachingMode(mode);
    domain.SetAbsoluteTimeout(3);
}
```

```

// get the first copy of the product
DataSet productDs1 = domain.GetProductByID(productId, mode);

decimal price1 = GetPrice(productDs1);
decimal newPrice1 = price1 + 0.01m;
domain.SetListPrice(newPrice1, productId);

// get the second copy of the product
DataSet productDs2 = domain.GetProductByID(productId, mode);
decimal price2 = GetPrice(productDs2);

Thread.Sleep(3000);

DataSet productDs3 = domain.GetProductByID(productId, mode);
decimal price3 = GetPrice(productDs3);

Assert.AreEqual(price1, price2,
    "price1 and price2 should match due to caching");
Assert.AreNotEqual(newPrice1, price2,
    "newPrice1 and price2 should not match due to caching");
Assert.AreEqual(newPrice1, price3,
    "newPrice1 and price3 should match once the cache expires the item");

domain.CompleteCachingMode(mode);
}

```

This test is a little more complex. The first and second price should be equal because even though the price was updated, the second price is retrieved while the old price should still be cached. With the absolute time-out set to 3 seconds, the third price should get the update value and match the update to the first price. This test specifically changes the absolute time-out to 3 seconds from the default of 120 seconds to ensure the test runs in a reasonable amount of time.

The next test checks that polling is functioning properly. The absolute time-out is still in place when the caching mode is set to polling, which is set to the default value. To speed this up, the poll time for the unit-testing project is set to 3 seconds. Listing 6-37 shows the polling test.

Listing 6-37. *Test103_Caching_Polling_Test Method*

```

/// <summary>
/// Tests that the system works properly with polling
/// </summary>
[Test]
public void Test103_Caching_Polling_Test()
{

```

```

CachingMode mode = CachingMode.Polling;
domain.PrepareCachingMode(mode);

// get the first copy of the product
DataSet productDs1 = domain.GetProductByID(productId, mode);

decimal price1 = GetPrice(productDs1);
decimal newPrice1 = price1 + 0.01m;
domain.SetListPrice(newPrice1, productId);

// get the second copy of the product
DataSet productDs2 = domain.GetProductByID(productId, mode);
decimal price2 = GetPrice(productDs2);

// poll time is set to 3 seconds
Thread.Sleep(3000);

DataSet productDs3 = domain.GetProductByID(productId, mode);
decimal price3 = GetPrice(productDs3);

Assert.AreEqual(price1, price2,
    "price1 and price2 should match due to caching");
Assert.AreNotEqual(newPrice1, price2,
    "newPrice1 and price2 should not match due to caching");
Assert.AreEqual(newPrice1, price3,
    "newPrice1 and price3 should match once the cache expires the item");

domain.CompleteCachingMode(mode);
}

```

This test runs much like the test for absolute expiration, but the messages printed to the output window show that instead of the cached item being removed due to the expiration, they are removed due to a changed dependency. Otherwise, the assertions are the same.

The test in Listing 6-38 gets into notifications. The time between updating the price and getting the updated value is very short, but long enough for the notification to remove the cached item from the cache.

Listing 6-38. *Test104_Caching_Notification_Test Method*

```

/// <summary>
/// Tests that the system works properly with notification
/// </summary>
[Test]
public void Test104_Caching_Notification_Test()
{

```

```

CachingMode mode = CachingMode.Notification;
domain.PrepareCachingMode(mode);

// get the first copy of the product
DataSet productDs1 = domain.GetProductByID(productId, mode);
Assert.IsNotNull(productDs1, "productDs1 cannot be null");

decimal price1 = GetPrice(productDs1);
decimal newPrice1 = price1 + 0.01m;
domain.SetListPrice(newPrice1, productId);

// sleep just long enough to allow the notification to work
Thread.Sleep(200);

// get the second copy of the product
DataSet productDs2 = domain.GetProductByID(productId, mode);
Assert.IsNotNull(productDs2, "productDs2 cannot be null");
decimal price2 = GetPrice(productDs2);

DataSet productDs3 = domain.GetProductByID(productId, mode);
Assert.IsNotNull(productDs3, "productDs3 cannot be null");
decimal price3 = GetPrice(productDs3);

Assert.AreNotEqual(price1, price2,
    "price1 and price2 should not match due to the cache notification");
Assert.AreEqual(newPrice1, price2,
    "newPrice1 and price2 should match due to the cache notification");
Assert.AreEqual(newPrice1, price3,
    "newPrice1 and price3 should match");

domain.CompleteCachingMode(mode);
}

```

The assertions verify that the first and second price do not match, while the new price does match the third price, which should come from the cache. The text in the output window should again show when the dependency change removes the item from the cache and when a cached copy is used.

Finally, the last test shown in Listing 6-39 uses the `SqlDependency` object, which does not use the cache, but still uses the notifications system.

Listing 6-39. *Test105_Caching_SqlDependency_Test Method*

```

/// <summary>
/// Tests that the system works properly with SqlDependency
/// </summary>
[Test]

```

```
public void Test105_Caching_SqlDependency_Test()
{
    CachingMode mode = CachingMode.SqlDependency;
    domain.PrepareCachingMode(mode);

    // get the first copy of the product
    Console.WriteLine("Getting price1");
    DataSet productDs1 = domain.GetProductByID(productId, mode);
    Assert.IsNotNull(productDs1, "productDs1 cannot be null");

    decimal price1 = GetPrice(productDs1);
    decimal newPrice1 = price1 + 0.01m;
    domain.SetListPrice(newPrice1, productId);

    // sleep just long enough to allow the notification to work
    Thread.Sleep(200);

    // get the second copy of the product
    Console.WriteLine("Getting price2");
    DataSet productDs2 = domain.GetProductByID(productId, mode);
    Assert.IsNotNull(productDs2, "productDs2 cannot be null");
    decimal price2 = GetPrice(productDs2);

    Console.WriteLine("Getting price3");
    DataSet productDs3 = domain.GetProductByID(productId, mode);
    Assert.IsNotNull(productDs3, "productDs3 cannot be null");
    decimal price3 = GetPrice(productDs3);

    Assert.AreNotEqual(price1, price2,
        "price1 and price2 should not match due to the SqlDependency");
    Assert.AreEqual(newPrice1, price2,
        "newPrice1 and price2 should match due to the SqlDependency");
    Assert.AreEqual(newPrice1, price3,
        "newPrice1 and price3 should match");

    domain.CompleteCachingMode(mode);
}
```

This final test works much like the other notifications test, except the internals use a custom collection to hold onto a copy of the product data instead of relying on the cache. When all of these tests pass without an error, it confirms that the caching system and the database are working properly. This small group of tests will at least provide a baseline of support that is necessary for the application as a whole. Some teams make it a requirement that all unit tests must pass before changes are committed to the source control system. It acts as a safeguard from breaking changes.

Once a testing system is in place, you can use an MSBuild script to run the tests without firing up Visual Studio 2005 (see Figure 6-6). If your current work relies on several other projects, you can sync up the source code for the dependency projects and run the MSBuild scripts to build and test them. This automation will save you time and identify problems with the dependencies before you start making changes in your project.

```

C:\WINDOWS\system32\cmd.exe
SqlDependency.OnChange:
  Type: Change
  Source: Data
  Info: Update
  Id: 82d276a2-2910-4151-8499-11bb3caa6de6;5b7e03ab-f14a-4a75-be12-0a63336e48
af
  - Removed data item <1>
  Getting price2
    - Using SqlDependency
  Getting price3
    - Using cached copy

Tests run: 5, Failures: 0, Not run: 0, Time: 13.680 seconds

Tests run successfully!

Build succeeded.
0 Warning(s)
0 Error(s)

Time Elapsed 00:00:15.89
D:\Projects\Apress\Book1\Chapter 05>pause
Press any key to continue . . .

```

Figure 6-6. MSBuild with unit testing

Problems with Caching

Unexpected problems come up when you make use of caching, unless you have devised a caching policy that you enforce through planning, training, and code reviews. You may decide as a policy that all data placed in the cache will use an absolute expiration no greater than five minutes. You may also decide that you will not make use of output caching and instead focus on data caching with dependency notifications to ensure the data used by the application is always current. By setting a clear policy and enforcing it with code reviews, it will be easier to keep things under control.

There are a few things to keep an eye on. When using data caching, you could easily hold cached items with a static reference in a code-behind that will cause it to never go away. I find it is best to avoid static references when working with cached data. To cope with it, I create wrapper methods that act as the access points where I get the data. The wrapper first checks for the cached item in the cache and returns it if it exists. If not, it pulls the data from the database, places it in the cache, and returns it. I never assume that the item is already in the cache.

With output caching, you will likely set the `VaryByParam` to `*`, thinking that will ensure that different URLs will get different results. For a product page, you want `Product.aspx?ProductID=808` to show a different result from `Product.aspx?ProductID=429`. This `VaryByParam` setting will make sure it works that way, but if you have a website that has been set to show content in multiple languages, the first user to hit the page will cache the content for his language. The next person who hits the page, with a different language preference, will see the language preference of the first user for the duration that the output cache holds the original request. To account for this situation, you can check the `Accept-Language` header as a part of uniquely identifying the contents of a page. Listing 6-40 shows the settings to use for output caching when language is a concern.

Listing 6-40. *VaryByHeader for Language*

```
<%@ OutputCache Duration="300" VaryByParam="*"
    VaryByHeader="Accept-Language" %>
```

Performance Strategies

It will be necessary to develop a comprehensive approach to performance. Simply caching the results from the database will not ensure your application performs well. If several of the queries are very slow, you will continue to experience performance problems. Wherever your application is slowest, you can try to speed up the queries.

In one scenario, you may have a query that is run occasionally, but it takes a painfully long time to return the results. In another scenario, you have a moderately slow query, but it needs to run multiple times with different parameters, which quickly adds up and slows down your application. In both cases, you have the choice of optimizing the query itself or changing the structure of the data that it is querying.

A query may run a long time because it has the join across multiple tables, even if there are indexes in place to assist with the table joins. If there is a great deal of data, you may be scanning across a great many rows to match your join criteria. This performance hit is hard to avoid by rewriting the query, but it is possible to instead restructure the data.

Data Warehousing

Database designers will do their best to break up data so that it is fully normalized. Doing so reduces the required storage to hold your data, but at the same time it requires your queries to be more complicated. Consider a product database again. If you have 50,000 products and all of the data related to the products is spread across ten tables, you will need to join across many tables to get to the data you need. And some products may not even be visible for purchase by customers, so you have extra data in the database, which adds to the cost of a full table scan.

In this scenario, you may be able to create a denormalized table to hold your product data so that it is not necessary to join tables. When the denormalized table is populated, it can also be limited to the active products to reduce the size. A simple `SELECT * FROM MYTABLE` is a great deal faster than any join.

To make this work, you still need to refresh the warehouse table occasionally. Perhaps you can do it during off-peak time periods a few times a day, or just do it once a day during the slowest usage period. You may even choose to spread the databases across separate physical machines and populate the warehouse with a remote query. Doing so will allow your vendors to work on the data for their products throughout the day without impacting the performance of the publicly used database.

I did all of this with a product database and pulled all of the contents of the warehouse table into a `DataSet` that I then dropped into the cache. I considered holding all of the products in memory when I discovered that the data only took up less than 10 MB of space while the server had well over 2 GB of total memory. Once I had the `DataSet` in memory, all I had to do was query it with the `RowFilter` property on the `DataView` object (see Listing 6-41).

Listing 6-41. Using RowFilter with ProductID

```
public DataTable GetProductByID(DataSet productDs, int productId)
{
    DataView dv = new DataView(productDs.Tables[0]);
    dv.RowFilter = "ProductID = " + productId;
    return dv.ToTable();
}
```

This sample takes a `DataSet` that already has all of the products and then makes use of a `DataView` with a `RowFilter` to create a new `DataTable` with just the matching rows. You can then use this `DataTable` with any databound control just as you would with a `DataSet`.

With .NET 2.0, the performance of `DataSets` improved dramatically with a revised indexing implementation. As a result, this technique was run very quickly. The `RowFilter` property can take any string that you would use in a normal `WHERE` clause in a SQL query.

What I also discovered was that loading every row from the warehouse table was faster than loading a single product using a query with joins with the normalized tables. If I had to keep the product data more current, it may have been reasonable to allow the `DataSet` to fall out of the cache more frequently without compromising the performance significantly.

The products also fit into a hierarchy of categories. A set of root categories held subcategories and so on until they reached products. I flattened this structure and placed it into another warehouse table for fast access. In the case of this table, I included many calculated fields such as the number of products and the start and end dates for the categories, as some categories were seasonal. These extra columns assisted with the `RowFilter` by making them easy to access without querying the database. I found that each time I had a new requirement to query the database for new criteria, I could simply add a column to the warehouse table instead of breaking out of the warehouse strategy, which was working extremely well.

Lazy Loading

Not every product detail was placed in the warehouse table. When there were one-to-many relationships, it was not possible to do so. I also did not want to load data that was not immediately necessary. As users dig into the website through the hierarchy of product categories, they only see summary data about each product such as the name and price. Loading the detail data can be deferred until it is necessary, and I did so by leveraging the object model.

What I typically do is translate the data from the database in `DataSet` form into a business object such as a `Product` that has properties such as `Name`, `Price`, and `VendorName`. These are the sort of properties that are always loaded when the object is constructed. I call the data that goes into these properties *summary data*. Other properties such as `Colors`, `Features`, and `Dimensions` are detail data that is not shown on category pages and not loaded into the `Product` object when it is first constructed. But when the property is accessed, starting with the null value, the database is then queried for the necessary data and returned. This is called *lazy loading*, and Listing 6-42 shows an example of this.

Listing 6-42. *Lazy Loading Example*

```
private ColorsCollection _colors = null;
public ColorsCollection Colors
{
    get
    {
        if (_colors == null)
        {
            _colors = GetColors(ProductID);
        }
        return _colors;
    }
}
```

Every subsequent request for the `Colors` property will have the data already and will return immediately. This all works on the assumption that the summary data will be used far more often than the detail data that is pulled in via lazy loading. In the case of a website listing summary data on the category pages, this will be true.

The individual `Product` object can be placed in the cache for a limited time, which will automatically hold onto the detail data held by the member variables. This detail reduces the number of hits on the database for the entire product. However, this may not be exactly what you want. In the same way the output cache will cache data for the entire page regardless of any individual data caching time-out, this will also hold onto the data as long as the parent object is held in the cache. Instead of holding onto the data as a member variable, it could use data caching and simply request the data each time and expect that the data will be cached for the appropriate amount of time. You have many available options with this scenario.

Summary

In this chapter, you saw the various options of caching with `System.Web.Caching` and other alternatives. You also learned the techniques used to invalidate the cache due to changed dependencies as well as several problems with caching. Finally, you explored a few techniques and strategies to make the most out of mixing and matching the available features to squeeze the best possible performance out of the system.



Manual Data Access Layer

Although there are many powerful tools built into Visual Studio to assist with binding data to your application, you will still manually build portions of your data access layer. A manual data access layer will give you the greatest flexibility in laying out the table structure, the stored procedures, and the layer of code to carry the data from the database to your application and back. As you work directly with your data access layer, you will become intimately aware of all the nuances of your system. You will also be able to put a good deal of thought into how the system will behave.

This chapter covers the following:

- Using DataSets, inline SQL, and stored procedures
- Using DataObjects and the ObjectDataSource
- Building the database
- Building the data access layer
- Building the website

When building a data access layer, you can make design decisions that affect performance. If you are fortunate, you can make changes to the way data is structured in the database, and even better, you could be starting from scratch. Your design decisions will have an impact for better or worse. Currently, our options are expanding in new ways with the introduction of WCF and LINQ to the .NET platform. These options push beyond the typical ADO.NET and DataSet model, which has become commonplace. Yet these new technologies are compelling enough to break into the designs reviewed in this chapter.

The sample application for this chapter is a website used to store and tag your favorite links. I will show you how every part of the application is built and explain the decisions made along the way.

Using DataSets, Inline SQL, and Stored Procedures

The traditional approach has been to use either inline SQL or stored procedures to fill a DataSet with query results and to bind that data to a databound control on the user interface. This process can be used to quickly throw together a rich interface that is actually quite functional.

However, too often the only cost that is considered is the time to build the first version of the application. But then the next version comes around. As the real world changes, the requirements for the application and the database also change. Going back and maintaining an application has historically shown to take up 80 percent of developers' workload.

You can open the Server Explorer in Visual Studio, drop a table onto the design surface of a Web Form, and automatically generate the GridView and associated SqlDataSource with full support for selecting, updating, deleting, paging, and sorting. But what happens when even the smallest change is required? What if you add a new column to the database table and want to show it on the GridView? Are you certain that you will properly update all the commands in the SqlDataSource? Now what happens if your query joins two or more tables? How do you handle the delete and update commands? This is where all the wizard tools leave you to your own devices and where you start to get your hands dirty and spend more time with a whiteboard and dry-erase marker. And if you have gotten to this point with the easy-to-use wizards, it will be difficult to form a functional plan to address the new feature requirements.

DataSets

The DataSet is an object we have come to know well. It is the primary object used to push data around in .NET. It comes in two major flavors: Typed (XSD) and Nontyped. You looked at their differences in Chapter 2, which exposed the rigid nature of a Typed DataSet that makes it a difficult animal to maintain as the database changes. So we fall back to the Nontyped DataSet. In practice, a DataSet is used as a glorified data transfer object (DTO), which is magically bound to a control without any real regard for the data type for each column. This automatic data binding is what allows us to build fast prototype applications by using the wizards. But this nontyped grouping of data with all of the automatically generated methods can confuse the developer, who must use it as a data access layer. We will get into these difficulties in the following sections.

Compile-Time and Runtime Support

The content of a Nontyped DataSet is defined from the query that populates it. If a column called BirthDate is a DateTime today but it updates to return a VARCHAR(15) tomorrow, the Nontyped DataSet will hold the changed column automatically without complaint. But if you are casting that value to a DateTime in your code, you will risk an InvalidCastException, which will happen at runtime. It is preferable to catch this at compile time, so you can take care of it while you are making changes to the application. Of course, you will be testing your application for the proper runtime behavior, but because the database is really disconnected from the application, the type coming back from a query could change at the worst time. Another database administrator who is not aware that your application is querying a table may adjust it to change an int column to bigint because the auto-incrementing values are about to reach the maximum value for 16-bit integers. Such changes are unfortunate, but they do happen.

Consider a table that has a primary key value set as `int`, which is also an `IDENTITY` column incremented by 1 each time a new record is created. You may be casting the column as an `Int16`, but once it goes beyond 32,767, you will get a runtime exception because it will now need to be at least an `Int32`. If you are just testing the application with a few records in the database, you will not reach this runtime exception. And when this problem does finally happen, it will be when the application has been out in the wild for a while like a time bomb.

And when this runtime error happens, how will it be logged and reported back to the development team? If the error happens on the user interface layer, the error will seem to indicate a binding problem instead of the real cause for the problem deep in the database. You can imagine that diagnosing this sort of problem will chew up your development time as you spin your wheels looking for the problem on your side of the system, which is using a different copy of the database that has not reached that troublesome threshold.

Instead of using a `Nontyped DataSet` that happily passes along any data dropped into it, you can manually load data into a business object with properties set to the exact type you intend the user interface layer to use. When the application is compiled, it will ensure that these types line up properly. When a change happens in the database that breaks this implicit agreement, the error can be handled and logged at the source of the problem so you are better able to fix it. A logged error message that states, "Type mismatch at Line 151 while loading Person object" will pinpoint the issue so that you can check that line of code for what it is expecting and which type the database query is actually returning.

Refactoring

A `nontyped` value also makes it hard to refactor the application. You cannot simply right-click on a property in a `DataSet` and select `Find All References`. Values from a `DataSet` are accessed as a `DataRow` or `DataRowView` by using an index by name or number, which cannot be tracked back to the source object through automatic code analysis. A generic `DataRow` could belong to any `DataSet`, and the refactoring features provided by Visual Studio and third-party add-ins will not assist you in making solution-wide changes. To make changes globally, you must resort to a search-and-replace, but doing so is problematic.

When your application is revised and you want to update a query to make it more descriptive, you may add another column called `VendorName`, rename `Name` to `CustomerName`, and eliminate the `Name` column. Suddenly all references to `Name` are now invalid and will cause runtime exceptions, so you must find all references to `Name` and correct them. Each time a change like this is made, you must run your searches and be sure that `row["Name"]` actually refers to the same query result you have just changed. You could easily have many queries with a column named `Name`. A global search-and-replace would break those references.

If you instead are using custom business objects, you can safely use refactoring tools to rename properties for the scope of a solution. To reach references beyond the scope of the application, you can also mark an older property with the `Obsolete` attribute and a note to use the new property while maintaining existing functionality. This technique is covered later in this chapter.

Debugging

You might expect that debugging with DataSets would be well supported. Surprisingly, the lack of debugging support is a significant shortcoming of working with DataSets. In Figure 7-1 you can see how you can get to the values for the first DataRow of the first DataTable in the DataSet. To get to the next DataRow, you must change the index value.

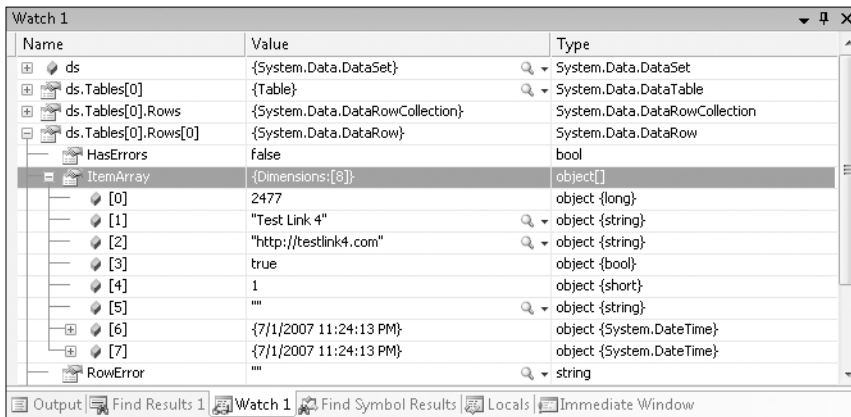


Figure 7-1. Viewing a DataSet in the debugger

At least you can view the values for the columns in the row, but these values are accessed by using the index and not the column name. When you are accessing this same row that is being attached to a databound control on the Web Form, such as a Repeater, you will access the row with `e.Item.DataItem`, which is a `DataRowView`. And as you see from Figure 7-2, the watch window is less helpful than Figure 7-1. You cannot directly navigate to the values, and you can hardly infer the type by looking at the watch window.

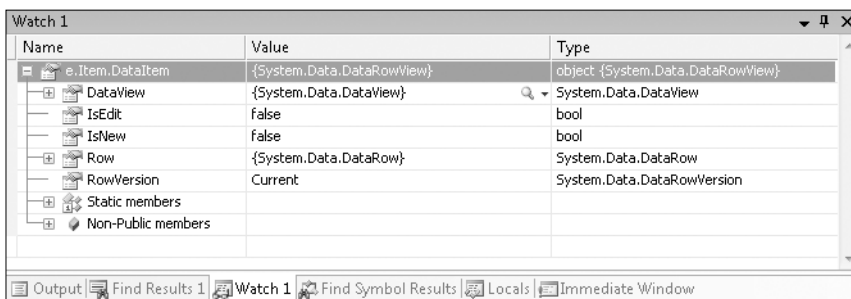


Figure 7-2. Viewing a DataRowView in the debugger

This poor debugging experience will decrease the amount of time you can spend creating and maintaining your application. When the data is moved immediately into a business object on the data access layer and passed back to the application, you will be able to access the data from the debugger in a much more useful way. The same data used in Figures 7-1 and 7-2 is

dropped into a `FavoriteLink` object in Figure 7-3. `FavoriteLink` is a key object used by the website that we will be building later in this chapter.

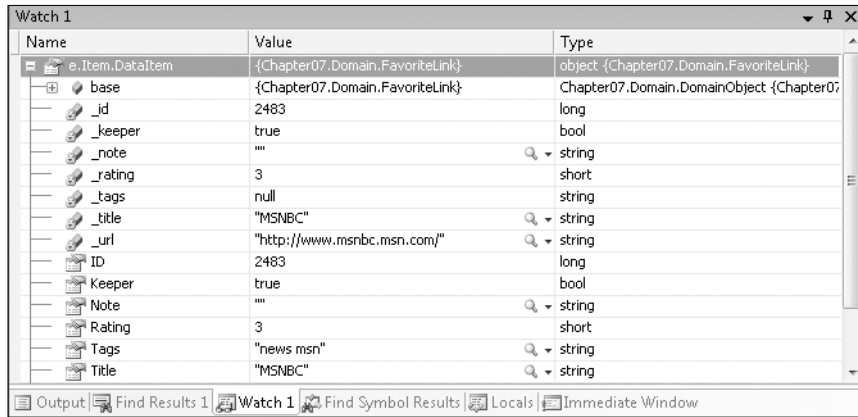


Figure 7-3. Viewing a `FavoriteLink` object in the debugger

The values shown in Figure 7-3 are definitely much more readable, and you know exactly what you are working with. Using a business object clearly offers a better debugging experience over `DataSets`. Compile-time support is restored for at least the user interface layer, while the type-safety issues are isolated deep inside the data access layer. When you set the `Url` property in the `FavoriteLink` object by casting `row["Url"]` to `String`, you will do so near the query that pulled the data. Any resulting exception will show a stack trace leading to this query and the line number if you have deployed the Program Debug Database (PDB) file with the assembly. Later you will see how to completely avoid the `InvalidCastException`.

Note Exceptions can provide a more detailed stack trace if the PDB file that is created when the class library is compiled is deployed with the assembly. Useful details such as the exact line number where the exception came from will assist you in tracking down and fixing bugs. The PDB file is created while in the Debug and Release configuration.

Like any other normal object, the `FavoriteLink` object is also more approachable for refactoring. You can find all references to the `Url` property wherever it is used in the code in your solution. Unfortunately, if you declaratively bind the `Url` property to a databound control, the refactoring search will not find that reference.

And finally, you can use the `FavoriteLink` object more easily while debugging, as shown in Figure 7-3. This even includes setting break points in the property accessors, which will be very useful when watching which values are being bound to a databound control.

Inline SQL

One temptation to avoid is writing your SQL right into your code. Although it is true that the code you create to communicate with the database will be tightly bound to the queries and other database commands, that is not the most efficient way to create or maintain the software.

Maintenance Considerations

One problem with placing SQL right into your code is the resulting extra maintenance work when those queries change. You end up making more work for yourself and increasing the opportunity for error. For example, your most basic query may start out looking like Listing 7-1.

Listing 7-1. Simple Inline Query

```
String sql = "SELECT * FROM Production.Product WHERE ProductID = 1"
```

You can see that it starts out all right with a readable query, but as the query gets longer, it starts to look more like Listing 7-2.

Listing 7-2. More Complex Inline Query

```
String sql = "SELECT " +  
    "p.ProductID, p.[Name], p.ProductNumber," +  
    "p.Color, p.ListPrice, " +  
    "p.SellStartDate, p.SellEndDate, " +  
    "p.DiscontinuedDate " +  
    "FROM Production.Product AS p " +  
    "JOIN Production.ProductProductPhoto AS ppp " +  
    "ON ppp.ProductID = p.ProductID " +  
    "WHERE ppp.ProductPhotoID != 1";
```

The query in Listing 7-2 is an obvious maintenance nightmare. As you make minor changes, you will do it inline and do your best to make sure you have the query wrapped carefully inside the quotes for the String. But there is so much that you give up.

If the query in Listing 7-2 was maintained as a stand-alone SQL script, you could use tools such as SQL Server Management Studio or Visual Studio with Database Projects to provide syntax coloring along with several other rich features. Third-party tools are also available to offer real-time IntelliSense support and give you access to listings of column and table names as you work with your SQL. An inline query cannot leverage all of that support.

Security Considerations

Another problem with using inline SQL is the risk of SQL injection attacks. This risk is often not considered but it is a very real problem. In Listing 7-1, the ProductID is hard-coded into the String, but you may be getting the value from a Query String with a URL such as `Product.aspx?ProductID=1`. This inline query can be easily exploited if the String to query the database looks like Listing 7-3.

Listing 7-3. *Exploitable Inline Query*

```
public String GetProductQuery(String productId)
{
    return "SELECT * FROM Production.Product WHERE ProductID = " + productId;
}
```

An attacker will need to only adjust the query string with a statement that ends the query and starts another command, such as an update or a delete command. A resourceful attacker could gain access to all your data by querying your schema to discover all the tables and copying the data without you ever knowing. And until the problem is corrected, the attacker can return anytime to repeat that query and compromise your data.

Protecting against a SQL injection attack starts by never trusting data coming from the user. Data should always be validated. If you expect the `ProductID` to be an integer, you could use the code in Listing 7-4 to more safely assemble the query.

Listing 7-4. *Validating Incoming Data*

```
int productId = 0;
String sql = "";
if (int.TryParse(Request.QueryString["ProductID"], out productId))
{
    sql = GetProductQuery(productId);
}
else
{
    // handle the failed attempt
}
```

To make matters worse, if your application shows the message from the exception directly to a potential attacker, it will give that attacker information that can help in modifying the query. When you catch an exception, you should log the error and throw a new exception without any information that would help an attacker. A user does not need to know that level of detail. The user does not need to notify you about the details of the exception, because you can always look into the error log for the details. You also should monitor that error log because it may show attempts to compromise your database.

The next step beyond validating all incoming data is to use parameterized queries to bind values to database commands. When values are specifically bound to placeholders in a query with type restrictions, not to mention the proper ordering of parameters, it makes it much more difficult for an attacker to assemble a functional attack. A parameterized query will also escape out characters such as quotes and wildcard characters, which eliminates the possibility of injecting a string that will close your query and start another. Listing 7-5 shows a parameterized query.

Listing 7-5. Parameter Binding

```

public DataSet GetProduct(int productId)
{
    DataSet ds;
    try
    {
        String sql = "SELECT * FROM Production.Product " +
            "WHERE ProductID = @ProductID";
        using (DbCommand dbCmd = db.GetSqlStringCommand(sql))
        {
            db.AddInParameter(dbCmd, "@ProductID",
                DbType.Int32, productId);
            ds = db.ExecuteDataSet(dbCmd);
        }
    }
    catch (Exception ex)
    {
        // handle exception
        throw;
    }
    return ds;
}

```

When ProductID is validated at the user interface layer, you can take proper action such as showing a validation message that is meaningful to the user. Doing so will prevent garbage data from getting to the database.

Another step can be taken to further protect the data, but it cannot be done with inline SQL. When all interactions with the database are done through stored procedures instead of inline SQL, you can lock down access to the tables and specifically allow access to stored procedures for particular users.

Stored Procedures

Stored procedures can be used to extend your application and provide a clean and flexible bridge between your user interface and the underlying data. By blocking access to the tables, you make the stored procedures the only entry point to the database. When all data passes through a stored procedure layer, which can include programming logic, you have a great deal of flexibility.

Stored procedures working as your public interface become an integral part of your data access layer. In fact, if you have five applications using your database and want to log each time a record is deleted from the Production.Product table, the stored procedure used to delete those records can be updated to handle the logging. It is not necessary to incorporate that change in each of the five applications. And if you choose to adjust the structure of your database, perhaps further normalizing the structure, the stored procedures can still use the same parameters and return the same output after the structural change in many cases.

You can also encapsulate what some developers may consider to be business logic in stored procedures. Generally, it is not a good idea to bury business logic in a stored procedure.

A stored procedure, typically created in T-SQL, is not as expressive and testable as C# code for the average developer who tries to avoid writing anything in T-SQL. Those developers who are comfortable with T-SQL may disagree about the viability of placing business logic into stored procedures.

Sometimes the business logic is already tightly bound to the structure of the database. For example, the logic to determine whether a product is on sale, and for what price, may be considered business logic. But joining the product table to the sales schedule and pricing tables is structural logic. If I run a stored procedure named `GetProduct` during a sale, it would give me the discounted price. Running it at any other time would give me the regular price. Encapsulating that sort of logic in the stored procedure is compelling because of the advantages it offers. It hides the structure of the database from your application and gives you the data you need. And if the database was first created without a sales feature, the stored procedure gives you the opportunity to roll out such a feature without also updating your application code. The responsibilities we assign to T-SQL vs. the rest of your application will be a line we cross from time to time.

Because you can identify which stored procedures are in place, you can identify exactly which commands are hitting the database. You also have direct access to all these commands. If you decided to restructure the tables for any reason, you can put together a set of scripts to adjust the tables and update the stored procedures to carry out the change. If you had queries spread across five applications, you would have to look into each application to see how each of the queries was working in the source code of each application. If a query is being assembled in a complex way, you may have a difficult time confirming that your updates will not cause breaking changes. Going forward, we will leverage all the advantages that stored procedures offer.

Using DataObjects and the ObjectDataSource

We come back to `DataObjects` and the `ObjectDataSource`, which were covered briefly in Chapter 2 as a part of our data model choices. From all those choices, these two deserve the most attention because of the flexibility they provide. These features were designed to work seamlessly together with rich tool support to give the developer a great deal of control over the data layer. These pieces give us an intelligent way to plug the application into the datasource. After the application is plugged into the datasource, the data flows over a pipeline in whatever form we choose.

As the developer creating the data access layer, you will create classes marked with a `DataObject` attribute with methods marked with `DataObjectMethodType` attributes, which indicates what the method will provide for an `ObjectDataSource`. Table 7-1 shows each of the values of a `DataObjectMethodType`.

Table 7-1. *DataObjectMethodType Enumeration*

Insert	Represents a method that performs an insert operation
Update	Represents a method that performs an update operation
Delete	Represents a method that performs a delete operation
Select	Represents a method that retrieves data
Fill	Represents a method that fills a <code>DataSet</code>

When the developer using the data access layer drags an `ObjectDataSource` onto the design surface, a wizard panel will point the developer at these methods by first allowing him to select any class marked as a `DataObject` and then methods with the `DataObjectMethodType` attribute for the selected class. It shows the developer the return type and the parameters, along with the method name, which should be explicit enough to indicate what it is doing (see Figure 7-4).

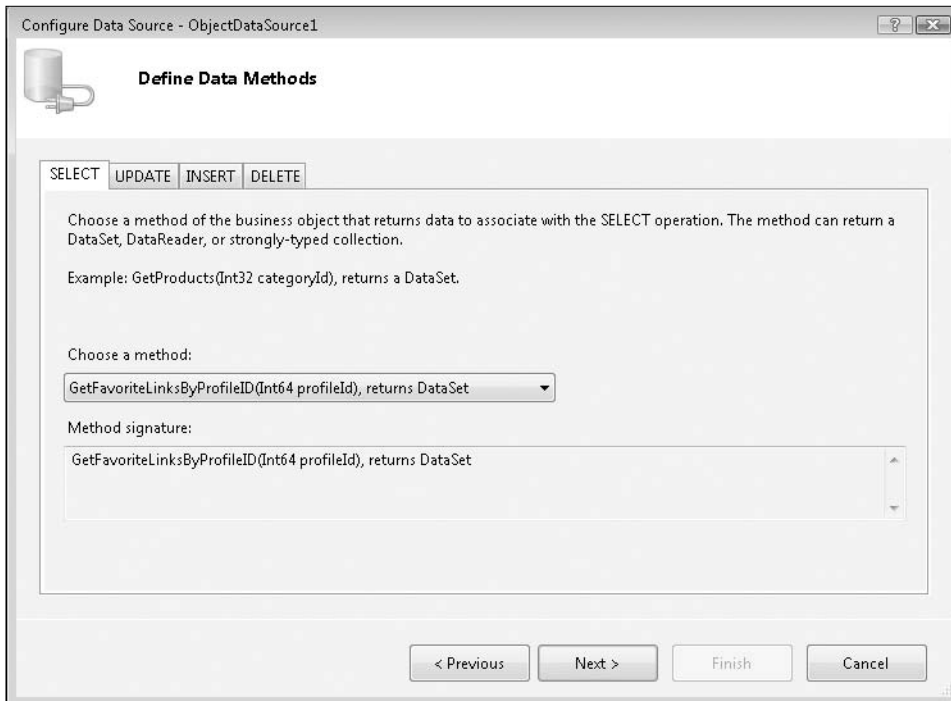


Figure 7-4. *ObjectDataSource* wizard method selection

By directing developers to the right objects and methods, you make their jobs much easier while encouraging them to use the right methods. Digging through the source code and guessing at which classes and methods to use would take more time and potentially introduce problems if they make the wrong choice. Even worse, they may give up and bypass the data access layer and directly access the database. By encouraging the development team to use the `ObjectDataSource` and the properly built data access layer, they will become accustomed to building the application by using the design surface in this way and expect the database developers to provide them with the classes and methods they need to implement features for the application. With this relationship established, and as the application developers begin to rely on the database developers, you can start to formalize communication.

Design Contract

When the application developer is displaying a set of data about products, the developer will use a databound control such as a `GridView`. And when the developer uses an `ObjectDataSource` to

bind the data to that GridView by using the data access layer you created, the developer will select a class and method that suit his needs. When selecting an object named `ProductDomain`, which has a method named `GetAllProducts` and which returns `ProductCollection`, the developer will assume that is the right choice. But these names should do more than just indicate some relation to products. The names and objects represent a Design Contract that the data access layer is maintaining for use by any application that will use it. How these objects and methods go about gathering the data and returning it is not a detail that is a part of the Design Contract, but what methods are available and what they return is important. A violation of the contract will cause runtime exceptions.

Another method named `GetProductsByCategory` that also returns `ProductCollection` should include the same columns that the `GetAllProducts` methods return. The methods on the `ProductDomain` class could be set as public or private. Making them private specifically excludes them from the Design Contract. And for the public methods, you can mark them with a `DataObjectMethodType`, which defines the intended use of the method. After the developer has used a method that returns `ProductCollection`, she should be able to use the other methods on the `ProductDomain` class without concern about the data being different than expected.

As the application developers require functionality from the data access layer, the Design Contract will be negotiated to include what is needed and when it will be provided. And after the new features have been published, the contract cannot be changed without giving the application developers proper notice of the change.

Data Contract

After a developer has plugged his application into your data layer with the implied Design Contract, there is also another agreement in place. The Design Contract defines the pipeline that the data flows through, but the messages flying around must be defined with a Data Contract. If `ProductCollection` holds onto `Product` objects that have properties named `ProductName`, `ProductNumber`, and `Price`, that Data Contract must be maintained. After the data binding is in place, it can be difficult, if not impossible, to change that name later without breaking the applications using it.

The Data Contract includes the name of the fields as well as their type. If the `ProductName` is a `String` and the `Price` is a `Decimal`, it should always be that way for future versions of the data access layer. The data bindings will be declared with those names, expecting the types that were defined when the bindings were first declared. Changing the Data Contract will require the applications to be updated to prevent runtime exceptions.

Of course, there will be times when the Data Contract has to change, just as the data itself also changes. When doing so, you should first increment the version number up to the next significant value. You should then produce documentation explaining any breaking changes and how the developers should adapt their applications to the new Data Contract. You can also help them out with a transitional period, when the old Data Contract is maintained with a defined end date.

In a simple scenario, you start with a `Person` object that has a `Name` property representing a person's full name. Later, because of a change in requirements, the `Name` is broken into properties called `FirstName`, `MiddleInitial`, and `LastName`. You can start a transitional period by adding the new properties while keeping the old property in place. Listing 7-6 shows a method marked as `Obsolete`.

Listing 7-6. *Breaking Up the Name Property*

```

/// <summary>
/// Expects a name formatted like "John Q Public" or "John Public"
/// </summary>
[Obsolete("Use FirstName, MiddleInitial and LastName")]
public string Name
{
    get
    {
        if (String.IsNullOrEmpty(MiddleInitial))
        {
            return FirstName + " " + LastName;
        }
        return FirstName + " " + MiddleInitial + " " + LastName;
    }
    set
    {
        string[] parts = value.Split(" ".ToCharArray()[0]);
        FirstName = parts[0];
        if (parts.Length > 2)
        {
            MiddleInitial = parts[1];
            LastName = parts[2];
        }
        else if (parts.Length > 1)
        {
            MiddleInitial = String.Empty;
            LastName = parts[1];
        }
    }
}

```

In Listing 7-6, the `Name` property wraps the newly introduced properties to maintain the existing Data Contract while giving the developers the option to start transitioning their applications. To assist the developers, the `Obsolete` attribute has been placed on the `Name` property to display a warning when their code is compiled. The warning includes a useful message explaining which properties should now be used.

Perhaps your data access layer starts out at version 1.2.0.8. When you apply these transitional changes, you set the version to 1.2.1.0 and release it to the developers. After the transitional period ends, you can bump the version more significantly to a value such as 1.3.0.0 or even 2.0.0.0, which should prompt the developers to recognize that a significant change has been made. If the developers see only a very minor version number increase, they may assume that it is only a minor bug fix release and start using it without properly evaluating and considering the changes.

Testing the Design and Data Contracts

Because you cannot fully leverage compile-time support to ensure that runtime exceptions are avoided, you must test your application. Naturally, this is a good place to add automated tests. These tests would confirm that the data retrieved from the datasource includes the expected fields with the proper types.

These tests can be done at two separate levels. Directly on top of the data access layer, a set of unit tests can ensure that the expected data is returned; a test database is populated with specific data that the data access layer and the tests use to return a consistent result. After the data access layer passes those tests, the next step is to include integration tests that account for the data being used through data binding.

Testing code is fairly easy. Testing data binding on a web page is not so easy. This step now requires loading up each page and checking that each value looks right and does not cause an exception. Tools such as Microsoft Visual Studio Team System offer tools to create and run such tests. Alternatively, you can use Selenium from OpenQA, which offers cross-browser and cross-platform testing functionality for web pages.

In these integration tests, you load up the page and verify that the expected values appear on the page.

Building the Database

Whenever I build an application, I always start with the database. It is a bottom-up approach. I know what data I will be managing and what I will need to do with the data, so I first plan what the database will hold, how relationships among the data will work, and how the application will interact with it. By keeping the scope of the work limited to just the database, the work progresses more quickly without any distractions about the interfaces or other ancillary details. After the database functionality is completed, building the front end of the application is much easier.

In this chapter, you will build a social bookmarking website that holds onto links and features a tagging system for browsing the links. The profiles will be directly integrated with the standard ASP.NET Membership User accounts so that you can leverage all the other features available to ASP.NET for managing user accounts.

Creating the Database Structure

The data structure will hold onto the links and tags and the association to the Membership User. It will be broken up in a fully normalized way to reduce the storage requirements as well as optimize the speed of access to the data. My assumption is that many users will use the application and that different users will save the same links and use the same tags, so I will want to store that common data only once and hold onto just the relationships to that data.

Figure 7-5 lists the database structure. It shows the central piece is the `FavoriteLinks` table with all the related data around it.

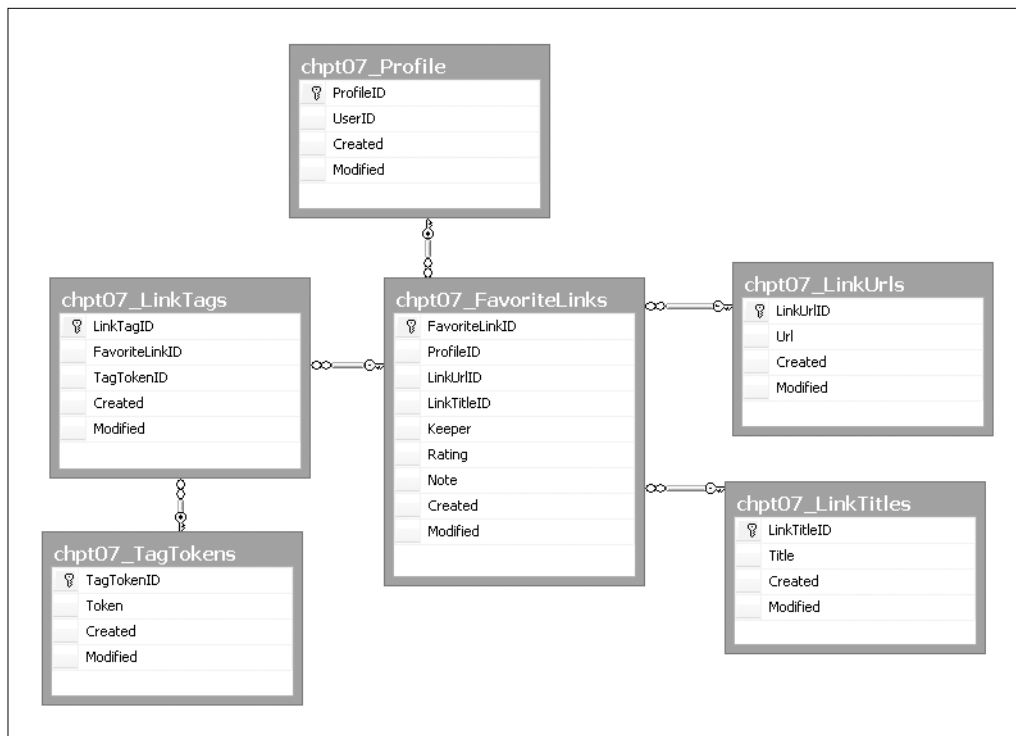


Figure 7-5. Database structure

Consolidating the Data

Because many users will store the same URL and title as a part of their links, that data is held in tables separate from the `FavoriteLinks` table. When a user stores a link for `http://youtube.com/`, the save procedure will create a new record in the `LinkUrls` table. The next user to save the same link will simply point to the existing record. The same is true for the `LinkTitles` table for any titles used with the links. This way, the least amount of data possible will be stored in the database.

Reducing the duplication of data also cuts down on the cost of a full table scan as well as a range scan of the tables. In the stored procedures to follow, efforts are made to ensure that the data is pruned of any unused data and that the database is as small as possible. When a link or a tag is no longer used by any user, it will be removed from the database.

Managing Relationships

Data does not live in a vacuum. There will likely be many relationships between the tables in the average database. The kinds of relationships can be broken down into three groupings:

- One-to-one
- One-to-many
- Many-to-many

A one-to-one relationship may not be terribly useful at first glance, but consider how you could break up the data. You could hold information about the most accessed data about a user in one table and hold the extra details in another. I call the first table the summary table and the second one the detail table. And I may have multiple detail tables. In the FavoriteLinks database, the Membership User record has a relationship with a single Favorite Link profile. This was not an effort to normalize the data but to keep the data grouped by use. A table with 30 columns will be difficult to query in terms of the hard drives scanning each record, not to mention reading and maintaining the indexing. By breaking up the data, you can enhance performance. When it is broken up, you also have the opportunity to consider partitioning the data for further performance gains.

You will want to carefully group your data among the tables to create the least number of joins possible across all the queries necessary for your application. Ideally, you would be accessing the detail tables directly with their primary key after you have accessed the related record from the detail table.

The next kind of relationship is one-to-many. For the FavoriteLinks database, there are several one-to-many relationships. A record in the Profile table can have many references to the FavoriteLinks table. The child record points back to the parent in each instance to allow for multiple relationships. These child-to-parent references are done with foreign keys, which require the pairing of one column from the child record to the primary key of the parent record. This implies access to an indexed column, which helps speed queries that make use of these relationships. Surprisingly, I have come across databases that did not formally use foreign keys. The relationships were there, but the extra step to add foreign key constraints to the database was not put forth. I am sure the developers who created such databases did not want to bother themselves with having to cope with foreign key constraints as they prepared their applications for the first release. A simple script to add and remove the constraints would have eliminated that problem.

I also find that when there are no foreign key constraints in a development database, the relationships can get a bit sloppy. You may have child records left behind when parent records are deleted, which adds up over time and fills your database with garbage data. When I work with a new database or maintain an existing one, I keep the foreign key constraints in place as I develop the stored procedures to ensure that I respect those relationships. It is simple enough to adjust a stored procedure to take the necessary steps to satisfy the requirements of a constraint.

Finally, there are many-to-many relationships logically, but they should not be represented directly with two tables. It is necessary to break such relationships into three tables, with the table in the middle acting as the link between two one-to-many relationships. In the current database, the FavoriteLinks database has a many-to-many relationship with the TagTokens table, which is organized with the LinkTags table. For this relationship, the same tag can be placed on the many links, and a single link can have many tags. I chose to call the table in the middle LinkTags. It is useful for the table in the middle to take parts of the related tables. This structure makes it easy to identify these “bridge” tables and to show the relationships at a glance.

Created and Modified

Whenever I create databases from the ground up, I like to add the created and modified values at the end of each table and maintain them for each insert and update. Having access to these

details is useful for many reasons. For starters, you can see how old some of the rows in a table are after the database has been in use for a long time. It gives you the opportunity to purge or archive older data that is not being accessed. You can also logically partition the older data based on the Created or Modified value. When a table grows very large, you may be accessing only the recently created or modified data. When the data is partitioned, you can run a scan on the subset that falls in the scope of your query without needing to run a full scan of the entire table.

Note Partitioning falls well within the realm of a database administrator and is not covered in depth in this book. When considering logical groupings of data, it will be useful to discuss partitioning strategies with your DBA to determine what can be done given your database structure and available hardware.

What About Nulls?

The decision about how to handle nulls is a hot-button issue for some development teams. Some prefer to not allow nulls at all and just use known default values, while others argue that a truly null value should be preserved as a null. The argument will hinge on how well you can relay the definition of a default value across all users of the database—which can be difficult if not impossible. Meanwhile, a null value is well known and will force the application developers to deal with it sooner or later. However, when you can prevent nulls from coming out of the database, you reduce the complexity of the data access layer.

When a column could return a null value, using the value requires the additional step to see whether the value can be safely cast to types such as `int` or `DateTime`, which do not allow null values. The additional check for a null value will have an impact on performance. The traditional way is to check whether the value matches the `DBNull.Value` and then cast it when it is not null. Listing 7-7 shows the traditional way to do so.

Listing 7-7. Traditional Code to Handle Nulls

```
DateTime birthDate;
if (!DBNull.Value.Equals(row["BirthDate"]) && row["BirthDate"] is DateTime)
{
    birthDate = (DateTime)row["BirthDate"];
}
```

Alternatively, you can streamline this code by using nullable types for `DateTime` and `int` so that you can directly cast the value using the `as` operator (see Listing 7-8).

Listing 7-8. Streamlined Code to Handle Nulls

```
DateTime? birthDate;
birthDate = row["BirthDate"] as DateTime?;
```

The streamlined code in Listing 7-8 uses the nullable types combined with the `as` operator to allow the null values to be assigned without the risk of a `NullReferenceException` or an `InvalidCastException`. The `as` operator first tests whether the cast is valid and whether it is completed. Checking for a match on `DBNull.Value` or using the `as` operator is unnecessary.

If you instead choose to use default values, you could use well-defined defaults that are used throughout the application. A natural choice for `String` is `String.Empty`, while types such as `int` and `DateTime` do not have obvious default values. Typically, a numerical value is positive, especially when it is used as a primary key set as an identity value starting at 0 and increasing. In this case -1 is the clear choice.

When it comes to `DateTime`, you may want to use `DateTime.MinValue`, but SQL Server does not allow that value for the `DateTime` type in the database because SQL Server limits the dates to greater than 1/1/1753. You will instead need to use an alternate value. Internally at Microsoft the value of 1/1/1754 is used, which will be a reasonable default value in most cases. Table 7-2 shows these default values.

Table 7-2. *Suggested Default Values*

<code>String</code>	<code>String.Empty</code>
<code>int</code>	-1
<code>DateTime</code>	1/1/1754

With these values set as constants in your data access layer, they can be checked throughout the rest of the application. At some point, if you choose to change the default values, you could update the constants to reflect the change throughout the application. When the data access layer and the database are both set to recognize the same default values, you can handle them properly with the stored procedures so that while you do not allow the stored procedures to return null values, you could still set nulls in the database.

Preventing Null Values

Assume that you do have a table that holds a `DateTime` column called `BirthDate` that does allow for null values. The stored procedure returning this value could use the `COALESCE` function to change the null value to the default date value. Listing 7-9 shows how to use the `COALESCE` function.

Listing 7-9. *Return a Default Date for a Null Value*

```
SELECT
    COALESCE(
        BirthDate,
        Convert(DateTime, '1/1/1754')
    ) AS BirthDate
FROM chpt07_Nullable
```

When sending a default value into the system, we can also convert the default value to a null for storage. This check and conversion is shown in Listing 7-10.

Listing 7-10. *Converting a Default Date to a Null*

```
DECLARE @DefaultDate DateTime
SET @DefaultDate = Convert(DateTime, '1/1/1754')
DECLARE @BirthDate DateTime
SET @BirthDate = Convert(DateTime, '1/1/1754')
IF (@BirthDate = @DefaultDate)
    BEGIN
        SET @BirthDate = NULL
    END
```

By handling default value conversion in a stored procedure, you get greater performance than is possible with CLR code and avoid the overhead of checking the type for `DBNull.Value` and casting the value.

COALESCE VS. ISNULL

The `COALESCE` function is a standard SQL function, while `ISNULL` is a proprietary function available with SQL Server. The `ISNULL` function is generally interchangeable with the `COALESCE` function and may offer better performance. You will need to run your own tests to gather metrics to check whether your individual queries would benefit from one over the other, because one can be faster than the other in certain conditions.

For the `FavoriteLinks` database, I chose to not allow null values at the table level. For this small application, it is reasonably possible to ensure that the default values are respected throughout the application. The application will also use validation to ensure that empty strings are not sent into the data access layer, which itself prevents null values from entering the system.

Using Database Projects

To construct the database, I used Database Projects, which are a feature of Visual Studio 2005 Professional Edition. Each table and stored procedure script is uniquely organized and source controlled, which makes it easier to manage changes to the database across releases of the application.

Managing the Scripts

The `FavoriteLinks` database has six tables, as shown in Figure 7-5. In addition to those tables, there are several scripts for stored procedures that access the tables in every way the application requires. There are also scripts to add and remove the constraints that maintain the relationships across the tables. All these scripts are organized into groups, in folders.

A hidden feature of Database Projects is the intelligent dependency analysis, which is done transparently when you select multiple scripts and run them. Visual Studio reviews the selected scripts, and based on the dependencies, it will run the scripts in the proper order to allow the scripts to run successfully. If you do happen to define a foreign key in a table script that references another table which is alphabetically listed after the dependency, the order will be adjusted when running the script.

Beyond foreign key constraints, stored procedures can also refer to other stored procedures. Selecting and running all of them will also help you with the order and will prevent the warnings you get from using a resource that has not been defined yet. I find that if I make several changes to all the stored procedures, selecting and running all of them is helpful after I am finished making the changes.

Planning the Stored Procedures

For the Favorite Links website, a user will store multiple links along with tags and other related data. To provide for this functionality, the stored procedures will cover all access functions from getting, saving, and purging the data. And although the relationships between the six tables are complex, the layer built on top of the table structure does not have to directly reflect that complexity. Beginning with the stored procedures, the abstraction will start to make the data easier to consume.

Getting Data

The first set of stored procedures to construct is the one to get the data from the database. The primary data will be the links, so we will look at those in detail. Among these, the central procedure will be the one getting all links related to a profile, as shown in Listing 7-11.

Listing 7-11. *chpt07_GetFavoriteLinksByProfileID.sql*

```
CREATE Procedure dbo.chpt07_GetFavoriteLinksByProfileID
(
    @ProfileID int
)
AS

SELECT
    fl.FavoriteLinkId AS ID,
    lt.Title,
    lu.Url,
    fl.Keeper,
    fl.Rating,
    fl.Note,
    fl.Created,
    fl.Modified
FROM chpt07_FavoriteLinks AS fl
JOIN chpt07_LinkUrls AS lu ON lu.LinkUrlId = fl.LinkUrlId
JOIN chpt07_LinkTitles AS lt ON lt.LinkTitleID = fl.LinkTitleID
WHERE fl.ProfileID = @ProfileID
ORDER BY fl.Created DESC

GO
```

We will compare this stored procedure to the next most used stored procedure, shown in Listing 7-12.

Listing 7-12. *chpt07_GetFavoriteLinksByTag.sql*

```
CREATE Procedure dbo.chpt07_GetFavoriteLinksByTag
(
    @ProfileID bigint,
    @Token nvarchar(30)
)
AS

SELECT
    fl.FavoriteLinkID AS ID,
    lt.Title,
    lu.Url,
    fl.Keeper,
    fl.Rating,
    fl.Note,
    fl.Created,
    fl.Modified
FROM chpt07_FavoriteLinks AS fl
JOIN chpt07_LinkUrls AS lu ON lu.LinkUrlId = fl.LinkUrlID
JOIN chpt07_LinkTitles AS lt ON lt.LinkTitleID = fl.LinkTitleID
JOIN chpt07_LinkTags AS lt2 ON lt2.FavoriteLinkID = fl.FavoriteLinkID
JOIN chpt07_TagTokens AS tt ON tt.TagTokenID = lt2.TagTokenID
WHERE fl.ProfileID = @ProfileID AND tt.Token = @Token
ORDER BY fl.Created DESC

GO
```

In both of the stored procedures, the result set is identical. Making the result sets identical is done intentionally to keep them compatible with each other. Later in the data access layer, the same business object will hold the data for the FavoriteLinks database and will require these values. The only differences in these stored procedures are the incoming parameters and how they filter the results. In *chpt07_GetFavoriteLinksByProfileID*, the *ProfileID* is used to access all the links saved to the specified profile. In *chpt07_GetFavoriteLinksByTag*, the *ProfileID* is provided again along with the *Token* parameter to limit the scope of the result to the matching tags. The series of joins used to bring together all the data remains consistent otherwise. There are other criteria for filtering the list of favorite links, which all return the same columns by using different parameters.

WHAT TO OPTIMIZE AND HOW

Improving the performance of a data access layer requires knowing what to optimize and how to go about doing it. By simply placing the “get” queries into a limited set of stored procedures, you allow SQL Server to focus on and tune the query plans for the queries that run most frequently. And because you query the database more than you save or remove data, you can spend most of your effort optimizing the process of getting data from the database. Over time, you can profile the performance of your limited set of stored procedures and further optimize them as needed.

Saving Data

Next, the stored procedures that are used to save links either with an insert or an update present a more challenging requirement. Not only does the procedure save the core details of the link, but it must take the `Title` and `Url` values and store them off to the secondary tables while not creating duplicate records of the same values. The design of the tables allows for a Favorite Link record to point to the same `Url` record that other Favorite Link records are already using. All the work to coordinate this will be managed in the `chpt07_SaveFavoriteLink` stored procedure. Starting out the stored procedure will take in all the parameters needed to save the data, as shown in Listing 7-13.

Listing 7-13. Parameters for `chpt07_SaveFavoriteLink`

```
CREATE Procedure dbo.chpt07_SaveFavoriteLink
(
    @ProfileID bigint,
    @Url nvarchar(250),
    @Title nvarchar(150),
    @Keeper bit,
    @Rating smallint,
    @Note nvarchar(500),
    @Created datetime,
    @Modified datetime,
    @OldFavoriteLinkID bigint,
    @FavoriteLinkID bigint OUTPUT
)
```

The first step in handling the save process is to save the `Url` and `Title` values to the secondary tables and get back the ID values to use for the foreign key references. Listing 7-14 shows where to get started.

Listing 7-14. Saving the Title and Url Values

```
DECLARE @LinkUrlID int
DECLARE @LinkTitleID int

EXEC chpt07_SaveLinkUrl @Url, @LinkUrlID OUTPUT
EXEC chpt07_SaveLinkTitle @Title, @LinkTitleID OUTPUT
```

After the values for `LinkUrlID` and `LinkTitleID` are set, it will be possible to save the main record. Because this is a save routine, the procedure could be inserting a new record or updating a new record. Before the decision between insert and update is possible, the value for the `@OldFavoriteLinkID` must be checked. If the value is positive, the save operation is meant to update an existing known record. Also, if the `@OldFavoriteLinkID` value is negative, meaning it is not defined, a check should be made for a Favorite Link record that already has the same `Url` and `Title` values. The code in Listing 7-15 handles this work.

Listing 7-15. *Checking an Existing Record*

```

IF (@OldFavoriteLinkID < 0 AND EXISTS (
    SELECT * FROM chpt07_FavoriteLinks
    WHERE LinkUrlID = @LinkUrlID AND ProfileID = @ProfileID
))
BEGIN
    SET @OldFavoriteLinkID =
        (SELECT FavoriteLinkID FROM chpt07_FavoriteLinks
         WHERE LinkUrlID = @LinkUrlID AND ProfileID = @ProfileID)
END

```

Finally, we should be able to key the insert or update decision on whether the value of @OldFavoriteLinkID points to an existing record for the specified profile. If a record does exist, an insert is done. Otherwise, an update is done, as shown in Listing 7-16.

Listing 7-16. *Saving with Insert or Update*

```

IF EXISTS (SELECT * FROM chpt07_FavoriteLinks
           WHERE FavoriteLinkID = @OldFavoriteLinkID
           AND ProfileID = @ProfileID)
BEGIN
    UPDATE chpt07_FavoriteLinks
    SET
        LinkUrlID = @LinkUrlID,
        LinkTitleID = @LinkTitleID,
        Keeper = @Keeper,
        Rating = @Rating,
        Note = @Note,
        Modified = GETDATE()
    WHERE
        FavoriteLinkID = @OldFavoriteLinkID AND
        ProfileID = @ProfileID

    SET @FavoriteLinkID = @OldFavoriteLinkID
END
ELSE
BEGIN
    INSERT INTO chpt07_FavoriteLinks
    (ProfileID, LinkUrlID, LinkTitleID, Keeper,
    Rating, Note, Created, Modified)
    VALUES (
        @ProfileID,
        @LinkUrlID,
        @LinkTitleID,
        @Keeper,
        @Rating,
        @Note,

```

```

        @Created,
        @Modified
    )

    SELECT @FavoriteLinkID = @@IDENTITY
END
GO

```

In either case, the value of @FavoriteLinkID is set with the identity of the affected record, which is made available to the caller of the stored procedure as an output parameter.

Deleting Data

The last major work handled by the stored procedures is deleting data. For the FavoriteLinks database, it is easy to remove a reference to a TagToken record that drops the tag association from a Favorite Link record, but it is not a small matter of a single delete statement. To remove all TagToken records that are no longer being referenced, a few steps must be completed, starting in Listing 7-17.

Listing 7-17. First Step of the RemoveLinkTag Stored Procedure

```

CREATE Procedure dbo.chpt07_RemoveLinkTag
(
    @FavoriteLinkID bigint,
    @Token nvarchar(30)
)
AS

DECLARE @LinkTagID bigint

SET @LinkTagID = (
    SELECT TOP 1 lt.LinkTagID
    FROM chpt07_LinkTags AS lt
    JOIN chpt07_TagTokens AS tt ON tt.TagTokenID = lt.TagTokenID
    WHERE lt.FavoriteLinkID = @FavoriteLinkID AND tt.Token = @Token
)

DELETE FROM chpt07_LinkTags WHERE LinkTagID = @LinkTagID

```

This first step takes the @FavoriteLinkID and @Token parameters and gets the value for the @LinkTagID. After the ID is known, the record can be deleted. Now we must check whether the @Token value is referenced by any other records, as shown in Listing 7-18.

Listing 7-18. Counting the Token References

```

DECLARE @TokenCount int

SET @TokenCount = (
    SELECT COUNT(*)

```

```
FROM chpt07_LinkTags AS lt
JOIN chpt07_TagTokens AS tt ON tt.TagTokenID = lt.TagTokenID
WHERE tt.Token = @Token
)
```

If the count is zero, we know there are no references to it and the record can safely be deleted in the final step, shown in Listing 7-19.

Listing 7-19. *Deleting the Token Record*

```
IF (@TokenCount = 0)
BEGIN
    DELETE FROM chpt07_TagTokens WHERE Token = @Token
END
```

This process will keep the database as small as possible, which will help sustain the performance of queries using this data. A more complex scenario is completely purging all favorite links for a profile that deletes records for Tags, Tokens, Titles, and Urls in the right sequence to respect the foreign key constraints. For this sequence, we will break the process into multiple stored procedures to make it manageable. Table 7-3 lists these stored procedures.

Table 7-3. *Purging Stored Procedures*

chpt07_PurgeProfile	Purges profile
chpt07_PurgeFavoriteLinksByProfileID	Purges favorite links including titles and URLs
chpt07_PurgeLinkTagsByProfileID	Purges link tags and tokens

From a high level, the scripts start the process with the profile, which executes the stored procedure to purge favorite links, which in turn executes the stored procedure to purge tags. It works like a cascade, where all child references are removed before the parent record is removed so the foreign key constraint is not violated. This can be a tricky process, because you cannot delete a child record if a parent record is pointing to it. You must create a list of the child records before deleting the parent so that you can delete the child records afterward.

We will dig in at the bottom with the link tags handled by the chpt07_PurgeLinkTagsByProfileID stored procedure, starting in Listing 7-20.

Listing 7-20. *First Step of chpt07_PurgeLinkTagsByProfileID*

```
CREATE Procedure dbo.chpt07_PurgeLinkTagsByProfileID
(
    @ProfileID int
)
AS

IF EXISTS (
    SELECT * FROM chpt07_LinkTags AS lt
    JOIN chpt07_FavoriteLinks AS fl ON fl.FavoriteLinkID = lt.FavoriteLinkID
```

```

WHERE fl.ProfileID = @ProfileID
)
BEGIN

```

This first step checks whether any link tags exist before starting the sequence. Because the work done to purge link tags requires a little setup, this step helps avoid the extra work. Next, the list of records to purge is assembled. The code in Listing 7-21 shows the purge routine.

Listing 7-21. *Assembling the Token Purge List*

```

DECLARE @TagTokensToPurge TABLE
(
    ID int IDENTITY,
    TagTokenID bigint,
    [Count] int
)

INSERT INTO @TagTokensToPurge (TagTokenID, [Count])
SELECT tt.TagTokenID, COUNT(*) AS [Count]
FROM chpt07_TagTokens AS tt
JOIN chpt07_LinkTags AS lt ON lt.TagTokenID = tt.TagTokenID
JOIN chpt07_FavoriteLinks AS fl ON fl.FavoriteLinkID = lt.FavoriteLinkID
WHERE tt.TagTokenID in (
    SELECT DISTINCT tt2.TagTokenID
    FROM chpt07_LinkTags AS lt2
    JOIN chpt07_TagTokens AS tt2 ON tt2.TagTokenID = lt2.TagTokenID
    JOIN chpt07_FavoriteLinks AS fl2 ON fl2.FavoriteLinkID = lt2.FavoriteLinkID
    WHERE fl2.ProfileID = @ProfileID
)
GROUP BY tt.TagTokenID
HAVING COUNT(*) = 1

```

To hold onto the list, a table variable is declared and then populated with a query. This table variable holds onto the TagTokenID and the Count for the number of references to it. In Listing 7-22, the link tags are finally deleted.

Listing 7-22. *Deleting the Link Tag Records*

```

DELETE FROM chpt07_LinkTags WHERE LinkTagID IN
(
    SELECT lt.LinkTagID FROM chpt07_LinkTags AS lt
    JOIN chpt07_FavoriteLinks AS fl ON fl.FavoriteLinkID = lt.FavoriteLinkID
    WHERE fl.ProfileID = @ProfileID
)

```

With the parent records in the chpt07_LinkTags removed, it is now possible to delete the child records in the chpt07_TagTokens table, as shown in Listing 7-23.

Listing 7-23. *Deleting the Token Records*

```
DELETE FROM chpt07_TagTokens WHERE TagTokenID IN
(SELECT TagTokenID FROM @TagTokensToPurge)
```

One-half of the dependencies have now been purged properly. Now the titles and URLs must be purged. This work is done by the stored procedure named `chpt07_PurgeFavoriteLinksByProfileID`, which starts off in Listing 7-24.

Listing 7-24. *First Step in chpt07_PurgeFavoriteLinksByProfileID*

```
CREATE Procedure dbo.chpt07_PurgeFavoriteLinksByProfileID
(
    @ProfileID bigint
)
AS

SET NOCOUNT ON

IF EXISTS
    (SELECT * FROM chpt07_FavoriteLinks WHERE ProfileID = @ProfileID)
BEGIN

    EXEC chpt07_PurgeLinkTagsByProfileID @ProfileID
    ...
```

The first step is to check whether there is any work to do, so the `@ProfileID` is used to look for any existing records. If there are, the previous stored procedure to purge link tags is executed. The next step is to assemble the list of URLs to purge, shown in Listing 7-25.

Listing 7-25. *Assembling the URLs Purge List*

```
DECLARE @LinkUrlsToPurge TABLE
(
    ID int IDENTITY,
    LinkUrlID bigint,
    [Count] int
)

INSERT INTO @LinkUrlsToPurge (LinkUrlID, [Count])
SELECT lu.LinkUrlID, COUNT(*) AS [Count]
FROM chpt07_LinkUrls AS lu
JOIN chpt07_FavoriteLinks AS fl ON fl.LinkUrlID = lu.LinkUrlID
WHERE lu.LinkUrlID in (
    SELECT DISTINCT LinkUrlID FROM chpt07_FavoriteLinks
    WHERE ProfileID = @ProfileID
)
GROUP BY lu.LinkUrlID
HAVING COUNT(*) = 1
```

Again, a table variable is used to hold the list of records. This time it holds the `LinkUrlID`. This step is repeated for the titles in Listing 7-26.

Listing 7-26. *Assembling the Titles Purge List*

```
DECLARE @LinkTitlesToPurge TABLE
(
    ID int IDENTITY,
    LinkTitleID bigint,
    [Count] int
)

INSERT INTO @LinkTitlesToPurge (LinkTitleID, [Count])
SELECT lt.LinkTitleID, COUNT(*) AS [Count]
FROM chpt07_LinkTitles AS lt
JOIN chpt07_FavoriteLinks AS fl ON fl.LinkTitleID = lt.LinkTitleID
WHERE lt.LinkTitleID in (
    SELECT DISTINCT LinkTitleID FROM chpt07_FavoriteLinks
    WHERE ProfileID = @ProfileID
)
GROUP BY lt.LinkTitleID
HAVING COUNT(*) = 1
```

The purge lists for URLs and titles hold only records that have a single reference. Because the records that are about to be deleted are the ones holding the reference, the last step is to simply delete all the parent records and all the items in the purge lists, as shown in Listing 7-27.

Listing 7-27. *Final Step for Purging Favorite Links*

```
DELETE FROM chpt07_FavoriteLinks WHERE ProfileID = @ProfileID

DELETE FROM chpt07_LinkUrls WHERE LinkUrlID IN
(SELECT LinkUrlID FROM @LinkUrlsToPurge)

DELETE FROM chpt07_LinkTitles WHERE LinkTitleID IN
(SELECT LinkTitleID FROM @LinkTitlesToPurge)
```

With all the stored procedures in place to get, save, and delete the data, we can start on the data access layer, which uses all the stored procedures.

The Data Access Layer

Although the stored procedures are a critical element of the data access layer, many consider the real data access layer to start where the C# code begins. In many cases, that is true. This is the case when Typed DataSets are used to automatically bind table adapters to the tables in the database. But the work done in this chapter directly marries the C# code to the stored procedures to distribute the work in a way that is designed to enhance performance.

To create a data access layer that works well with data binding, we will create a domain object that has a default constructor so that it can be instantiated declaratively with an

ObjectDataSource. We will also create a business object that represents a favorite link. This object will encapsulate the inner workings of the data access layer and just expose the data to the application layer. By hiding the implementation details, updating future releases is easier. It makes it possible to rework the implementation without requiring the rest of the application to be updated.

COMMON FOLDER ADDITIONS

The get, save, and delete stored procedures are going to be commonly reused with different values. You could take a set of these stored procedures from the downloadable code samples and create templates for yourself and add them to your Common folder in the Templates folder (D:\Projects\Common\Templates\Data Access Layer). Starting with these templates should save you time when putting together your data access layers.

Creating the Class Library

The data access layer should be reusable, and the first step to making it so is placing it in a class library. The alternative is to include the data access layer directly within your application. In the case of a Website Project, you would place the classes in the App_Code folder. Doing so does not allow any other projects to use those classes as a dependency. Our goal will be to treat the data access layer as an internal release that is used by multiple applications even if it is initially going to be used by only a single website. Doing so does not require much additional effort and helps keep the priorities of the application and the data access layer separate.

The organization of this project should already include the Database Project as a part of a solution. Now a class library can be added to the solution to hold onto the Data Access Library. I prefer to give the projects generic names, so the Database Project is named Database, and the class library is simply called ClassLibrary. This simplifies the scripting that will be covered in Chapter 8 because a nearly identical MSBuild script can be used across multiple solutions with minimal adjustments. Naturally, as the ClassLibrary project takes on a significant number of features, it can be broken into multiple class libraries. With proper use of namespaces, it should be simple enough to redistribute the namespaces across multiple class libraries with minimal impact on the applications using the libraries. In fact, in the properties panel for the class library, the value for the Default Namespace should be set to the name of the application, and classes should be added to folders within the project. For this chapter, the value of the Default Namespace is Chapter07, while all of the code for the data access layer will go into a folder called Domain. The namespace automatically set when creating new classes will be Chapter07.Domain.

Separation of Concerns

To preserve the modular nature of the data access layer, it may be helpful to create a team to work on the data layer while another team works on the application. The application developers can handle the concerns of the business while requesting the necessary features from the data team. This clean separation helps ensure that application-specific details do not get baked into the data access layer. When a new feature is required for the application, the data team can consider how to best provide that feature without compromising the integrity of the

data access layer. Later, after the class library holding the data access layer is used by multiple applications, the library should not have application-specific details that the next application will not be able to provide, making it harder to be reused.

Creating the Data Access Methods

Creating the data access methods is done easily with the Data Access Application Block from the Enterprise Library, using the code snippets covered in Chapter 2. We will start with the class library and a folder called Domain. In that folder, we will create a class named FavoriteLinkDomain. With the code snippets from Chapter 2 in place, we will create a new variable by using the first Data Access code snippet, which has the shortcut **datypes**. You can either press the Ctrl+K, Ctrl+X key sequence to open the snippet selector and navigate to this first snippet, or you can use the shortcut and type **datypes** and press Tab twice to instantly insert that snippet. This first snippet simply declares a reference to the Database object, which will be used throughout the rest of the class.

Next we need to instantiate the Database object in the default constructor. The second snippet does this work and has the **dacreation** defined as the shortcut. Create the default constructor for the domain class and insert this code snippet there. This snippet includes a placeholder for the connection string name. For now, we will set it as FavoriteLinks so that it looks like Listing 7-28.

Listing 7-28. *Default Constructor for FavoriteLinkDomain*

```
public FavoriteLinkDomain()
{
    db = DatabaseFactory.CreateDatabase("FavoriteLinks");
}
```

DATABASE CONFIGURATION

A simple application may have only a single database connection. In such an application, it will be possible to configure the data access layer to use the default database connection as a part of the Enterprise Library configuration. In Chapter 8, custom configurations will offer greater flexibility than hard-coding connection string names into the data access layer.

The next step is to create the methods that get the data from the database. The Get DataSet Method snippet will create a method that calls a stored procedure and constructs a DataSet with the result. This snippet has **dagetds** as a shortcut. Listing 7-29 shows one of the get methods.

Listing 7-29. *Get Recent Favorite Links Method*

```
public DataSet GetRecentFavoriteLinksByProfileID(
    long profileId, int startDaysBack, int endDaysBack)
{
    DataSet ds;
```

```

try
{
    using (DbCommand dbCmd = db.GetStoredProcCommand(
        "chpt07_GetRecentFavoriteLinksByProfileID"))
    {
        db.AddInParameter(dbCmd, "@ProfileID",
            DbType.Int64, profileId);
        db.AddInParameter(dbCmd, "@StartDaysBack",
            DbType.Int32, startDaysBack);
        db.AddInParameter(dbCmd, "@EndDaysBack",
            DbType.Int32, endDaysBack);

        ds = db.ExecuteDataSet(dbCmd);
    }
}
catch (Exception ex)
{
    throw GetException(
        "Error in GetRecentFavoriteLinksByProfileID", ex);
}

//return the results
return ds;
}

```

This method takes in the three parameters and populates the `DataSet` with the result. For this example project, there are several other get methods that you will find included in the downloadable code for this chapter.

As an alternative to filling a `DataSet` with the results from a stored procedure, you can also use the snippet that uses an `IDataReader` to return the data. This snippet has `dagetdr` as the shortcut. It looks identical to the `DataSet` method but returns an `IDataReader` instead of a `DataSet`.

The save methods work differently than the get methods. Instead of populating a `DataSet` or an `IDataReader`, they simply execute a nonquery and get back an optional output parameter. The code snippet for the nonquery methods has `danonquery` as the shortcut. Listing 7-30 shows a save method.

Listing 7-30. *Save Favorite Link Method*

```

public long SaveFavoriteLink(
    long profileId, string url, string title,
    bool keeper, int rating, string note, string tags,
    DateTime created, DateTime modified, long oldFavoriteLinkId)
{
    long favoriteLinkId;

    try
    {

```

```

using (DbCommand dbCmd =
    db.GetStoredProcCommand("chpt07_SaveFavoriteLink"))
{
    db.AddInParameter(dbCmd, "@ProfileID",
        DbType.Int64, profileId);
    db.AddInParameter(dbCmd, "@Url",
        DbType.String, url);
    db.AddInParameter(dbCmd, "@Title",
        DbType.String, title);
    db.AddInParameter(dbCmd, "@Keeper",
        DbType.Boolean, keeper);
    db.AddInParameter(dbCmd, "@Rating",
        DbType.Int16, rating);
    db.AddInParameter(dbCmd, "@Note",
        DbType.String, note);
    db.AddInParameter(dbCmd, "@Created",
        DbType.DateTime, created);
    db.AddInParameter(dbCmd, "@Modified",
        DbType.DateTime, modified);
    db.AddInParameter(dbCmd, "@OldFavoriteLinkId",
        DbType.Int64, oldFavoriteLinkId);
    db.AddOutParameter(dbCmd, "@FavoriteLinkId",
        DbType.Int64, 0);

    db.ExecuteNonQuery(dbCmd);
    object obj = db.GetParameterValue(dbCmd, "@FavoriteLinkId");
    favoriteLinkId = (long) obj;
}
}
catch (Exception ex)
{
    throw GetException("Error in SaveFavoriteLink", ex);
}
SaveLinkTags(favoriteLinkId, tags);
return favoriteLinkId;
}

```

The methods to delete or purge data also use the nonquery snippet to call the stored procedure to carry out the work necessary to remove data.

Each of these methods to get, save, and remove data work with raw data. This primitive data is bound directly to the stored procedures, which act as the proxy between the data access layer and the underlying tables in the database. On top of these methods, the objects that are used by the user interface layer should represent the data in a way that makes sense to the application owners and the developers who must maintain this software over the long term.

Handling Exceptions

In these data access methods, the call into the database is always wrapped with a `try...catch` block. I have two main reasons for including this extra layer of protection. First, I want to be able to handle the problem within the data access layer, where I know how to handle the problem. Second, I want to trap in the data access layer any exception that may expose information about the database to a user. Doing so will head off problems that could lead to a security hole. However, the addition of the `try...catch` block can reduce performance. This choice was made with that fact in mind. The performance penalty should be minimal.

To avoid the performance hit of entering this `try...catch` block, I can still use caching. If the data to be requested by the database query is already in the cache, the call to the database can be avoided along with the exception handling. And if you choose to lose the `try...catch` block completely, you can strip it off and just leave the `using` clause for resource management of the `DbCommand`. You may find that the trade-off for performance is not significant enough to give up the stability and security that the close exception handling offers.

Creating Business Objects

When you work with your application, you will want to pass around objects that make sense to your business. A business object such as a `Customer` has properties such as `FirstName`, `LastName`, `CurrentOrder`, and `LastOrder`. This business object fully implements the three parts of an object: data, behavior, and relationships. A `Customer` has a first and last name and a relationship to his current and last order. A `Customer` can also complete his `CurrentOrder`. These are all processes that an application owner understands and conveys to the development team through carefully designed business objects. Instead of having disjointed blobs of data held in a `DataSet` that directly mirrors the structure of your relational database, you have the opportunity to truly represent the real-world objects for the application. In addition to properly representing the real-world objects, you can also fully encapsulate the data access layer behind these business objects, which gives you the opportunity to use techniques such as lazy loading to boost performance while maintaining a consistent public interface.

The central business object of the Favorite Links website is the `FavoriteLink` object. As a business object, it holds onto the various properties of a link such as the `Title`, `Url`, `Rating`, `Note`, and `Tags`. Each of these properties is populated by columns returned from calls to stored procedures. Mapping these values can be a considerable amount of work, especially in the early stages of your application, when you are still adjusting the columns and properties. Updating the mappings can be time-consuming.

Loading Data

Perhaps the strongest argument against manually creating your own data access layer is the amount of work necessary to map the data from the database to the objects. Object/Relational (O/R) mapping has been a common problem since objects have been used in combination with relational databases. The impedance mismatch between these generally incompatible systems requires you to understand how both ends are constructed so that you can use that understanding to map them together.

One approach to O/R mapping has been to create configuration files that explicitly set the mapping between every data column and object property. Popular O/R mapping tools such as `NHibernate` use extensive XML configuration that defines all the mappings. Many consider

this to be a powerful solution to the O/R mapping requirement. I find that it is an extra, unnecessary layer.

To facilitate the O/R mapping work, I created a base class that automatically maps data columns to properties by matching up the names and types for properties that are writable. Every business object populated by the data access layer inherits this base class. One load method takes a `DataRow` while another takes an `IDataReader` object to set the properties. With reflection, the base class compares the columns coming from the query with the properties in the business object. I call this base class `DomainObject`. The core properties it always provides are `ID`, `Created`, and `Modified`. As I add new columns to the stored procedures, and new properties to the business objects, the values are automatically loaded into the objects without any additional effort. (See the code downloads for the `DomainObject` class.)

I made sure that `DomainObject` would work with either a `DataSet` or an `IDataReader` because both are worthwhile in different scenarios. I always want to maintain the flexibility of switching between them. For example, I can cache a `DataSet` deep in the data access layer because it is holding onto the data, while an `IDataReader` is just a data enumerator that can be used only once. But `IDataReader` is faster, so if I do not cache the data, I can leverage the speed of the `IDataReader`.

When data is pulled from the database and the code loops over the results, we will pass the `DataRow` or `IDataReader` into the load method of the business object to prepare it to be sent back to the application layer. These objects are held in a `FavoriteLinkCollection` object, which is shown in Listing 7-31.

Listing 7-31. *FavoriteLinkCollection Object*

```
using System.Collections.Generic;

namespace Chapter07.Domain
{
    /// <summary>
    /// Collection of FavoriteLink objects
    /// </summary>
    public class FavoriteLinkCollection : List<FavoriteLink>
    {
    }
}
```

Notice that the `FavoriteLinkCollection` object simply inherits from the generic collection `List<FavoriteLink>`, which gives us a clean object that can be extended later. It is also strongly typed with the `FavoriteLink` business object. A `DataSet` can be used to populate a `FavoriteLinkCollection`, as shown in Listing 7-32.

Listing 7-32. *Loading with DataRow*

```
private void AddToFavoritesLinkCollection(
    FavoriteLinkCollection collection, DataSet ds)
{
    if (ds.Tables.Count > 0)
    {
```

```

        foreach (DataRow row in ds.Tables[0].Rows)
        {
            FavoriteLink fl = new FavoriteLink(row);
            collection.Add(fl);
        }
    }
}

```

A constructor for the `FavoriteLink` object simply takes a `DataRow` and internally calls the `load` method defined by the `DomainObject`. Alternatively, an `IDataReader` could be used for potentially better performance, as shown in Listing 7-33.

Listing 7-33. *Loading with IDataReader*

```

private void AddToFavoritesLinkCollection(
    FavoriteLinkCollection collection, IDataReader dr)
{
    while (dr.Read())
    {
        FavoriteLink fl = new FavoriteLink(dr);
        collection.Add(fl);
    }
}

```

The `FavoriteLink` object also includes a constructor that takes an `IDataReader`, which is passed along to the `load` method. In the case of the `DataSet` and the `IDataReader`, the columns used to populate `FavoriteLink` should be identical, especially when they call the same stored procedures.

Tuning the Data Load

Because of the performance hit that is incurred, when using reflection there is a clear performance difference from using it to load data vs. directly casting the data to the properties of the business object. If you know a data column named `Title` is a `String`, you can simply set the property while directly casting the object, as shown in Listing 7-34.

Listing 7-34. *Directly Casting a Data Value to a Property*

```

this.Rating = (int)row["Rating"];

```

If the column is an `int` and the value is not null, this will work—but that is never a guarantee. Because the database is disconnected and can be updated separately from the application, changes to the database can be a potential problem. The approach using reflection creates mappings that match the names and types. Every time the mappings are used to set the property values, the data column is always checked for a null value, and this check eliminates the risk of casting the type incorrectly. This protection does come at a cost. Fortunately, that cost has been minimized.

The `FavoriteLink` object features the ability to switch between the reflection and direct casting modes by setting a static enum value. This setting allows for testing the performance of

both modes. The unit tests for this data access layer include load tests that measure the time it takes to load a large set of `FavoriteLink` objects and then display the average time for both modes. These load tests are included with the code downloads for this chapter.

These load tests, when run with a high number of items, showed that the reflection cost does not have a significant impact on the time to load the data into the `FavoriteLink` objects. The performance has been refined to minimize the cost of reflection. The mappings created by using reflection are created once when the first object is loaded. The mappings are reused for every new instance so that the performance penalty is nearly eliminated while the benefits are gained.

However, if you want to take advantage of directly casting the data columns to properties without using reflection, the load methods are marked as virtual so you can override them.

Building the Website

With the data access layer ready to use, we can now start assembling the website. The data will be brought into the website by using `ObjectDataSources` that are associated with databound controls held in user controls. The user controls can be placed on any page, wherever they are needed. The pages will also use a single master page for the website.

The user controls created for the website will become the building blocks for the website, and these building blocks will interface directly with the data access layer. A user control may then expose properties, methods, and events to the parent container to provide rich functionality for the user interface layer.

You may choose to start building the website with just large pages, but you will soon have pages with 2,000-line code-behind files, which are hard to organize and maintain. Starting with just a page is a good way to prototype a new interface, but ultimately you will want to break out the sections of a page into the user controls for maintainability and to take advantage of decoupling the various sections of a page into well-defined containers.

Ultimately, you may choose to convert a user control into a server control, which can be deployed purely as an assembly. This conversion makes the control work well as a dependency on many websites with minimal effort. A server control requires more work to create and maintain, so creating one from the start is not normally the best first step. As a user control matures, it may eventually be a good candidate for conversion. Currently, if you want to use a user control across multiple websites, you have to copy the `.ascx` markup file to each website along with either the code-behind file or a compiled assembly for the code-behind. If a user control will be useful across many websites, it will be desirable to take the extra step to make it into a server control.

Over time you may have a suite of user controls and server controls, much like all of the standard controls in the Toolbox in Visual Studio. This may lead to a set of databound controls much like the Login controls, which integrate directly with the Membership provider system. The following sample pages and user controls are designed with such a goal in mind.

Connecting the Data Access Layer

Because the data access layer is a class library within the same solution, it can be associated to the website by using a project reference. Because an ASP.NET website does not have a project file, the reference is stored by the solution instead. The connection string name is set to `FavoriteLinks`, so the `Web.config` file should include a connection string with the same name.

Using a hard-coded string for the connection string name may be sufficient when a website is simple, but as it grows and more components are created, coordinating all the hard-coded strings will become difficult.

As your website becomes more complex, you can implement custom configuration sections that define settings for each of your components, such as the connection string name. Then you can adjust the connection string name of a component by using the custom configuration. Creating a custom configuration is covered in Chapter 9. The sample project for this chapter uses a custom section to define the database connection.

Creating User Controls

User controls can be used to encapsulate at least one databound control and the associated `ObjectDataSource`. Working with the data in this isolated container makes it easier to handle the data and the features required of the databound control. This container can expose properties and events to communicate with the parent container.

Properties

To display the links, we will create the `LinksControl` user control, which exposes a few properties that the parent control should set to assist with binding the data. The properties used by the `ObjectDataSource` are `StartDaysBack` and `EndDaysBack`, which hold `int` values. These properties are shown in Listing 7-35.

Listing 7-35. *LinksControl Properties*

```
private int _startDaysBack = 7;

[Browsable(true), DefaultValue(7), Category("Links")]
public int StartDaysBack
{
    get { return _startDaysBack; }
    set { _startDaysBack = value; }
}

private int _endDaysBack = 0;

[Browsable(true), DefaultValue(0), Category("Links")]
public int EndDaysBack
{
    get { return _endDaysBack; }
    set { _endDaysBack = value; }
}
```

These are regular properties with member variables that hold onto the `int` values; however, the attributes decorating the properties are specific to user controls. The `Browsable` attribute allows the property to show up in the Properties window when the user control is selected in the design surface. The `Category` attribute places these two properties into the `Links` category so they show up together in the Properties window. Finally, the `DefaultValue`

attribute indicates the default value. This attribute causes the value in the Properties window to show up as bold when the value does not match the default value. These additional features assist the developer using the user control and further encapsulate the data as it is brought into the application.

With these properties set, the `ObjectDataSource` uses their values for parameters defined by the `ObjectDataSource` in Listing 7-36.

Listing 7-36. *LinksControls ObjectDataSource1*

```
<asp:ObjectDataSource ID="ObjectDataSource1" runat="server"
    OldValuesParameterFormatString="original_{0}"
    TypeName="Chapter07.Domain.FavoriteLinkDomain"
    SelectMethod="GetRecentFavoriteLinkCollection"
    OnSelecting="ObjectDataSource1_Selecting">
    <SelectParameters>
        <asp:Parameter DefaultValue="0"
            Name="profileId" Type="Int32" />
        <asp:Parameter DefaultValue="7"
            Name="startDaysBack" Type="Int32" />
        <asp:Parameter DefaultValue="0"
            Name="endDaysBack" Type="Int32" />
    </SelectParameters>
</asp:ObjectDataSource>
```

You can see the `startDaysBack` and `endDaysBack` parameters, which are set with the `ObjectDataSource1_Selecting` event handler shown in Listing 7-37.

Listing 7-37. *ObjectDataSource1_Selecting*

```
protected void ObjectDataSource1_Selecting(
    object sender, ObjectDataSourceSelectingEventArgs e)
{
    e.InputParameters["profileId"] = Utility.GetProfile().ProfileID;
    e.InputParameters["startDaysBack"] = StartDaysBack;
    e.InputParameters["endDaysBack"] = EndDaysBack;
}
```

All that is left is to bind the data to the databound control, which is a `Repeater` control shown in Listing 7-38.

Listing 7-38. *Repeater Control*

```
<asp:Repeater ID="rptLinks" runat="server"
    DataSourceID="ObjectDataSource1"
    OnItemDataBound="rptLinks_ItemDataBound">
    <HeaderTemplate>
    <div id="divLinks">
    <h3 class="Title" id="h3Title" runat="server">
    <asp:Literal
```

```

        ID="ltTitle" runat="server"
        Text="Title"></asp:Literal>
    </h3>
    <ul class="Links">
</HeaderTemplate>
<ItemTemplate>
<li>
<asp:HyperLink
    ID="hl1" runat="server"
    Text='<%= Bind("Title") %>'
    NavigateUrl='<%= Bind("Url") %>'></asp:HyperLink>
<span class="Rating">
    (<asp:Literal
        ID="lt1" runat="server"
        Text='<%= Bind("Rating") %>'>1</asp:Literal>)
    </span>
</li>
</ItemTemplate>
<FooterTemplate>
</ul>
<br style="clear: left;" />
</div>
</FooterTemplate>
</asp:Repeater>

```

By breaking up the content of the website into smaller user controls and controlling them through properties, you get a great deal of flexibility. The values from these properties can be relayed to the data access layer, which allows you to use the same user control multiple times while getting different results when they are displayed. Using user controls in this way essentially parameterizes their functionality and lines them up with the data access layer in a way that is much easier to manage.

Events

Events can also be used by a user control to relay information back to the parent control when a subscribed event occurs. Another user control called `TagCloudControl` displays a tag cloud with a set of links. When a user clicks on a link in the `TagCloudControl`, it raises an event that the parent page uses to change the content displayed by another user control called `Tagged-LinksControl`. Using events in this way further encapsulates the internals of the user control, making it more reusable.

The `TagCloudControl` declares one event named `TagSelected`, which is raised whenever one of the links in the tag cloud is clicked. The event and the method to raise the event are shown in Listing 7-39.

Listing 7-39. *TagSelected* Event

```

public event EventHandler<TagCloudEventArgs> TagSelected;

[Category("Tags")]

```

```
protected virtual void OnTagSelected(TagCloudEventArgs e)
{
    if (TagSelected != null)
    {
        TagSelected(this, e);
    }
}
```

The EventHandler uses a control EventArgs implementation named TagCloudEventArgs, which includes a property named Token that is made available to the event subscriber. When the event is raised, this value can be set as the Token property in the TaggedLinksControl. The TaggedLinksControl works much like the LinksControl but it uses the Token property to get the list of links using the Token property.

Note User controls can be declared in each page that uses them, but they can also be declared in the pages/controls section of the Web.config file so that the user controls can be referenced with a consistent tag prefix throughout the website. Normally, when you drag a user control onto the design surface, the generated declaration sets the tag prefix as uc1 or uc2, etc. Later, if a control is converted from a user control to a server control, there will be a single place in the website to adjust the reference.

Creating the Pages

Combining the user controls together on a page and wiring up their properties and events may now be the easiest step. With a master page already created, this page will just hold onto the user controls to list the links and the tag cloud. We can create a new page called Home.aspx and drag the user controls onto the design surface. The end result will look like Listing 7-40.

Listing 7-40. Home.aspx

```
<%@ Page Language="C#" MasterPageFile="~/FavoriteLink.master"
    AutoEventWireup="true"
    CodeFile="Home.aspx.cs"
    Inherits="Home"
    Title="Favorite Link: Home" %>

<asp:Content ID="Content1" runat="server"
    ContentPlaceHolderID="MainContentPlaceHolder">
    <div id="HomeContent">
        <div class="tagCloud" style="float: right; width: 150px;">
            <h3 class="Title">Tags</h3>
            <fl:TagCloudControl ID="tagCloud" runat="server"
                OnTagSelected="tagCloud_OnTagSelected" />
        </div>
        <div id="Links">
            <fl:TaggedLinksControl
```

```

        ID="taggedLinks" runat="server"
        Visible="false" />

        <fl:LinksControl
            ID="lcToday" runat="server"
            Title="Today"
            EndDaysBack="0"
            StartDaysBack="1"
            TitleVisible="true" />

        <fl:LinksControl
            ID="lcThisWeek" runat="server"
            Title="This Week"
            EndDaysBack="1"
            StartDaysBack="7"
            TitleVisible="true" />

        <fl:LinksControl
            ID="lcThisMonth" runat="server"
            Title="This Month"
            EndDaysBack="8"
            StartDaysBack="31"
            TitleVisible="true" />
    </div>
</div>
</asp:Content>

```

Each of the user controls does not declare the user control references at the top of the page because they have been predeclared in the `Web.config` file in the pages section, shown in Listing 7-41.

Listing 7-41. User Control Configuration

```

<pages>
  <controls>
    <add tagPrefix="fl" tagName="HeaderNavigation"
      src="~/Controls/HeaderNavigation.ascx"/>
    <add tagPrefix="fl" tagName="LinksControl"
      src="~/Controls/LinksControl.ascx"/>
    <add tagPrefix="fl" tagName="LoginControl"
      src="~/Controls/LoginControl.ascx"/>
    <add tagPrefix="fl" tagName="TagCloudControl"
      src="~/Controls/TagCloudControl.ascx"/>
    <add tagPrefix="fl" tagName="TaggedLinksControl"
      src="~/Controls/TaggedLinksControl.ascx"/>
  </controls>
</pages>

```

Each user control is given `fl` as the `tagPrefix`, which you can see in `Home.aspx`. This detail cleans up the page a bit and creates a consistent way of declaring each user control.

This main page lists the `LinksControl` three separate times to show links that were added today, this week, and this month. The `TagCloudControl` lists the tags associated with links added by the user. When the links are clicked, the `Token` property is captured by the `tagCloud_OnTagSelected` event handler, as shown in Listing 7-42.

Listing 7-42. *tagCloud_OnTagSelected*

```
protected void tagCloud_OnTagSelected(object sender, TagCloudEventArgs e)
{
    lcToday.Visible = false;
    lcThisWeek.Visible = false;
    lcThisMonth.Visible = false;
    taggedLinks.Visible = true;
    taggedLinks.Title = "Tagged with " + e.Token;
    taggedLinks.Token = e.Token;
}
```

These few lines of code are all that is needed to instrument all of the user controls on the page. All of their independent details are fully encapsulated and managed internally. Enhancing the performance of this website can now be done at various levels within the layers of the architecture. Some adjustments could be done with the tables and coordinated with the necessary changes to the stored procedures. Other adjustments could be done in the data access layer by introducing caching. And at the application layer, the user controls and pages could use additional techniques such as data or output caching. Each layer is fully independent and encapsulated so that the next layer will not need modification when optimizations are implemented.

Summary

This chapter covered the steps and choices to consider when manually creating a data access layer. You looked at how manually building this layer requires extra work in order to leverage the performance enhancements it offers. Finally, you connected the data access layer to a functional website by using the `ObjectDataSource` and user controls in a way that minimizes maintenance while leaving plenty of places to optimize performance without requiring major rework of the application.



Generated Data Access Layer

Not many developers like to touch the data access layer. Most developers prefer to stick to their favorite languages, such as C#, and avoid the languages they use rarely, such as T-SQL. This is understandable. A developer builds up a level of expertise and confidence with a language the more he uses it. So if you are lucky enough to have a skilled database developer on your team with deep knowledge of T-SQL, you surely leverage his skills as much as possible. However, these types of programmers are a rare breed. To cope with the situation, you can seek out ways to generate the data access layer and greatly reduce the time you will need to spend away from the language of choice, C#.

This chapter covers the following:

- Code generation
- SubSonic
- Bliinq

The common term used when discussing data access layers is CRUD, for Create, Read, Update, and Delete. Data access layers are mostly boilerplate code. If you take this to mean that for each table you just need to create the equivalent SQL statements, you can easily generate such code automatically by scanning the database and constructing classes that carry out all of the CRUD tasks. If you want to add smart functionality, such as relationships, you can read the foreign key constraints during the code generation process to infer the relationships and bake in the object-oriented functionality into your generated code. The tools covered in this chapter do all of this work for us in an elegant way that does not leave a bad taste in the mouth like Typed DataSets.

Code Generation

Code generation is an old concept. If you go all the way back to the beginning of software development, you will see how source files were compiled into assembly code, and developers at the time argued that a compiler did not produce code nearly as efficient as assembly instructions written by hand. The benefit of using a compiler was implementing the intent of the source code with a dramatically shorter development cycle. These days, we are several steps beyond this initial step that has taken us well into code generation. We write code in an IDE in the language we choose, it is converted into an intermediate language (IL), and eventually the Just-In-Time compiler converts it into machine code. The compilation process has been improved over the past few decades to introduce optimizations the developer does not

even know about yet benefits from continually. We have embraced code generation, and we are finding new ways to make use of it to further accelerate our development cycles while generating more efficient code than if we were to develop it ourselves.

When you work with ASP.NET, you are already making use of code generation with each page and user control that generates a partial class for each code-behind into memory. Each variable reference for the controls placed on the markup side of a page is placed in this partial class while you define the rest of the partial class in the code-behind file. You may remember the generated code region in ASP.NET 1.1 that warned you not to modify it directly. The code for this region is what is now safely generated into the hidden partial class. The code generation that makes this work is due to the build providers introduced with ASP.NET 2.0.

Note If you are not building a Website Project and are instead building a Web Application Project, you will still be using the old ASP.NET 1.1 model where the code is generated directly into the code-behind file. A Web Application Project cannot make use of build providers because it requires the dynamic compiler that is only available for websites. A Web Application Project is technically a class library, which does not have access to this functionality.

Build Providers

A build provider is a code generator that places the results of the build into memory so that it is accessible by the runtime. Visual Studio is able to also run the build providers and make the generated code available immediately without needing to compile the project. Build providers are preconfigured for all of the common file types such as .aspx, .ascx, .asmx, and several others. In the base Web.config file, which all websites inherit from, you will find a configuration element named buildProviders, which lists all of the mapped providers. In particular, the .aspx extension is mapped to a provider called PageBuildProvider, while .xsd is mapped to XsdBuildProvider, which creates Typed DataSets.

You can implement your own build provider and associate it with an extension of your choosing. Listing 8-1 shows how you would map a build provider for all files with .xyz as the extension.

Listing 8-1. Build Provider for .xyz Files

```
<buildProviders>
  <add extension=".xyz" type="Chapter08.BuildProviders.XyzBuildProviders" />
</buildProviders>
```

When the ASP.NET compiler comes across files with the .xyz extension, it will use the configured build provider to generate code into memory so that it can be used by pages and controls in a website as well as provide IntelliSense support. Figure 8-1 shows how the IntelliSense appears for the generated class.

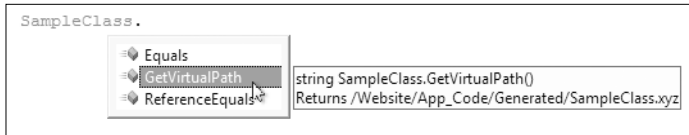


Figure 8-1. *IntelliSense for generated class*

The BuildProvider class, which is found in the `System.Web.Compilation` namespace, includes a method named `GenerateCode`, which you must override in order to inject your generated code into a website. Listing 8-2 shows an implementation of the `GenerateCode` method.

Listing 8-2. *GenerateCode Method*

```
public override void GenerateCode(AssemblyBuilder assemblyBuilder)
{
    assemblyBuilder.AddCodeCompileUnit(this,
        new CodeSnippetCompileUnit(GetGeneratedCode()));
}
```

The sample class that contains the `GenerateCode` method is named `XyzBuildProviders`, and it inherits the `BuildProvider` class. The first task you will likely want to do is read the contents of the source file. To read the file, you can either call `OpenStream`, which returns a `Stream`, or `OpenReader`, which returns a `Reader`. Listing 8-3 shows you how to get the contents of the source file as a string.

Listing 8-3. *GetContents Method*

```
private string GetContents()
{
    TextReader reader = OpenReader();
    string contents = reader.ReadToEnd();
    reader.Close();
    return contents;
}
```

The contents of a source file could easily be XML or a custom file format of your choosing that you can parse and use while constructing your generated code. For this example, you will just have a single line in the source file, which will be used as the comment for the class.

You may also want to know the name of the source file. Perhaps for each source file you want to generate a single class, and the filename of the source file will be reused as the name of the generated class. In this case, you can use the `VirtualPath` property, which is inherited from the base `BuildProvider` class. For a website, the source file must be in the `App_Code` folder, so a typical value would be `/AppCode/SomeFile.xyz`. Listing 8-4 shows how the class name can be extracted from this value.

Listing 8-4. *GetClassName Method*

```
private string GetClassName()
{
    int startIndex = VirtualPath.LastIndexOf("/") + 1;
    int length = VirtualPath.IndexOf(".") - startIndex;
    string className = VirtualPath.Substring(startIndex, length);
    return className;
}
```

You could also allow the class name to have some significance as well, but here you will assume the same namespace regardless of the directory structure. Finally, the `GetGeneratedCode` method called in Listing 8-2 is shown in Listing 8-5.

Listing 8-5. *GetGeneratedCode Method*

```
public string GetGeneratedCode()
{
    string className = GetClassName();
    string contents = GetContents();

    StringBuilder code = new StringBuilder();
    code.AppendLine("namespace Chapter08.Website {");
    code.AppendLine("/// <summary>");
    code.AppendLine("/// " + contents);
    code.AppendLine("/// </summary>");
    code.AppendLine("public partial class " + className + " {");
    code.AppendLine("/// <summary>");
    code.AppendLine("/// Returns " + VirtualPath);
    code.AppendLine("/// </summary>");
    code.AppendLine("public static string GetVirtualPath() {");
    code.AppendLine("return \"" + VirtualPath + "\";");
    code.AppendLine("}\n}\n}");

    return code.ToString();
}
```

The code generated in Listing 8-5 is simply a string returned to the caller that is used to create a new `CodeSnippetCompileUnit` class. For the moment, all you need to know is that this class reads in the string value for the code and passes it into the `AssemblyBuilder` class, which generates the code into memory.

Note A new dynamic language runtime (DLR) is currently under development for the .NET platform. It will support languages such as IronPython and IronRuby, which you can then use as your language of choice beyond C# and VB in a production environment. Many developers think this is great news because they feel these languages can do a great deal more with less code. It is an area worth some attention over the next few years as these technologies mature and become a core part of the .NET platform.

CodeDom Namespace

Generating code for the .NET platform is conveniently supported by the CodeDom namespace, specifically System.CodeDom. The namespace represents code as a document object model that can be manipulated just as easily as XML. You can create new classes and add fields, properties, and methods to them in a language-neutral way and then generate them out to supported languages such as C# and VB.

In Listing 8-5, the code was assembled as a string. You can do the very same work with CodeDom, which will give you more structure. By assembling the code as a string within quotes across multiple lines, you have to be careful to end each statement with a semicolon and close each block. When the code is assembled programmatically with CodeDom, these concerns go away.

Starting with a new custom build provider named *AbcBuildProvider*, you will define an XML file format with files using an *.abc* extension. The compilation section of the *Web.config* file now includes the additional build provider, shown in Listing 8-6.

Listing 8-6. *Adding Build Provider for .abc Files*

```
<compilation debug="true">
  <buildProviders>
    <add extension=".abc" type="Chapter08.BuildProviders.AbcBuildProvider"/>
    <add extension=".xyz" type="Chapter08.BuildProviders.XyzBuildProvider"/>
  </buildProviders>
</compilation>
```

The new file format will define a custom data class that the build provider will generate using a list of fields defined in each source file. Listing 8-7 shows a sample file.

Listing 8-7. *Person.abc*

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

<dataClass>
  <fields>
    <add name="FirstName" type="System.String"/>
    <add name="LastName" type="System.String"/>
    <add name="BirthDate" type="System.DateTime"/>
    <add name="Location" type="System.String"/>
  </fields>
</dataClass>
```

As with the *XyzBuildProvider*, this new build provider will use the name of the source of the generated class. The sample file, *Person.abc*, will create a class named *Person* with a set of fields and properties that are defined by the file. When the build provider and website are compiled, the *Person* class should be available to the website.

Starting with the *GetGeneratedCode* method, which was also created for the previous build provider, you will adjust it to return a *CodeCompileUnit* instead of a string, have it parse the source file as an XML stream, and process each of the fields defined within the *fields* element. Listing 8-8 shows the new method.

Listing 8-8. *GetGeneratedCode for AbcBuildProvider*

```

private CodeCompileUnit GetGeneratedCode()
{
    CodeCompileUnit code = new CodeCompileUnit();

    CodeNamespace ns = new CodeNamespace(GetNamespace());
    CodeNamespaceImport import = new CodeNamespaceImport("System");
    ns.Imports.Add(import);
    code.Namespaces.Add(ns);

    string className = GetClassName();
    CodeTypeDeclaration generatedClass =
        new CodeTypeDeclaration(className);
    generatedClass.IsPartial = true;
    ns.Types.Add(generatedClass);

    XmlDocument document = new XmlDocument();
    using (Stream stream = OpenStream()) {
        document.Load(stream);
        XmlNode rootNode = document.SelectSingleNode(RootPath);
        if (rootNode != null)
        {
            ProcessFieldNodes(generatedClass, rootNode);
        }
    }

    return code;
}

```

There are several new types referenced in the new `GetGeneratedCode` method. Types like `CodeNamespace`, `CodeNamespaceImport`, and `CodeTypeDeclaration` are all pieces of a `CodeCompileUnit`, which is used to create the generated class. The `CodeNamespace` type is the namespace that wraps the generated class. A single `CodeNamespaceImport` for `System` is added to the namespace. You would add others as needed just as you would to a source file you are editing in Visual Studio. Finally, the `CodeTypeDeclaration` is used for an instance of `generatedClass`, which will hold the fields that are defined in the `ProcessFieldNodes` method shown in Listing 8-9.

Listing 8-9. *ProcessFieldNodes*

```

private void ProcessFieldNodes(
    CodeTypeDeclaration generatedClass, XmlNode rootNode)
{
    XmlNodeList fieldNodes = rootNode.SelectNodes(FieldsAddPath);
    foreach (XmlNode addFieldNode in fieldNodes)
    {
        XmlNode nameNode = addFieldNode.SelectSingleNode("@name");
    }
}

```

```

XmlNode typeNode = addFieldNode.SelectSingleNode("@type");

string propertyName = nameNode.Value;
string fieldName = GetFieldName(propertyName);
Type fieldType = Type.GetType(typeNode.Value);

// private field
CodeMemberField field = new CodeMemberField(fieldType, fieldName);
generatedClass.Members.Add(field);

AttachProperty(generatedClass, propertyName, fieldName, fieldType);
}
}

```

The `ProcessFieldNodes` method starts by looping over all of the field nodes under the root node from the XML source file. The two nodes you want to access are the attributes in the `add` element named `name` and `type`. These references are done with XPath references such as `@name` and `@type`. At the top of the `AbcBuildProvider` are two constants defined to assist with traversing this XML document. These XPath constants are shown in Listing 8-10.

Listing 8-10. *XPath Constants*

```

public const string RootPath = "/dataClass";
public const string FieldsAddPath = "fields/add";

```

The first XPath constant, `RootPath`, accesses the first XML node, which then allows the second XPath constant, `FieldsAddPath`, to access all the `add` elements that hold onto the attributes we want to use. In Listing 8-9, these two nodes are read and later used to set the values for the `propertyName`, `fieldName`, and `fieldType` variables. Then a new `CodeMemberField` instance is created and added to the list of `Members` for the `generatedClass`. These are all private member variables that are accessed via properties defined by the `AttachedProperty` method shown in Listing 8-11.

Listing 8-11. *AttachProperty*

```

private static void AttachProperty(
    CodeTypeDeclaration generatedClass,
    string propertyName,
    string fieldName,
    Type fieldType)
{
    // public property
    CodeMemberProperty prop = new CodeMemberProperty();
    prop.Name = propertyName;
    prop.Type = new CodeTypeReference(fieldType);
    prop.Attributes = MemberAttributes.Public;

    CodeFieldReferenceExpression fieldRef;

```

```

    fieldRef = new CodeFieldReferenceExpression();
    fieldRef.TargetObject = new CodeThisReferenceExpression();
    fieldRef.FieldName = fieldName;

    // property getter
    CodeMethodReturnStatement ret;
    ret = new CodeMethodReturnStatement(fieldRef);
    prop.GetStatements.Add(ret);

    // property setter
    CodeAssignStatement assign = new CodeAssignStatement();
    assign.Left = fieldRef;
    assign.Right = new CodePropertySetValueReferenceExpression();
    prop.SetStatements.Add(assign);

    generatedClass.Members.Add(prop);
}

```

The `AttachProperty` method takes in the variables necessary to refer back to the related private member variable that the property will access. The property is created using the `CodeMemberProperty` type along with `CodeFieldReferenceExpression`, `CodeMethodReturnStatement`, and `CodeAssignStatement` classes. This method first creates the instance of the property, creates the reference to the field, and then adds the getter and setter statements to the property. At the end, the property is added to the `Members` collection of the `generatedClass`.

That is it for the heavy lifting. There are a few small private methods that help out the process. Listing 8-12 shows the `GetNamespace` method, which gets the string value from the configuration. Listing 8-13 shows the `GetFieldName` method, which converts a property name to field name using the naming conventions followed in all of the previous code samples with a preceding underscore before the field name.

Listing 8-12. *GetNamespace Method*

```

private string GetNamespace()
{
    string ns = ConfigurationManager.AppSettings["AbcNamespace"];
    if (String.IsNullOrEmpty(ns))
    {
        ns = "Abc";
    }
    return ns;
}

```

The setting for the `AbcNamespace` should be added to the website with the value of `Chapter08.Website`; otherwise it will default to `Abc`.

Listing 8-13. *GetFieldName*

```

private string GetFieldName(string propertyName)
{

```

```

return "_" +
    propertyName.Substring(0,1).ToLower() +
    propertyName.Substring(1);
}

```

Now when the website is compiled along with the build provider, it will generate a new class named `Person` with all of the fields shown previously in Listing 8-7. After the build has run successfully, you can open the Object Browser and see the newly generated class. Figure 8-2 shows the `Person` class with all of the generated fields and properties.

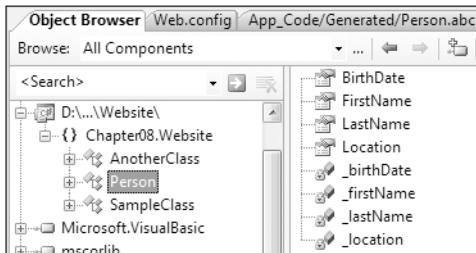


Figure 8-2. *Object Browser with generated Person class*

The original source file is ten lines of XML, and the resulting class that it generates is over 60 lines of code that I did not have to write. You can take this further: instead of having the XML file define the fields directly, the XML data could instead point to a table or stored procedure in the database that would provide all of the necessary details to generate all of these fields and properties. Several of the popular data access layer projects do just that. SubSonic and Blinq, which are covered later in this chapter, both read the schema from the database and generate classes based on what it finds. I will show you what these two projects offer. But first, consider how you would generate all of this code. Although you would be able to do it very quickly with CodeDom alone, you should consider templating.

Templating

While CodeDom gives you a structured way to generate code, using CodeDom can be tedious and difficult to maintain. CodeDom also feels unnatural. As developers, we are more comfortable with code that looks like code, not the derivative form that CodeDom uses to describe the pieces that make up the code. When you skim over a set of methods that use CodeDom to generate the code, you see references to `CodeTypeDeclaration` and `CodeMemberProperty`. When there is a lot of CodeDom code, it can be hard to follow and piece it together in your mind.

So when there is a lot of code to be generated, it helps to place sections of code into template files that can be read in a form that is as close to the actual code as possible. When the code is generated, the process will read these templates and load the resulting string into a `CodeCompileUnit` to generate the classes.

ASP.NET makes use of templates for pages and user controls that are made up of largely HTML content along with special directives specific to ASP.NET for controls. If it was not done this way, you would have to piece together all of the HTML from a source file as was done in the early days of web development.

Generating all of the fields from the XML file in the previous section only required about 100 lines of CodeDom code. That is pretty good, but it also only defined the fields and properties. Creating methods that call the stored procedures, potentially with several input parameters, would require a great deal more work with CodeDom. And as you have seen how a call into a stored procedure is very much boilerplate code, especially with the Enterprise Library, you can get a lot of value out of templating.

The first code generator I will review is SubSonic, which includes a rich templating system.

SubSonic

SubSonic is an open source project, partially hosted on CodePlex, that has become popular because of the many compelling features it provides. Recently, SubSonic is in its 2.0 release, which added support for the Enterprise Library as well as the ability to generate code for more than one database. Visit <http://www.subsonicproject.com> for more details. Here, I will focus on how it uses a template to generate the code and how it reads in details from the database to generate the code.

Note CodePlex (<http://www.codeplex.com>) is the newest code repository provided by Microsoft. It features Team System as the source control and work item tracking system. A free Team System-enabled version of Visual Studio is available for use with CodePlex. CodePlex also supports a basic bridge for TortoiseSVN, a Subversion client that integrates with Windows Explorer.

You will want to download the SubSonic project files for this section and place them into the common tools folder as shown in Figure 8-3. You will be referencing the command-line tool from this location as well as the SubSonic assembly. You will also need the SQL Server SDK installed if you want to compile the SubSonic project.

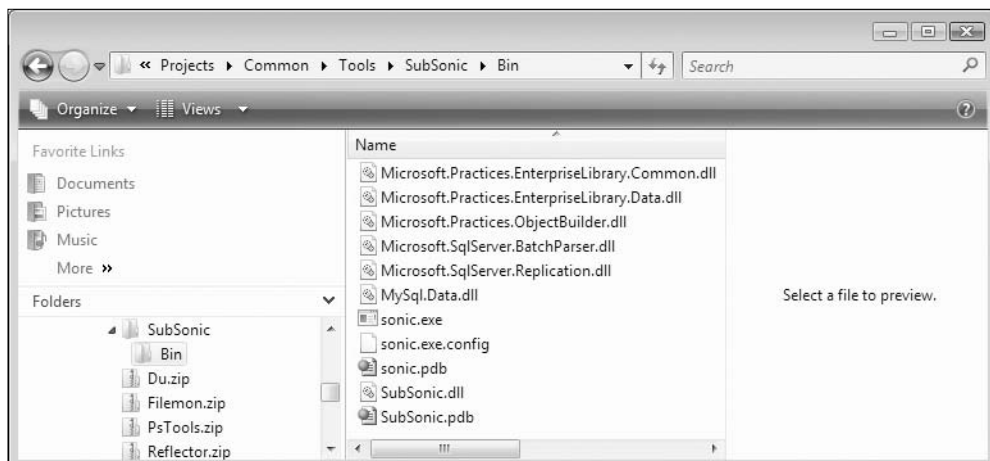


Figure 8-3. *SubSonic Tools directory*

Running SubSonic is done with the command-line tool called SubCommander, which processes multiple commands to generate both code and database scripts. The generate command recognizes many parameters. The most important parameters are used with the /config and /out switches. Listing 8-14 shows a sample command used to generate the code for a data access layer.

Listing 8-14. Running SubSonic

```
sonic.exe /config Website\Web.config /out Website\App_Code\Generated
```

The resulting classes can be included with your project, which could be a website or a class library. Originally, when SubSonic was created, it worked exclusively with ASP.NET 2.0 websites because it leveraged the Build Provider model, which only works with websites. To extend the reach of the generated code, the ability to generate the classes to source files was added. Limiting the code generation to the website prevented the data access layer from being used as a dependency for other types of applications such as desktop and console applications. Several developers also did not feel comfortable with the data access layer dynamically changing as the database changed.

Note See the SubSonic website (<http://www.subsonicproject.com>) for the full documentation covering the code generate command parameters as well as walkthrough videos.

SubSonic Templating

The templating system in SubSonic mirrors the templating system used by ASP.NET. The format of the files looks much like the .aspx markup files, but instead of generating HTML, it generates code for your data access layer. Figure 8-4 shows a sample template that is used to generate a class.

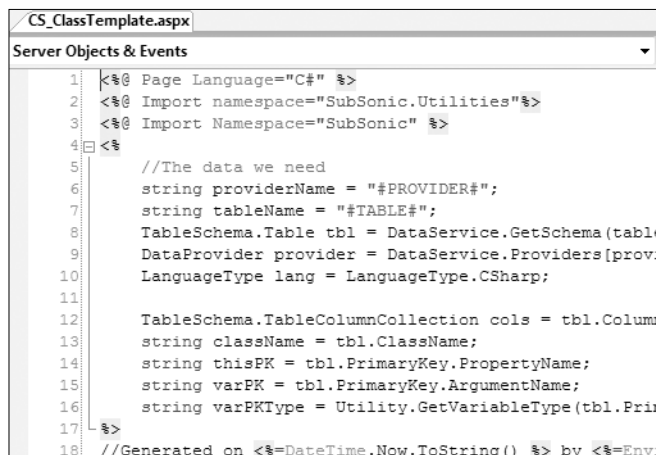


Figure 8-4. SubSonic class template

The .NET framework does not provide the facilities to process these template files except as a part of the ASP.NET runtime. To make these templates work for code generation in SubSonic, a parser and code processor was created for the project. It reads template files embedded within the assembly, and parses, processes, and generates the output code, which is then saved out to source files. All of the code to create working code from the templates is included with the source code download so that you can adjust it to your needs. You can also specify an alternative location for your own templates if you do not want to use the embedded templates. The `/templateDirectory` parameter is used to override the standard templating. To process the templates with values from the “outside world,” some placeholders are replaced while parsing and processing the templates. In Figure 8-3, you can see the `providerName` and `tableName` variables are set to `#PROVIDER#` and `#TABLE#`, which are placeholders that are replaced in the early steps of template processing. These variables are used throughout the rest of the template when the code is generated.

The advantages of emulating the `.aspx` template processing is that it supports not only placeholder replacement, but also loops. Listing 8-15 shows a loop within the class template that is used to generate all of the properties matching the columns from the database table.

Listing 8-15. *Loop in Template for Table Properties*

```
<%
foreach(TableSchema.TableColumn col in cols){
    string propName = col.PropertyName;
    string varType = Utility.GetVariableType(col.DataType, col.IsNullable, lang);
}%
[XmlAttribute("<%=propName%>")]
public <%=varType%> <%=propName%>
{
    get { return GetColumnValue[<=<%= varType%>[>]("<%= col.ColumnName %>"); }
    set
    {
        MarkDirty();
        SetColumnValue("<%=col.ColumnName %>", value);
    }
}
}<%
}%>
```

The properties generated for each table class will conform to the template in Listing 8-15, which includes calling the `MarkDirty` method. That simple call is something that could be a tedious detail if all of the code were written manually. Here, calling `MarkDirty` is automatic so that it is reliably called each time the value on a property is changed. This example is from the standard template, but you could change it to check whether the new value is different from the current value and only call `MarkDirty` as appropriate. If you wanted to do so now, you could copy the standard templates to your own templates directory and adjust this template to look like Listing 8-16.

Listing 8-16. *Adjusted Template for Table Properties*

```

<%
foreach(TableSchema.TableColumn col in cols){
    string propName = col.PropertyName;
    string varType = Utility.GetVariableType(col.DataType, col.IsNullable, lang);
    %>
    [XmlAttribute("<%=propName%>")]
    public <%=varType%> <%=propName%>
{
    get { return GetColumnValue[<]<%= varType%>[>]("<%= col.ColumnName %>"); }
    set
    {
        if (!GetColumnValue[<]<%= varType%>[>]("<%= col.ColumnName %>") ➡
.Equals(value)) {
            MarkDirty();
            SetColumnValue("<%=col.ColumnName %>", value);
        }
    }
}
}
<%
}
%>

```

Now the class will only be marked dirty when a property value changes. Because anything in the template can be changed, you could make many more changes. The code generated from SubSonic also leverages partial classes, which are commonly used for ASP.NET page code-behind files.

Note SubSonic works well to generate code that is very consistent so that the `MarkDirty` method is always called when a property value is changed. Similarly, you can use the Spring Framework to do what is called *dependency injection* to add similar behavior to your methods and properties without generating the code or even recompiling your existing code. (See <http://www.springframework.net>.)

Partial Classes

Opening up a class for extensibility can be done with partial classes. Such a feature is very helpful when much of the code is generated. The generated files can remain separate from the manually edited files so that your changes are not lost when you generate updates to the code. Each of the classes that the SubSonic generator creates is marked as a partial class. If you want to add functionality to the generated classes, you can create a new partial class with additional methods and properties. Assume you have a table called `Person` that has a couple of fields named `FirstName` and `LastName`. When the `Person` class is generated with SubSonic, you will get properties for these table columns. You could create a partial class to add a property called `FullName` as shown in Listing 8-17.

Listing 8-17. *FullName Property*

```

namespace Chapter08.SubSonicDAL
{
    public partial class Person : ActiveRecord<Person>
    {
        /// <summary>
        /// FirstName and LastName combined
        /// </summary>
        public string FullName {
            get {
                return this.FirstName + " " + this.LastName;
            }
        }
    }
}

```

With the `FirstName` and `LastName` properties already defined, you can access them as a part of the same class despite the fact that they are defined in another source file. You can also add a method to provide behavior that SubSonic does not provide natively. One standard method is the `FetchByID` method, which takes the primary key of the target table. It simply queries the database for the single record. The `FetchByID` method is defined in the `PersonController` class. What you could do is add caching behavior to the class with a nearly identical method signature. Instead of just passing in the ID parameter, you will add a Boolean value called `cached`. When it is true, it will use caching as shown in Listing 8-18.

Listing 8-18. *FetchByID with Caching*

```

namespace Chapter08.SubSonicDAL
{
    public partial class PersonController
    {

        public PersonCollection FetchByID(object ID, bool cached)
        {
            if (!cached)
            {
                return FetchByID(ID);
            }
            string cacheKey = (typeof(Person)).ToString() + "-" + ID;
            Cache cache = HttpRuntime.Cache;
            if (cache[cacheKey] != null) {
                return cache[cacheKey] as PersonCollection;
            }

            PersonCollection coll = new PersonCollection().Where ➤
(Person.Columns.ID, ID).Load();

```

```

        cache.Insert(cacheKey, coll, null,
            DateTime.Now.AddMinutes(5), TimeSpan.Zero);
        return coll;
    }
}

```

Caching the Person record in this way may eliminate a bottleneck in your application, but you will want to also ensure that as a Person record is updated, it is removed from the cache. The `ClearCachedItem` in Listing 8-19 takes care of removing the item from the cache.

Listing 8-19. *ClearCachedItem Method*

```

public void ClearCachedItem(object ID)
{
    string cacheKey = (typeof(Person)).ToString() + "-" + ID;
    Cache cache = HttpRuntime.Cache;
    cache.Remove(cacheKey);
}

```

Each time you call into the `Insert` and `Update` methods, you can ensure the `ClearCachedItem` method is called. Adding functionality in this way is a great way to optimize the application using this data access layer.

Query Tool

SubSonic uses a built-in query tool that generates the SQL used to get, save, and delete data in the database. This query tool was developed long before the LINQ technology was released as a Community Technology Preview (CTP), yet it has a significant amount in common.

The query tool uses a class named `Query`. It is constructed through a series of calls from many methods that refine the query. The goal is to allow the C# code to look much like a query written with SQL. Listing 8-20 shows an example method that loads all Person records by LocationID.

Listing 8-20. *FetchPersonsByLocationID Method*

```

public PersonCollection FetchPersonsByLocationID(object LocationID)
{
    return new PersonCollection().
        Where(Person.Columns.LocationID, LocationID).
        OrderByAsc(Person.Columns.LastName).Load();
}

```

There are a couple of important details about the `FetchPersonsByLocationID` method. The first detail is that the method is completely type safe. A strong reason for using Typed DataSets is for type safety, but because the code generated by SubSonic is based directly on the database, it automatically creates type-aware code. And in this method the call to the `Where` method could use a literal string, but instead the constant defined for the `LocationID` column is used.

The second detail is the fact that the query is completely constructed transparently by the query tool using the Person table with the LocationID column. It is extremely easy to write this query in C# with additional refinements as they are needed. The resulting query constructs a parameterized query that will be tuned nicely by SQL Server as the query is used repeatedly.

Scaffolding

One last major benefit to a generated data access layer is the ability to automatically hook up a scaffolding system to the generated code. *Scaffolding* is a set of interfaces that you can use to add sample data to your database without writing a single line of code. Many developers find this very useful, especially when the scaffolding is intelligent about relationships among tables. SubSonic includes an automatic scaffolding system that was inspired by work done on Ruby on Rails, a popular web framework built on the Ruby scripting language. SubSonic actually borrows several concepts from Ruby on Rails.

For SubSonic, you can use the AutoScaffold page that is in the SubSonicCentral website, which is a part of the downloadable source code. You can copy this page and the dependencies to the administrative section of your own website, and it will give you an interface for your database. Figure 8-5 shows the AutoScaffold page editing the Person table.

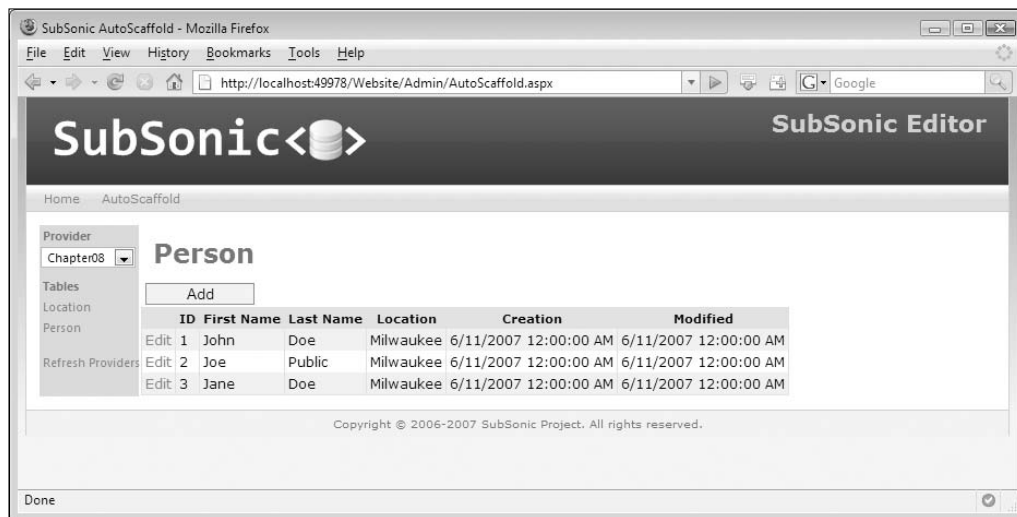


Figure 8-5. AutoScaffold page

While the AutoScaffold page will not be perfect, as the sole editor for your application it does provide automatic access to your data with very minimal work. And as you update your database and regenerate the data access layer, the AutoScaffold page will adapt to match the changes. No effort will be necessary on your part to keep it up to date.

In addition to simply wrapping an editor around tables, the AutoScaffold system is aware of relationships as well. The Person table has a foreign key reference to the Location table. This simple reference is enough for the AutoScaffold page to provide a drop-down list to select a location while editing a Person record. Figure 8-6 shows a Person record while in edit mode.

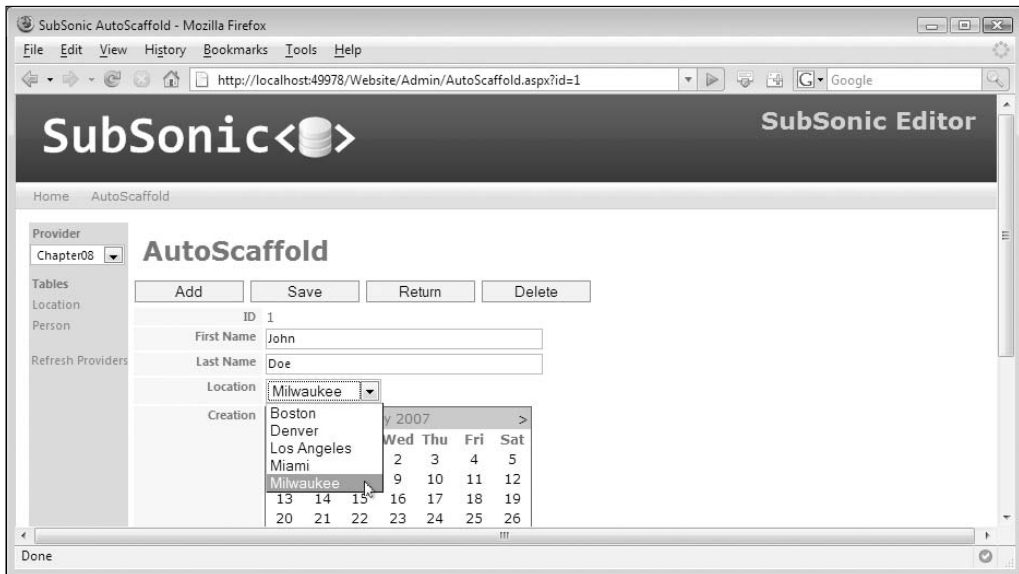


Figure 8-6. Editing a Person record with location reference

With all of the features that SubSonic offers, you should be able to get up and running very quickly. The generated data access layer and development experience made possible by SubSonic is dramatically different from what you get with Typed DataSets, which also generate a great deal of code but are extremely rigid and difficult to update as the database changes. As you start building an application and as changes happen, you should be able to adapt with ease. When you add a new column to a database, you can simply run the generate command to update your data access layer. And with partial classes and the code generating templates, you can easily modify the code generation process to fit your unique needs.

Blinq

Another code generator available for ASP.NET is Blinq, which builds on the LINQ technology that is a part of the .NET 3.5 release. LINQ stands for Language Integrated Query, and it extends the C# language. It essentially turns your application into an in-memory database, as you are able to query not only databases, but also collections of objects using syntax that is very similar to SQL.

The Blinq project is a prototype created as a sandbox project on the ASP.NET website (<http://www.asp.net>). The tool does not simply generate the classes for the data access layer. It actually generates a complete website with pages and a default theme. It is meant to act as a starting point for new projects.

COMMON FOLDER ADDITIONS

SubSonic and Blinq are tools that you can use purely for their scaffolding features, if not for your whole data access layer. You can place these tools in your Common folder in the Tools folder (D:\Projects\Common\Tools\Code Generators). Blinq will be installed in your Program Files folder, while you can unzip the SubSonic tools and assemblies into a folder you can reference from active projects.

Because the project references LINQ, it should generate optimized queries at runtime, as LINQ has been finely tuned to work with SQL Server. LINQ also works with other databases so long as they have LINQ provider implementation. The specific part of LINQ that queries databases is DLINQ, while XLINQ queries XML. See the MSDN documentation for a full introduction and primer to LINQ technologies. Here, I will focus on how to use Blinq, what features it offers, and how you can leverage it to build a real-world application.

To get started, you will need to download Blinq from the ASP.NET website. It is located in the sandbox section. It requires LINQ support, so you will need either the LINQ CTP or a preview of Visual Studio 2008, codename Orcas. Once you have Blinq installed, you can run the command in Listing 8-21 to generate a Blinq-powered website.

Listing 8-21. *Blinq Generate Command*

```
"C:\Program Files\Microsoft ASP.NET\Blinq\blinq.exe" /l:cs /server:.\SQLEXPRESS ➔  
/database:Chapter08 /namespace:Chapter08.BlinqDAL /t:$(BlinqTmpDir) /f
```

The Blinq utility can generate code for C# or VB with the /l switch. The value will be cs or vb with C# as the default output language. The /server and /database switches specify the server and database and assume they should use a trusted connection. The command in Listing 8-21 connects to a local SQL Express database and the Chapter08 database. The /namespace switch defines the namespace that the generated code will be placed under. The /t switch specifies the target directory where all of the files will be created. The file switch, /f, forces the existing file to be replaced when the command is run again.

The sample database used for the following example has the same structure that was used for the SubSonic samples. It simply includes a Person and Location table with a foreign key reference between them. The files created by running the Blinq utility are shown in Figure 8-7. And Figure 8-8 shows the website, which features a scaffolding system much like the SubSonic project does.

And again, you may not expect an end user to use this interface, but you can use it to enter the initial sample data as you develop the application. What you want to take away from this generated website is the code placed in the App_Code folder as well as the markup in the pages that provide access to the data.

The pages themselves have virtually no code. Everything is done with markup declarations with databound controls and ObjectDataSource references. These pages work well as a reference when you are working on your custom pages. Listing 8-22 shows the ObjectDataSource declaration used for the Person table.

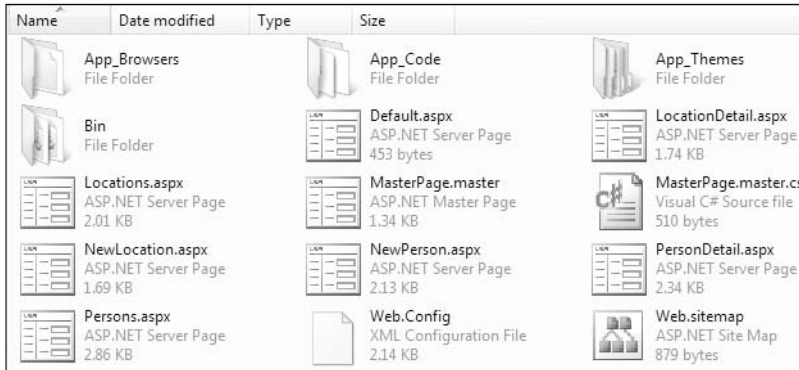


Figure 8-7. Blinq-generated files

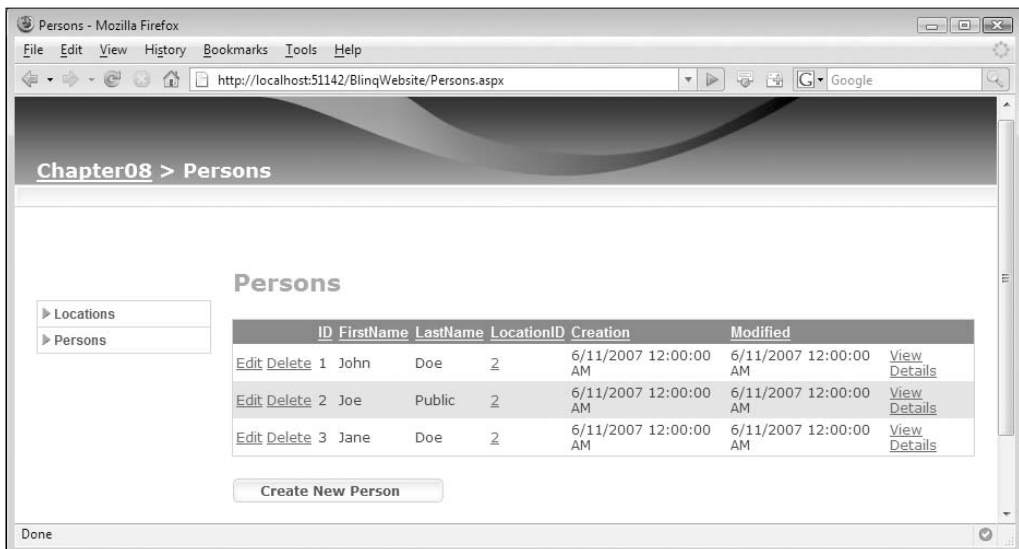


Figure 8-8. Blinq-generated website

Listing 8-22. ObjectDataSource for Person

```
<asp:ObjectDataSource ID="PersonsDataSource" runat="server"
    TypeName="Chapter08.BlinqDAL.Person"
    DataObjectType="Chapter08.BlinqDAL.Person"
    OldValuesParameterFormatString="original_{0}"
    ConflictDetection="CompareAllValues"
    SelectMethod="GetPerson"
    InsertMethod="Insert"
    UpdateMethod="Update"
    DeleteMethod="Delete"
    EnableCaching="True">
```

```

<SelectParameters>
  <asp:QueryStringParameter QueryStringField="ID"
    Name="ID"
    ConvertEmptyStringToNull="False">
  </asp:QueryStringParameter>
</SelectParameters>
</asp:ObjectDataSource>

```

The *Person* class created by Blinq in the *Chapter08.BlinqDAL* namespace includes the *GetPerson*, *Insert*, *Update*, and *Delete* methods. The ID value that is used to query the database for an individual record is set to be pulled from the query string.

The code generated by Blinq is placed in two source files, *Chapter08.cs* and *StaticMethods.cs*, in the *App_Code* folder of the website. The name of the first source file comes from the name of the database. You could pull these files out to a class library to make them reusable from a central point as a dependency. What is so interesting about Blinq is how simple and to the point the code is for working with the data. Listing 8-23 shows the *GetPerson* method from the *Person* class.

Listing 8-23. *GetPerson Method*

```

public static Person GetPerson(Int64 ID) {
    Chapter08 db = Chapter08.CreateDataContext();
    return db.Persons.Where(x=>x.ID == ID).FirstOrDefault();
}

```

The query is a single line. There is no stored procedure on the other end. The LINQ system creates all of the necessary SQL and uses it to get the data and populate the *Person* object. That is all there is to it! The LINQ query simply matches the ID column with the ID value passed into the method. Another method, *GetPersonsByLocation*, is a little more complex because it handles the relationship between the *Person* and *Location* tables. It is shown in Listing 8-24.

Listing 8-24. *GetPersonsByLocation Method*

```

public static IQueryable<Person> GetPersonsByLocation(Int64 ID) {
    Chapter08 db = Chapter08.CreateDataContext();
    return db.Locations.Where(x=>x.ID == ID).SelectMany(x=>x.Persons);
}

```

The *GetPersonsByLocation* query takes the ID for a *Location*, and from there it selects multiple *Person* records. As you can see, there is some query chaining happening here. This is going to become a familiar coding style as you use LINQ more and more in your work. The results from one query can be passed through another query to filter the final result. And while you may think the query appears to be inefficient, you will be pleasantly surprised that the inner workings of the LINQ runtime environment handle several optimizations that speed up your queries while allowing you to write code that is more readable to you.

I will get into LINQ a great deal more in Chapter 10 with a series of examples that will explain what is going on here in more detail.

Summary

In this chapter, you saw how code can be generated with build providers as well as a couple of powerful utilities that generate new source files. You should now understand how you can fully leverage an automatically generated data access layer, saving the time you would normally spend writing all of the code while still retaining the ability to extend the generated code through templating and partial classes.



Deployment

The largest impediment to improving the performance of an application is the cost of deployment. This process is associated with a certain amount of risk because it involves changing the production environment in a way that could potentially cause downtime. As a result, it requires a good deal of quality assurance testing as well as the resources necessary to push out the release, which may be done after hours to minimize the impact on the users of the application. This all adds up to a costly process, which may be enough to delay performance improvements until a release window that may be weeks or months away. With the right deployment model in place, the risk and cost can be greatly minimized so that timely updates can be pushed to the production environment.

This chapter covers the following:

- Automation with MSBuild
- Deploying the website
- Deploying the database
- Automating configuration changes
- Automating database updates
- Custom configuration sections

There is a right way and a wrong way to deploy your application releases. A few years ago I worked with a developer who would create a backup of the production database, download it, restore it locally, and make changes to the database. Then he would create a new backup, upload it, and restore it on the server. There are clearly some problems with this process, the memory of which makes me smile and cringe at the same time. I am glad that I have never had to follow this same process. There are far better ways to push out updates with the tools available today.

Prior to working with SQL Server, many developers hooked up Microsoft Access databases to classic ASP applications by uploading a copy of the database file to the server where the website could directly access the file. Because Microsoft Access was not typically installed on the server, a developer would take the application offline, pull down the database file, update it, and then put it back in place and bring the application back online. This old process may explain why the developer in the previous example used a very similar process with a SQL Server database. He was just doing what he was always doing, even though there is a better way.

While some of these old practices are becoming rare, I continue to see changes carried out manually from environment to environment. If all that was changed was adding columns and

perhaps adding an index, the steps are easy enough to carry out manually in each environment, so these developers do so each time without ever creating a script to carry out these changes. A specific example of such a change is altering a column from `varchar(20)` to `varchar(25)`. For small databases, such a minor change may be acceptable, but over time some changes may be forgotten, and the database environments become inconsistent. With a little preparation mixed with some automation, you can ensure that your environments remain consistent.

Automation with MSBuild

Updating the database with minor changes continually can be a tedious and repetitive job, but it has to be done. Fortunately, the job can be automated with minimal effort. The automation utility MSBuild is the standard build tool for the .NET platform. It is a scripting system that uses XML to define a series of build targets that include a set of tasks. The targets are chained together using dependencies. Visual Studio provides IntelliSense support for the MSBuild scripts that will help you get started once you know the basics.

MSBuild scripts can do nearly anything. And if a standard task does not do what you need, you can either use the `Exec` task to call out to a command line to run another utility or create a custom task. But you do not want to go too far with MSBuild scripts. I follow three overriding rules when I work with MSBuild scripts.

First, I keep the work that the script does to the bare minimum. It can be tempting to automate absolutely everything, but you will end up spending more time maintaining the script instead of getting work done on the application, which should be your primary concern.

Second, I only automate the time-consuming work that I find myself doing over and over. The tasks that take several minutes multiple times a day are ideal automation candidates. It may take me a while to add the functionality to the script, but it will ultimately add up to saving many hours of work for each release.

Finally, I keep the script's version controlled with the solution. As the build process changes over time, you will update the build script to adapt to those changes. These adjustments should stay current with the rest of the application in source control. And as you produce releases, you will be able to pull an older version from source control and build it even if the build process has changed since the release was deployed. With release branching in your source control system, you can support a production version of the application while development continues on the main branch.

RELEASE BRANCHING

There are many ways to manage projects in source control. With modern source control systems such as Microsoft Team Foundation Server and Subversion, you can create branches to isolate changes on a project. You can create branches for each developer, bug fix, or any other reason you choose. However, the branching model that larger teams settle on eventually is *release branching*. Development is carried out on the main branch (trunk), and as releases are prepared, a new branch is created for the release so that it is isolated from continuing development. The application is then pushed from this release branch. Later, when the deployed application needs updates, the changes are made on the branch and merged back to the trunk as necessary.

An MSBuild script is an XML document that is made up of three primary elements: PropertyGroup, ItemGroup, and Target, which are all held in the root element, Project. This is the basic structure for every MSBuild script. The PropertyGroup element defines properties that are used throughout the rest of the script. The ItemGroup element defines collections of properties. The Target element contains multiple tasks such as MakeDir and Copy. Listing 9-1 shows the skeleton of a basic MSBuild script.

Listing 9-1. *Skeleton of a Basic MSBuild Script*

```
<?xml version="1.0" encoding="utf-8"?>

<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <PackageName>MyProject</PackageName>
    <DatabaseDirectory>Database</DatabaseDirectory>
    <WebsiteBinDirectory>Website\bin</WebsiteBinDirectory>
  </PropertyGroup>

  <ItemGroup>
    <!-- Collection Definitions -->
  </ItemGroup>

  <Target Name="Build">
    <!-- Build Tasks -->
  </Target>

</Project>
```

While the standard tasks included with MSBuild cover many common tasks, they do not cover everything you will need to automate your build and deployment work. One task in particular is the Zip task, which I use frequently to create collections of files so they can be easily archived and moved from environment to environment. Fortunately, there is an open source project called MSBuild Community Tasks that includes a Zip task among many other useful tasks (see the sidebar “MSBuild Community Tasks”).

MSBUILD COMMUNITY TASKS

MSBuild can be extended by creating custom tasks. An open source project that has become the gathering point for custom MSBuild tasks is the MSBuild Community Tasks project, which continually adds more custom tasks to the already extensive list (see <http://msbuildtasks.tigris.org/>). Tasks such as Zip, FxCop, and Prompt are a few examples that add key functionality to this scripting environment.

It is useful to note that some project files are also MSBuild scripts. The project file for a class library is an MSBuild script, and you can place tasks in the `BeforeBuild` and `AfterBuild` targets shown in Listing 9-2. These targets are commented out initially, but you can uncomment them. You can then place whatever tasks you want in them to be run before or after the build process.

Listing 9-2. *BeforeBuild and AfterBuild Targets*

```
<Target Name="BeforeBuild">
  <Message Text="### BeforeBuild ###" Importance="high"></Message>
</Target>
<Target Name="AfterBuild">
  <Message Text="### AfterBuild ###" Importance="high"></Message>
</Target>
```

You can also use the `PreBuildEvent` and `PostBuildEvent` properties, which work more like a batch file with a series of commands. These properties are empty by default, but you can call out to the command line. The example in Listing 9-3 runs the MSBuild utility with parameters that specify a script in the solution directory.

Listing 9-3. *PreBuildEvent and PostBuildEvent Properties*

```
<PropertyGroup>
  <PreBuildEvent>%25windir%25\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe ➡
"$ (SolutionDir)build.proj" /t:PreBuild /p:Configuration=$(ConfigurationName)
</PreBuildEvent>
  <PostBuildEvent>%25windir%25\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe ➡
"$ (SolutionDir)build.proj" /t:PostBuild /p:Configuration=$(ConfigurationName)
</PostBuildEvent>
</PropertyGroup>
```

With the details about the individual projects managed independently, you can create a build script at the solution level that will simply use the project files, calling the `Build` target. This is done with the MSBuild target. With the standard solution layout—that is, with the solution file in the root folder with projects in subfolders—it is common to place a file called `build.proj` in the root folder, which coordinates the automated processes.

The main build script, `build.proj`, will primarily handle the build process, but it can also clean up the project, and run unit tests and other automated tasks. The sample in Listing 9-4 shows a script that uses the `Prompt` task from the MSBuild Community Tasks project to get a response from the user after printing a series of options. Doing so makes the build script work interactively with the user.

Listing 9-4. *Sample Main Build Script (build.proj)*

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <NoWarn Condition="'$(NoWarn)'!=''>$(NoWarn),</NoWarn>
```



```

<NoWarn>$(NoWarn)MSB4078</NoWarn>
<Configuration Condition=" '$(Configuration)' == '' ">Release</Configuration>
<ApplicationName>Chapter09</ApplicationName>
<ApplicationVersion>v1.0</ApplicationVersion>

<Interactive Condition=" '$(Interactive)' == '' ">False</Interactive>

</PropertyGroup>
<Import Project="$(MSBuildExtensionsPath)\MSBuildCommunityTasks\
MSBuild.Community.Tasks.Targets" />
<Target Name="Clean">
  <Message Text="Running Clean Target..." Importance="high"></Message>
  <MSBuild Projects="ClassLibrary\ClassLibrary.csproj"
    Targets="Clean" ContinueOnError="false"></MSBuild>
</Target>
<Target Name="PreBuild">
  <Message Text="No PreBuild Tasks"></Message>
</Target>
<Target Name="Build">
  <Message Text="Running Build Target..." Importance="high"></Message>
  <MSBuild Projects="ClassLibrary\ClassLibrary.csproj"
    Targets="Build" ContinueOnError="false"></MSBuild>
</Target>
<Target Name="PostBuild">
  <Message Text="No PostBuild Tasks"></Message>
</Target>
<Target Name="RunTests" DependsOnTargets="Build">
  <Message Text="No Tests Tasks"></Message>
</Target>
<Target Name="Rebuild" DependsOnTargets="Clean;PreBuild;Build;PostBuild">
  <Message Text="Full Rebuild Successful!" Importance="high"></Message>
</Target>
<Target Name="FullBuild"
  DependsOnTargets="Clean;PreBuild;Build;PostBuild;RunTests">
  <Message Text="Full Build Successful!" Importance="high"></Message>
</Target>

<Target Name="PromptForTarget" Condition=" '$(Interactive)' == 'True' ">

  <Message Text=" "></Message>
  <Message Text="1) Clean" Importance="high"></Message>
  <Message Text="2) PreBuild" Importance="high"></Message>
  <Message Text="3) Build" Importance="high"></Message>
  <Message Text="4) PostBuild" Importance="high"></Message>
  <Message Text="5) RunTests" Importance="high"></Message>
  <Message Text="6) Rebuild" Importance="high"></Message>
  <Message Text="7) FullBuild" Importance="high"></Message>

```

```

    <Prompt Text="    Enter a target:">
      <Output TaskParameter="UserInput" PropertyName="SelectedTarget"/>
    </Prompt>

    <Message Text="Selected target is $(SelectedTarget)" Importance="high">
    </Message>

    <MSBuild Targets="Clean"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '1'"></MSBuild>
    <MSBuild Targets="PreBuild"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '2'"></MSBuild>
    <MSBuild Targets="Build"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '3'"></MSBuild>
    <MSBuild Targets="PostBuild"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '4'"></MSBuild>
    <MSBuild Targets="RunTests"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '5'"></MSBuild>
    <MSBuild Targets="Rebuild"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '6'"></MSBuild>
    <MSBuild Targets="FullBuild"
      Projects="build.proj" Condition="'$(SelectedTarget)' == '7'"></MSBuild>
  </Target>
</Project>

```

You cannot simply double-click an MSBuild script to run it, so it is helpful to create a script that can be clicked in Windows Explorer, such as `RunBuild.cmd` shown in Listing 9-5.

Listing 9-5. *RunBuild.cmd*

```

%windir%\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe build.proj ➡
/t:PromptForTarget /p:Configuration=Release;Interactive=True
Pause

```

The MSBuild utility takes several parameters. Specifically, the `/t` switch specifies a semi-colon-delimited list or targets, while the `/p` switch specifies a series of properties that the script will be able to use. The command in Listing 9-5 calls a target named `PromptForTarget` and sets the `Configuration` property to `True` and the `Interactive` property to `True`. You can get a full listing of command-line options for MSBuild by running `MSBuild \?` from the Visual Studio command prompt.

Deploying the Website

Moving a website from your development environment to a server can be done in many ways. The most prescribed way of doing so has been the `xcopy` command: a standard Windows utility for the command line that has many switches, giving you a great deal of control over the copying process. Listing 9-6 shows a simple command that copies all of the files from the source directory to the destination directory. If there are already files in the destination folder, it will only replace the destination file if the source file is more current.

Listing 9-6. Using *xcopy*

```
xcopy /E/D/Y "D:\SourceDir" "\\Server1\DestinationDir"
```

While using `xcopy` may get the job done some of the time, you may not always want to blindly copy files forward. The `xcopy` command does nothing to ensure that old files are removed with each deployment. It also does not adjust the configuration for the destination environment. To handle these requirements, you can make use of Web Deployment Projects.

Website Deployment Projects

Web applications created with the ASP.NET 2.0 website model do not use project files like class libraries. This new model made many new features possible, such as page-level compilation, but it also eliminated a few features we had with .NET 1.1 projects. Without a project file, you cannot run actions automatically before and after builds, and you also cannot manage references and resources in the same way. The `Web.config` file handles some details, but not everything. Managing ASP.NET websites inside of a solution helps manage dependencies and relationships with other projects in the solution, but ultimately you need a way to build a website and prepare it for deployment. This is where Web Deployment Projects come in.

Web Deployment Projects are an extension to Visual Studio that must be installed as an add-in. Once the add-in is in place, you can right-click a website in the Solution Explorer and click the option Create Web Deployment Project, which will create a new project that includes a single MSBuild script that Visual Studio can manage with a set of wizards.

WHERE TO GET WEB DEPLOYMENT PROJECTS

Web Deployment Projects can be found on the ASP.NET website (<http://www.asp.net>). Click the Downloads link at the top and look for the Web Deployment Projects section. That will take you to the page where you can download the installer.

Automating Configuration Changes

The most useful feature of Web Deployment Projects is the ability to modify the contents of the Web.config file based on the targeted configuration. This is done on a section-by-section basis. The project file is an MSBuild script that runs a sequence of tasks that leads it to read the settings you put in place. Figure 9-1 shows the dialog window for inserting these settings.

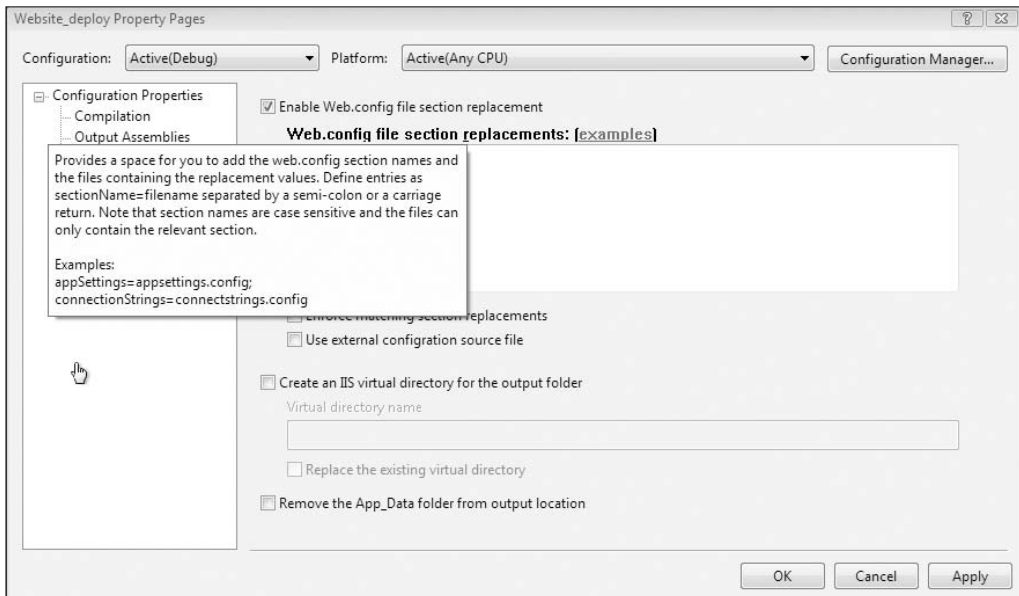


Figure 9-1. Configuration section replacements

The replacement section settings are unique for each configuration. A standard project in Visual Studio starts out with just the Debug and Release configurations. You can create a new one for each of your target environments, such as QA (Quality Assurance), Staging, and Production. These unique environments surely have different values for connection strings and perhaps other settings as well.

The sources for your replacement sections must exist under the website that is being modified. The original Web.config file will not be changed. The updated copy of the Web.config file is placed in the output directory along with all of the other files for the website. To create a new configuration, you can select the New option from within the Configuration Manager, as shown in Figure 9-2.

With all of your configurations defined, you can open the Property Pages for the Web Deployment Project. You can open the Property Pages from the context menu from the Web Deployment Project. The Deployment tab on the left side is where the configuration section replacement options are. For the Production configuration, you could create a folder called Config in the website and place a file named ConnectionStrings.Production.config in that folder. Then you would check the box in the Property Pages to enable configuration section replacements with the settings shown in Listing 9-7.

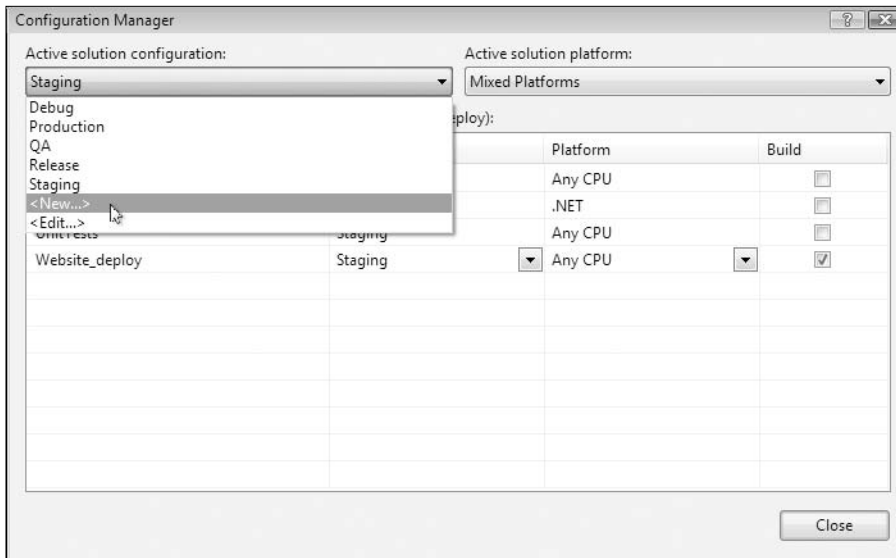


Figure 9-2. Creating a new configuration

Listing 9-7. Settings for Production Connection Strings

```
connectionStrings=Config\ConnectionStrings.Production.config;
```

You can place the configuration files anywhere you like in the website. Although you can use whatever extension you like on the file, it is best to use the standard .config extension because it is a protected extension: it is specifically blocked so that the web server will not serve it up to a user. The contents of the file holding the replacement section should just contain the single section, as is shown in Listing 9-8.

Listing 9-8. *ConnectionStrings.Production.config*

```
<?xml version="1.0"?>
<connectionStrings>
  <add name="chpt09" connectionString="Data Source=ProductionDB\SQL2005; ➡
Initial Catalog=Chapter09;Integrated Security=True"
  providerName="System.Data.SqlClient" />
</connectionStrings>
```

When the Web Deployment Project is compiled with the Production configuration, it will replace the connectionStrings section with the contents in Listing 9-8. When it is finished building, you can confirm that it has been replaced by looking in the Production folder under the directory for the Web Deployment Project. It will create a folder for each configuration just like a typical project.

Another useful option is the `configSource` attribute, which is a feature of all .NET application configuration files that allows you to refer to an external file for the contents of a section. This can be done automatically by checking the second option below the text box, as shown in Figure 9-3.

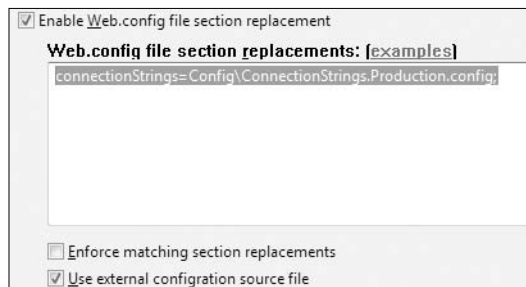


Figure 9-3. Using an external configuration source file

Having the connection string settings in a separate file outside of the `Web.config` file for a production environment could allow you to safely isolate the connection strings so that you can always drop a new `Web.config` file on top of an older copy with each deployment without overwriting the external file. Doing so could allow you to lock down the `Config` folder so that only a limited group of users can read and modify those files. You could give the user running the worker process for the website, typically Network Service, permission to read the configuration files while local administrators can modify those files.

PostBuild Deployments

A shortcoming of the Web Deployment Projects is the fact that you must build each time you want to create a new deployment with the adjusted configuration. Moving an application from QA to staging and finally to production requires building it once for each environment. It would be better to build it once, place the files in a drop location, and generate your environment-specific outputs from there. Fortunately, the task that handles the configuration section replacements can be used outside of Web Deployment Projects, skipping the build process, which only needs to be done once.

The task you want to use for your PostBuild deployments is called `ReplaceConfigSections`. It is a custom MSBuild task included with the assemblies for the Web Deployment Projects in an assembly called `Microsoft.WebDeployment.Tasks.dll`. You can create a custom MSBuild script and include this custom task with the `UsingTask` directive. A simple example of what this looks like is shown in Listing 9-9. The example looks very much like the contents of the project file for a Web Deployment Project.

Listing 9-9. PostBuild Configuration Section Replacements

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="ReplaceConfigSections"
    AssemblyFile="$(MSBuildExtensionsPath)\Microsoft\WebDeployment\v8.0\ ➔
```

```

Microsoft.WebDeployment.Tasks.dll"/>

  <PropertyGroup>
    <DeploymentDir>$(MSBuildProjectDirectory)\Website_deploy\$(Configuration) ➡
  </DeploymentDir>
    <WDTargetDir>$(DeploymentDir)</WDTargetDir>
    <ValidateWebConfigReplacement>false</ValidateWebConfigReplacement>
    <UseExternalWebConfigReplacementFile>false ➡
  </UseExternalWebConfigReplacementFile>
  </PropertyGroup>

  <ItemGroup Condition=" '$(Configuration)' == 'Release' ">
    <WebConfigReplacementFiles Include="Config\ConnectionStrings.Release.config">
      <Section>connectionStrings</Section>
    </WebConfigReplacementFiles>
  </ItemGroup>

  <Target Name="Build">
    <ReplaceConfigSections
      RootPath="$(WDTargetDir)"
      WebConfigReplacementFiles="@(\WebConfigReplacementFiles)"
      UseExternalWebConfigReplacementFile="$(UseExternalWebConfigReplacementFile)"
      ValidateSectionElements="$(ValidateWebConfigReplacement)"
    />
  </Target>
</Project>

```

The example in Listing 9-9 defines a replacement for the `connectionStrings` section for the Release configuration. The attributes for `RootPath`, `WebConfigReplacementsFiles`, `UseExternalWebConfigReplacementFile`, and `ValidateSectionElements` are all set by the `PropertyGroup` and `ItemGroup` elements. The `WebConfigReplacementsFiles` attribute is a collection of items. This example just defines a single section, but more could be added to that collection by simply duplicating the existing `ItemGroup` definition, such as is shown in Listing 9-10.

Listing 9-10. Multiple Replacement Sections

```

<ItemGroup Condition=" '$(Configuration)' == 'Release' ">
  <WebConfigReplacementFiles Include="Config\AppSettings.Release.config">
    <Section>appSettings</Section>
  </WebConfigReplacementFiles>
  <WebConfigReplacementFiles Include="Config\ConnectionStrings.Release.config">
    <Section>connectionStrings</Section>
  </WebConfigReplacementFiles>
  <WebConfigReplacementFiles Include="Config\Compilation.Release.config">
    <Section>compilation</Section>
  </WebConfigReplacementFiles>
</ItemGroup>

```

To manage the creation of these preconfigured deployment files, you can create a second MSBuild script called `deploy` that will handle these `PostBuild` tasks. You will want to leave the compilation work to the `build.proj` script while the deployment work is handled by another script called `deploy.proj`. When the output from a project is prepared, the deployment script will use the build script to ensure a Release build has been compiled. Then it will copy the output of that build to where the Web Deployment Project can adjust `Web.config` using the settings for the replacement sections. The full script in Listing 9-11 handles these tasks.

Listing 9-11. *deploy.proj*

```
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="ReplaceConfigSections"
    AssemblyFile="$(MSBuildExtensionsPath)\Microsoft\WebDeployment\v8.0\ ➡
      Microsoft.WebDeployment.Tasks.dll"/>
  <Import Project="$(MSBuildExtensionsPath)\MSBuildCommunityTasks\ ➡
    MSBuild.Community.Tasks.Targets" />

  <PropertyGroup>
    < Environment Condition=" '$( Environment)' == ' ' ">QA</ Environment >
    <Interactive Condition=" '$(Interactive)' == ' ' ">False</Interactive>

    <WebDeploymentDir>Website_deploy</WebDeploymentDir>

    <SourceDeploymentDir>$(WebDeploymentDir)\Release</SourceDeploymentDir>
    <DestinationDeploymentDir>$(MSBuildProjectDirectory)\Deployments\ ➡
      $(Environment)</DestinationDeploymentDir>

    <!-- Replacement Settings -->
    <WDTargetDir>$(WebDeploymentDir)\$(Environment)</WDTargetDir>
    <ValidateWebConfigReplacement>>false</ValidateWebConfigReplacement>
  </PropertyGroup>

  <PropertyGroup Condition=" '$( Environment)' == ' ' ">
    <UseExternalWebConfigReplacementFile>>false</UseExternalWebConfigReplacementFile>
  </PropertyGroup>

  <PropertyGroup Condition=" '$( Environment)' == 'QA' ">
    <UseExternalWebConfigReplacementFile>>false</UseExternalWebConfigReplacementFile>
  </PropertyGroup>

  <PropertyGroup Condition=" '$( Environment)' == 'Staging' ">
    <UseExternalWebConfigReplacementFile>>false</UseExternalWebConfigReplacementFile>
  </PropertyGroup>

  <PropertyGroup Condition=" '$( Environment)' == 'Production' ">
    <UseExternalWebConfigReplacementFile>>true</UseExternalWebConfigReplacementFile>
```



```

</PropertyGroup>

<ItemGroup Condition=" '$( Environment)' == 'QA' ">
  <WebConfigReplacementFiles Include="Config\ConnectionStrings.QA.config">
    <Section>connectionStrings</Section>
  </WebConfigReplacementFiles>
</ItemGroup>

<ItemGroup Condition=" '$( Environment)' == 'Staging' ">
  <WebConfigReplacementFiles Include="Config\ConnectionStrings.Staging.config">
    <Section>connectionStrings</Section>
  </WebConfigReplacementFiles>
</ItemGroup>

<ItemGroup Condition=" '$( Environment)' == 'Production' ">
  <WebConfigReplacementFiles Include="Config\ConnectionStrings.Production.config">
    <Section>connectionStrings</Section>
  </WebConfigReplacementFiles>
</ItemGroup>

<Target Name="Build" Condition=" !Exists('$(WebDeploymentDir)\Release') ">
  <MSBuild Projects="build.proj" Targets="Build"
    Properties="Configuration=Release"></MSBuild>
</Target>

<Target Name="InitDeployment" DependsOnTargets="Build">
  <Delete Files="$(DestinationDeploymentDir)\**\*.*"
    Condition=" Exists('$(DestinationDeploymentDir)') "></Delete>
  <MakeDir Directories="$(DestinationDeploymentDir)"
    Condition=" !Exists('$(DestinationDeploymentDir)') "></MakeDir>

  <CreateItem Include="$(SourceDeploymentDir)\**\*.*">
    <Output ItemName="WebsiteFiles" TaskParameter="Include"/>
  </CreateItem>

  <Copy SourceFiles="@(\WebsiteFiles)"
    DestinationFiles="@(\WebsiteFiles->'$(DestinationDeploymentDir)\
    %(RecursiveDir)%(Filename)%(Extension)') " SkipUnchangedFiles="true"></Copy>

</Target>

<Target Name="PrepDeployment" DependsOnTargets="InitDeployment">
  <ReplaceConfigSections
    RootPath="$(DestinationDeploymentDir)"
    WebConfigReplacementFiles="@(\WebConfigReplacementFiles)"
    UseExternalConfigSource="$(UseExternalWebConfigReplacementFile)"
    ValidateSectionElements="$(ValidateWebConfigReplacement)"
  >
</Target>

```

```

    />
    <Message Text="See Output: $(DestinationDeploymentDir)"></Message>
</Target>

<Target Name="PromptForTarget" Condition="'$(Interactive)' == 'True'">

    <Message Text="Select an Environment: "></Message>
    <Message Text="1) QA" Importance="high"></Message>
    <Message Text="2) Staging" Importance="high"></Message>
    <Message Text="3) Production" Importance="high"></Message>

    <Prompt Text="    Enter a target:">
        <Output TaskParameter="UserInput" PropertyName="SelectedTarget"/>
    </Prompt>

    <Message Text="Selected target is $(SelectedTarget)"></Message>

    <MSBuild Targets="PrepDeployment"
        Projects="deploy.proj"
        Properties="Environment=QA"
        Condition="'$(SelectedTarget)' == '1'"></MSBuild>
    <MSBuild Targets="PrepDeployment"
        Projects="deploy.proj"
        Properties="Environment=Staging"
        Condition="'$(SelectedTarget)' == '2'"></MSBuild>
    <MSBuild Targets="PrepDeployment"
        Projects="deploy.proj"
        Properties="Environment=Production"
        Condition="'$(SelectedTarget)' == '3'"></MSBuild>
</Target>

</Project>

```

The script in Listing 9-11 supports three environments: QA, staging, and production. It can be run with `RunDeploy.cmd`, which is shown in Listing 9-12.

Listing 9-12. *RunDeploy.cmd*

```

%windir%\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe deploy.proj ➤
/t:PromptForTarget /p:Environment=QA;Interactive=True
pause

```

When `RunDeploy.cmd` is double-clicked in Windows Explorer, it will run the script, which displays the three options for each of the environments. You can enter one of them, and it will carry out the tasks to prepare that environment. Each selection simply specifies a different value for the `Environment` property when calling the `PrepDeployment` target.

Before the `PrepDeployment` target is run it will run `InitDeployment`, which is set as a dependency with the `DependsOnTarget` attribute. The `InitDeployment` target copies the contents of the

prebuilt website to the destination directory for the selected environment. But before it runs, it also has a dependency on the Build target, which will only run if the prebuilt deployment directory does not exist. This prevents it from being built multiple times. The Build target calls the build.proj script, which is shown in Listing 9-13.

Listing 9-13. *build.proj*

```
<?xml version="1.0" encoding="utf-8"?>
<Project DefaultTargets="Build"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <NoWarn Condition="'$(NoWarn)'!=''>$(NoWarn),</NoWarn>
    <NoWarn>$(NoWarn)MSB4078</NoWarn>
    <Configuration Condition=" '$(Configuration)' == '' ">Release</Configuration>
    <ApplicationName>Chapter09</ApplicationName>
    <ApplicationVersion>v1.0</ApplicationVersion>

    <Interactive Condition="'$(Interactive)' == ''">False</Interactive>

  </PropertyGroup>
  <Import Project="$(MSBuildExtensionsPath)\MSBuildCommunityTasks\
MSBuild.Community.Tasks.Targets" />
  <Target Name="Clean">
    <Message Text="Running Clean Target..." Importance="high"></Message>
    <MSBuild Projects="ClassLibrary\ClassLibrary.csproj"
      Targets="Clean" ContinueOnError="false"></MSBuild>
    <MSBuild Projects="UnitTests\UnitTests.csproj"
      Targets="Clean" ContinueOnError="false"></MSBuild>
  </Target>
  <Target Name="PreBuild">
    <Message Text="No PreBuild Tasks"></Message>
  </Target>
  <Target Name="Build">
    <Message Text="Running Build Target..." Importance="high"></Message>
    <MSBuild Projects="ClassLibrary\ClassLibrary.csproj"
      Targets="Build" ContinueOnError="false"></MSBuild>
    <MSBuild Projects="Website_deploy\Website_deploy.wdproj"
      Targets="Build" ContinueOnError="false"></MSBuild>
    <MSBuild Projects="UnitTests\UnitTests.csproj"
      Targets="Build" ContinueOnError="false"></MSBuild>
  </Target>
  <Target Name="PostBuild">
    <Message Text="No PostBuild Tasks"></Message>
  </Target>
  <Target Name="RunTests" DependsOnTargets="Build">
    <Message Text="No Tests Tasks"></Message>
  </Target>
  <Target Name="Rebuild" DependsOnTargets="Clean;PreBuild;Build;PostBuild">
```

```

    <Message Text="Full Rebuild Successful!" Importance="high"></Message>
</Target>
<Target Name="FullBuild"
    DependsOnTargets="Clean;PreBuild;Build;PostBuild;RunTests">
    <Message Text="Full Build Successful!" Importance="high"></Message>
</Target>

<Target Name="PromptForTarget" Condition="'$(Interactive)' == 'True'>

    <Message Text=" "></Message>
    <Message Text="1) Clean" Importance="high"></Message>
    <Message Text="2) PreBuild" Importance="high"></Message>
    <Message Text="3) Build" Importance="high"></Message>
    <Message Text="4) PostBuild" Importance="high"></Message>
    <Message Text="5) RunTests" Importance="high"></Message>
    <Message Text="6) Rebuild" Importance="high"></Message>
    <Message Text="7) FullBuild" Importance="high"></Message>

    <Prompt Text="    Enter a target:">
        <Output TaskParameter="UserInput" PropertyName="SelectedTarget"/>
    </Prompt>

    <Message Text="Selected target is $(SelectedTarget)"></Message>

    <MSBuild Targets="Clean" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '1'"></MSBuild>
    <MSBuild Targets="PreBuild" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '2'"></MSBuild>
    <MSBuild Targets="Build" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '3'"></MSBuild>
    <MSBuild Targets="PostBuild" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '4'"></MSBuild>
    <MSBuild Targets="RunTests" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '5'"></MSBuild>
    <MSBuild Targets="Rebuild" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '6'"></MSBuild>
    <MSBuild Targets="FullBuild" Projects="build.proj"
        Condition="'$(SelectedTarget)' == '7'"></MSBuild>
</Target>
</Project>

```

The build.proj script also has a menu that is displayed when the RunBuild.cmd script is used, which is shown Listing 9-14.

Listing 9-14. RunBuild.cmd

```

%windir%\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe build.proj ➡
/t:PromptForTarget /p:Configuration=Release;Interactive=True
pause

```

The seven options displayed by the menu are the typical build options you would expect. When the `deploy.proj` calls the main build script, the configuration is set to Release, which will cause the Build target to run the build for the Web Deployment Project to generate the output to the Release folder. The Release folder is then used by the `deploy.proj` script to prepare the selected environment.

Finally, the contents of the prepared output directory can be wrapped up into a zip file, which can be moved to the destination environments more easily than a large set of files. This can be done with the Zip task, which is a part of the MSBuild Community Tasks. First you need to define the name of the zip file in the default PropertyGroup, as shown in Listing 9-15.

Listing 9-15. *ZipFilename Defined*

```
<ZipFilename>$(MSBuildProjectDirectory)\Deployments\$(Configuration).zip ➡
</ZipFilename>
```

Then the Zip task can be added to the end of the PrepEnvironment target. The Zip task declaration is shown in Listing 9-16.

Listing 9-16. *Zip Declaration*

```
<CreateItem Include="$(DestinationDeploymentDir)\**\*.*)"
  <Output ItemName="ZipFiles" TaskParameter="Include"/>
</CreateItem>

<Zip Files="@ (ZipFiles)"
  ZipFileName="$(ZipFilename)"
  WorkingDirectory="$(DestinationDeploymentDir)" />
```

The result is a zip file containing all of the files for the website with the modified configuration file. If the web server is located at a remote hosting facility, you can upload the zip file to the server and unzip it in place without any additional work needed.

COMMON FOLDER ADDITIONS

The MSBuild scripts in this chapter are very reusable in your projects. Using them will cut down on the manual work you do with each project and will capture the details that you must know to build a project. These scripts can be placed in your Common folder in the Scripts subfolder to be referenced in your projects (D:\Projects\Common\Scripts\MSBuild).

Deploying the Database

As each release of your web application is pushed to the various environments, there may be updates to the database that are necessary for the application to work properly. Perhaps an additional column was added to a table or a new stored procedure was defined. If the updated website calls that stored procedure, the database must be updated before the new application version is put into use. All you may need to do to bring the database structure up to the

are going to write your own statements manually. Another adjustment you may make to a table is to add an index. The table updated in Listing 9-17 has a column named Name. It may speed up queries to this table if this column had an index.

An index can be added visually as well with Management Studio using the Manage Indexes and Keys button while a table is being edited, as shown in Figure 9-5.

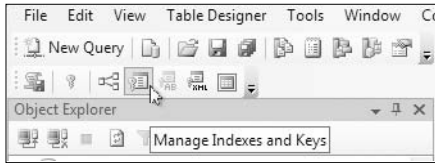


Figure 9-5. *Manage Indexes and Keys button*

Once the index has been added, you can generate the change script and copy the part that creates the index and use it in your custom script. The result would look like Listing 9-18.

Listing 9-18. *Index Creation Script*

```
BEGIN TRANSACTION
GO
CREATE NONCLUSTERED INDEX IX_chpt09_Names ON dbo.chpt09_Names
(
    Name
) WITH(
    STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
    ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
GO
COMMIT
```

Here again a large amount of data will cause the creation of the index to take a while. As you work through a release cycle, you may add change scripts to your database project that you will run as a part of the next deployment. These update scripts can be placed in the database project for the solution in a folder called *Change Scripts*. At the start of each release, you may want to incorporate the changes from those scripts in the creation scripts for the database and start with a new set of update scripts as you have new changes. Doing so will cut down on the complexity of preparing a new environment from scratch. How you manage the update scripts will depend on how frequently you push out releases and the level of change each release requires.

USING SCRIPT TEMPLATES

SQL Server Management Studio has a collection of templates that you can use to create scripts to modify an existing database and to carry out various database-related tasks. You can view these templates by opening the Template Explorer. Double-clicking a template will open it in a new query window. To replace all of the parameters in the template with actual values, you can click *Specify Values for Templates Parameters* from the Query menu.

Automating Database Updates

The change scripts can be manually run directly against the target database if you have access to it. Either you will have direct access from your computer or you will upload the scripts to the database server, connect to the server via Remote Desktop, and use a locally installed copy of Management Studio to run the scripts. For a limited set of changes for updates that are infrequent, you may find this manual process is sufficient. But if there are many changes and you are not fully aware of all of them, you may want the changes controlled by an automated process that has been carefully crafted and run on the QA and staging environments without failure. Repeating that controlled and proven process in the production environment will ensure a more reliable deployment.

Using MSBuild and ExecuteDDL

The ExecuteDDL task, which is included with the MSBuild Community Tasks, can be used to run scripts to carry out the database updates. As scripts are created, they can be referenced from an MSBuild script that will be used in each environment to update the databases. Listing 9-19 shows the contents of *db.proj*, which is the MSBuild script that runs two update scripts for four environments: development, QA, staging, and production.

Listing 9-19. *db.proj*

```
<Project DefaultTargets="RunScripts"
  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="$(MSBuildExtensionsPath)\MSBuildCommunityTasks\
    MSBuild.Community.Tasks.Targets" />

  <PropertyGroup>
    <Environment Condition="'$(Environment)' == '' ">Development</Environment>
    <Interactive Condition="'$(Interactive)' == ''">False</Interactive>
  </PropertyGroup>

  <PropertyGroup Condition="'$(Environment)' == 'Development' ">
    <ConnectionString>
      Data Source=.\SQLEXPRESS;Initial Catalog=Chapter09;Integrated Security=True
    </ConnectionString>
  </PropertyGroup>

  <PropertyGroup Condition="'$(Environment)' == 'QA' ">
    <ConnectionString>
      Data Source=QADB\SQL2005;Initial Catalog=Chapter09;Integrated Security=True
    </ConnectionString>
  </PropertyGroup>

  <PropertyGroup Condition="'$(Environment)' == 'Staging' ">
    <ConnectionString>
      Data Source=StagingDB\SQL2005;Initial Catalog=Chapter09;
      Integrated Security=True
```



```

    </ConnectionString>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Environment)' == 'Production' ">
    <ConnectionString>
      Data Source=ProductionDB\SQL2005;Initial Catalog=Chapter09; ➡
      Integrated Security=True
    </ConnectionString>
  </PropertyGroup>

  <Target Name="RunScripts">
    <ExecuteDDL Files="Database\Change Scripts\Update1.sql"
      ConnectionString="$(ConnectionString)"
      Condition=" Exists('Database\Change Scripts\Update1.sql') ">
    </ExecuteDDL>
    <ExecuteDDL Files="Database\Change Scripts\Update2.sql"
      ConnectionString="$(ConnectionString)"
      Condition=" Exists('Database\Change Scripts\Update2.sql') ">
    </ExecuteDDL>

  </Target>

  <Target Name="PromptForTarget" Condition=" '$(Interactive)' == 'True' ">

    <Message Text="Select an Environment: "></Message>
    <Message Text="1) Development" Importance="high"></Message>
    <Message Text="2) QA" Importance="high"></Message>
    <Message Text="3) Staging" Importance="high"></Message>
    <Message Text="4) Production" Importance="high"></Message>

    <Prompt Text="    Enter a target:">
      <Output TaskParameter="UserInput" PropertyName="SelectedTarget"/>
    </Prompt>

    <Message Text="Selected target is $(SelectedTarget)"></Message>

    <MSBuild Targets="RunScripts"
      Projects="db.proj"
      Properties="Environment=Development"
      Condition=" '$(SelectedTarget)' == '1' "></MSBuild>
    <MSBuild Targets="RunScripts"
      Projects="db.proj"
      Properties="Environment=QA"
      Condition=" '$(SelectedTarget)' == '2' "></MSBuild>
    <MSBuild Targets="RunScripts"
      Projects="db.proj"
      Properties="Environment=Staging"

```

```

        Condition="'$(SelectedTarget)' == '3'"></MSBuild>
    <MSBuild Targets="RunScripts"
        Projects="db.proj"
        Properties="Environment=Production"
        Condition="'$(SelectedTarget)' == '4'"></MSBuild>
</Target>

</Project>

```

To run the `db.proj` script, `RunDb.cmd`, which is shown in Listing 9-20, can be used.

Listing 9-20. *RunDb.cmd*

```

%windir%\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe db.proj
/t:PromptForTarget /p:Configuration=Release;Interactive=True
Pause

```

When the script is run, it will present a menu displaying the choices for the four supported environments. The selection controls the value for the `ConnectionString` property, which is used by the `ExecuteDDL` task. The successful result of running this MSBuild script is shown in Figure 9-6.

```

C:\Windows\system32\cmd.exe
1) Development
2) QA
3) Staging
4) Production
Enter a target:
1
Selected target is 1

Project "D:\Projects\Chapter 09\db.proj" is building "D:\Projects\Chapter 09
\db.proj" (RunScripts target(s)):

Target RunScripts:
    Executing DDL file 'Database\Change Scripts\Update1.sql'
        Successfully executed SQL command with result = -3
    Executing DDL file 'Database\Change Scripts\Update2.sql'
        Successfully executed SQL command with result = -3

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:03.42

D:\Projects\Chapter 09>pause
Press any key to continue . . .

```

Figure 9-6. Successfully running `db.proj`

Using Embedded Scripts

Database updates can be integrated directly within your application by embedding scripts right in the assembly with the rest of your data access layer. Initially when the application is used, the database can be initialized, and future releases of the application can update the existing database schema with the embedded scripts. This will all be done automatically and transparently. Upgrading a website can be as simple as copying the new assemblies into place, which automatically restarts the worker process. The database updates will be applied automatically.

Such functionality is not an automatic feature of ASP.NET but can be implemented with the building blocks that are available in the application. For the following examples, you will

create a table to hold names. The initialization script will create a schema versions table that is used to keep track of the database schema. Then there will be a series of update scripts. Additional update scripts can be added with each release.

Initializing the Database

To get started, you will create the schema versions table as you do normally with a database project. This table holds a name and an integer for the version. As each update is executed successfully, the version is set to the next version. Listing 9-21 shows the creation script for the schema versions table.

Listing 9-21. *chpt09_SchemaVersions.sql*

```
IF EXISTS (
    SELECT * FROM sysobjects WHERE type = 'U' AND name = 'chpt09_SchemaVersions')
BEGIN
    DROP Table chpt09_SchemaVersions
END
GO

CREATE TABLE [dbo].[chpt09_SchemaVersions](
    [Name] [nvarchar](20) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Version] [smallint] NOT NULL,
    CONSTRAINT [PK_chpt09_SchemaVersions] PRIMARY KEY CLUSTERED
    (
        [Name] ASC
    )WITH (PAD_INDEX = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]

GO
```

The Name column is the primary key. For the embedded scripts, there is going to just be a single schema named names. This structure can support any number of other schemas that can all be versioned independently. Breaking up all the data held in the database into distinct named schemas will help isolate the updates, which should simplify the update scripts.

The schema table uses two stored procedures to get and set the schema version. Listing 9-22 shows the script to set the version of a named schema.

Listing 9-22. *chpt09_SetSchemaVersion.sql*

```
IF EXISTS (
    SELECT * FROM sysobjects WHERE type = 'P' AND name = 'chpt09_SetSchemaVersion')
BEGIN
    DROP Procedure chpt09_SetSchemaVersion
END
GO

CREATE Procedure dbo.chpt09_SetSchemaVersion
```

```

(
    @Name nvarchar(20),
    @Version smallint
)
AS

IF NOT EXISTS (
    SELECT * FROM chpt09_SchemaVersions
    WHERE Name = @Name
)
BEGIN
    INSERT into chpt09_SchemaVersions
    ([Name],Version)
    values (@Name, @Version)
END
ELSE
BEGIN
    UPDATE chpt09_SchemaVersions
    SET Version = @Version
    WHERE [Name] = @Name
END

GO

GRANT EXEC ON chpt09_SetSchemaVersion TO PUBLIC
GO

```

Setting the version is a simple process of either inserting or updating the schema versions table. Getting the schema version requires a similar process. The script to get the named schema version is shown in Listing 9-23.

Listing 9-23. *chpt09_GetSchemaVersion.sql*

```

IF EXISTS (
    SELECT * FROM sysobjects WHERE type = 'P' AND name = 'chpt09_GetSchemaVersion')
BEGIN
    DROP Procedure chpt09_GetSchemaVersion
END

GO

CREATE Procedure dbo.chpt09_GetSchemaVersion
(
    @Name nvarchar(20),
    @Version smallint OUTPUT
)
AS
IF EXISTS (

```

```

SELECT * FROM chpt09_SchemaVersions
WHERE Name = @Name
)
BEGIN
    SET @Version = (
        SELECT Version
        FROM chpt09_SchemaVersions
        WHERE Name = @Name)
END
ELSE
BEGIN
    SET @Version = 0
END

GO

GRANT EXEC ON chpt09_GetSchemaVersion TO PUBLIC
GO

```

The version of the named schema is returned as an output parameter. If the named schema does not exist, the version is defaulted to 0; otherwise the actual value is returned.

Updating the Database

With the schema table and stored procedures in place, you can copy the scripts into the class library as embedded scripts. Simply create a folder in the class library named `Scripts`. To add a new SQL script, you will need to add it as a simple text file, but you can set the extension as `.sql`. When you open the file, it will be treated as a SQL script. For this first script, you will want to create the schemas table and two stored procedures used to get and set the version numbers. The contents of the three scripts to define these resources will be placed in order in a script called `init.sql`. The first script is the table script, and it will be modified to drop off the first part, which checks for the existence of the schema table. You want to take an extra precaution here. If the table does exist and somehow this initialization script is run, you want it to fail right away when it tries to create this first table.

To ensure this script is included within the assembly, you will set the `Build Action` to `Embedded Resource` using the Properties panel shown in Figure 9-7. This setting makes it available as a resource stream on the assembly. (You will load the script as a resource stream later.)

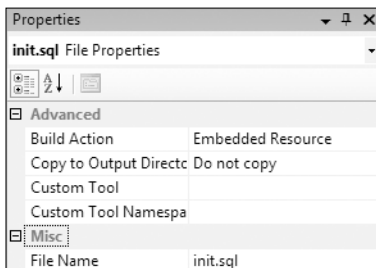


Figure 9-7. *init.sql as an embedded resource*

Next you will create a few sample scripts that will run as updates to test the functionality of the update routine. These scripts will be placed in the Scripts folder in the class library and set as embedded resources. They will be named `NamesUpdate.00.sql`, `NamesUpdate.01.sql`, and `NamesUpdate.02.sql`. The first script defines the names table and stored procedure. The second script adds three names, and the third script adds one more name. These scripts could easily adjust table columns, add or remove indexes, or whatever you choose to have them do.

There are two important details about these embedded scripts. First the commands are all separated with a `GO` statement after each command. This is done so the commands can be run independently as SQL commands with the Data Access Block in the Enterprise Library as you have done with stored procedures in previous chapters. Second, the update scripts should always set the version of the database to the next number. This is done at the end after all of the other commands have run successfully. However, the order is not strictly necessary as these commands will be grouped together as a transaction, so if any of the commands fail, all changes in the update script will be undone. Listing 9-24 shows `NamesUpdate.01.sql`.

Listing 9-24. *NamesUpdate-01.sql*

```
-- update 1 (add stooges to names table)

EXEC chpt09_SaveName 'Larry'
GO

EXEC chpt09_SaveName 'Moe'
GO

EXEC chpt09_SaveName 'Curly'
GO

-- be sure to update the names schema version
EXEC chpt09_SetSchemaVersion 'names', 2
GO
```

Each name is added using the `chpt09_SaveName` stored procedure created by the first update script, `NamesUpdate.00.sql`, which is not shown here (see the code downloads). You can see how each command ends with a `GO` statement with the last statement setting the schema version for the names schema to 2.

Running the Scripts

The scripts embedded in the assembly are now ready to be loaded and run using a class called `DatabaseManager`. This class is in the same class library as the update scripts, which has the default namespace set to `Chapter09`—an important detail when loading the scripts as resource streams. The folder structure is also important. It should look like Figure 9-8.

To reference the scripts, you will use a prefix to access them much like a namespace. This prefix is defined as a constant in the `DatabaseManager` class, shown in Listing 9-25.

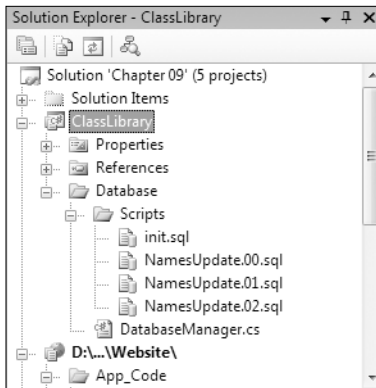


Figure 9-8. *Class library structure*

Listing 9-25. *ScriptsPrefix Constant*

```
private const string ScriptsPrefix = "Chapter09.Database.Scripts.";
```

The scripts prefix uses the default namespace and the folder structure to define the path to the embedded scripts. You are now ready to construct the initialization and update routine, which starts with the `InitializeDatabase` method shown in Listing 9-26.

Listing 9-26. *InitializeDatabase Method*

```
public void InitializeDatabase()
{
    if (IsAutoUpdatesEnabled())
    {
        bool success = true;
        if (!IsInitialized())
        {
            List<string> initCommands = GetSqlCommands(ScriptsPrefix + "init.sql");
            success = RunSqlCommands(initCommands);
        }
        if (success)
        {
            UpdateDatabase();
        }
    }
}
```

The `InitializeDatabase` method first checks whether the automatic updates feature is enabled. This is based on a configuration setting named `EnableAutoUpdates`. The value must be set to `true` in order for the initialization process to run. Next, the commands for the script named `init.sql` are loaded and run. If the script is successful, meaning no exceptions were thrown, it continues on to run the `UpdateDatabase` method shown in Listing 9-27.

Listing 9-27. *UpdateDatabase Method*

```

private void UpdateDatabase()
{
    int version = GetSchemaVersion("names");

    if (version == 0)
    {
        List<string> commands =
            GetSqlCommands(ScriptsPrefix + "NamesUpdate.00.sql");
        if (RunSqlCommands(commands))
        {
            version = GetSchemaVersion("names");
        }
    }

    if (version == 1)
    {
        List<string> commands =
            GetSqlCommands(ScriptsPrefix + "NamesUpdate.01.sql");
        if (RunSqlCommands(commands))
        {
            version = GetSchemaVersion("names");
        }
    }

    if (version == 2)
    {
        List<string> commands =
            GetSqlCommands(ScriptsPrefix + "NamesUpdate.02.sql");
        if (RunSqlCommands(commands))
        {
            version = GetSchemaVersion("names");
        }
    }
}

```

The `UpdateDatabase` method gets the initial version of the names schema. If it is not defined, it will default to 0, which will cause the first script to be run. Once the first script has completed, the version should be reset to 1, and the next script will run, which follows the same pattern. If this was the first run for this update process, all three of the update scripts will run. You can see that adding another block to the end of this method that references a new update script will be an easy addition.

There are a couple of private methods, `GetSqlCommands` and `RunSqlCommands`, that make this update process work. The first method is shown in Listing 9-28.

Listing 9-28. GetSqlCommands Method

```
public List<string> GetSqlCommands(string scriptName)
{
    List<string> commands = new List<string>();
    Type type = GetType();

    Stream stream = type.Assembly.GetManifestResourceStream(scriptName);
    if (stream != null)
    {
        StringBuilder sb = new StringBuilder();
        StreamReader sr = new StreamReader(stream);
        string line = null;

        while (sr.Peek() >= 0)
        {
            line = sr.ReadLine();
            if (!CommandDelimiter.Equals(line))
            {
                sb.AppendLine(line);
            }
            else {
                commands.Add(sb.ToString());
                sb = new StringBuilder();
            }
        }
        if (!String.IsNullOrEmpty(sb.ToString()))
        {
            commands.Add(sb.ToString());
        }
    }

    return commands;
}
```

The `GetSqlCommands` method uses the `GetManifestResourceStream` method on the `Assembly` class to load the resource stream. It then uses a `StreamReader` to iterate over each line looking for the command delimiter, which is defined as a constant named `CommandDelimiter`. It is set to the standard `GO` statement. Each line that does not match the delimiter is added to the `StringBuilder` variable. And when a delimiter line is reached, the `StringBuilder` value is added to a collection of commands, which is eventually passed to the `RunSqlCommands` method shown in Listing 9-29.

Listing 9-29. RunSqlCommands Method

```
private bool RunSqlCommands(List<string> commands)
{
    bool success = true;
```

```
using (DbConnection connection = db.CreateConnection())
{
    connection.Open();
    DbTransaction transaction = connection.BeginTransaction();

    try
    {
        foreach (string command in commands)
        {
            using (DbCommand dbCmd = db.GetSqlStringCommand(command))
            {
                db.ExecuteNonQuery(dbCmd, transaction);
            }
        }
        transaction.Commit();
    }
    catch (Exception ex)
    {
        success = false;
        Trace.WriteLine(ex.Message);
        // Rollback transaction
        transaction.Rollback();
    }
    connection.Close();
}
return success;
}
```

In the `RunSqlCommands` method, the commands passed in are run individually inside of a transaction. If any command fails, the entire transaction will be rolled back, and the return value that indicates whether or not the commands executed successfully will be set to false.

Tip While developing the embedded scripts to update the database, you can leverage the fact that the commands are run in a transaction to prevent the script from being committed. As you add statements to the update script, you can place an invalid statement at the very end to ensure the changes will be rolled back; this way, you can continue working on the script without having to restore the database to the state it was in before you executed the updates. Once you have completed the update script to your satisfaction, you can remove the intentional error.

The final step to ensuring the database updates are put in place each time the application is updated is to call the `InitializeDatabase` method each time the application starts. For a website, this can be done using the `Application_Start` method in the `Global.asax` file (see Listing 9-30).

Listing 9-30. *Application_Start Calling InitializeDatabase*

```

void Application_Start(object sender, EventArgs e)
{
    DatabaseManager dbm = new DatabaseManager();
    dbm.InitializeDatabase();
}

```

Now each time the website is updated, it will run the `InitializeDatabase` method, which will ensure your database schema is up to date.

Custom Configuration Sections

In previous chapters, you have hard-coded the connection string name in the data access layer. If you continue to hard-code this value in your data access layer and build modular components that work with different sets of data, you may hard-code the same value in each component so that you do not have to maintain the connection string configuration for each of those components. But there is a problem with assuming you will only have a single database connection. It forces you to place all of your data in the same database.

Alternatively, you may use a unique connection string name for each component, which will require you to maintain a connection string for each component. This approach gives you some flexibility but duplicates several connection strings in your configuration. Another alternative is to use application settings for each component that indicate the connection string name to use. The class library created earlier in this chapter, which works with a names database, could use an application setting named `NamesDatabase`, which matches the name of the connection string to use. This would be done for each component. This may be a reasonable approach.

The last approach is to create a custom configuration, which allows you to set the values as you want them in a structure you choose. This is done by leveraging the existing `ConfigurationSectionGroup` and `ConfigurationSection` classes. In Chapter 5, you created providers that used a custom `ConfigurationSection`. Through the provider base classes, some of the extra work was done for you. Here you will build a custom configuration, read it all in manually, and use it in the `DatabaseManager` class created in the previous sections.

What you will build is a few classes that provide access to the connection string name you will use for the `DatabaseManager`. One class will represent the configuration group while another will represent the section. The third class will be a utility to load the configuration programmatically.

The first class is `Chapter09SectionGroup`, which inherits from `ConfigurationSectionGroup` and defines a single property. This class is shown in Listing 9-31.

Listing 9-31. *Chapter09SectionGroup*

```

using System.Configuration;

namespace Chapter09.Configuration
{
    public class Chapter09SectionGroup : ConfigurationSectionGroup
    {
        [ConfigurationProperty("chapter09Group")]

```

```

        public Chapter09Section Chapter09Section
        {
            get
            {
                return (Chapter09Section)Sections["chapter09"];
            }
        }
    }
}

```

The property named `Chapter09Section` returns a class by the same name by accessing the `Sections` property, which is inherited from the base class. `Chapter09Section` is shown in Listing 9-32.

Listing 9-32. *Chapter09Section*

```

using System.Configuration;

namespace Chapter09.Configuration
{
    public class Chapter09Section : ConfigurationSection
    {
        [ConfigurationProperty("connectionStringName",
            DefaultValue = "chapter09", IsRequired = true),
            StringValidator(MinLength = 1, MaxLength = 50)]
        public string ConnectionStringName
        {
            get
            {
                return (string) this["connectionStringName"];
            }
            set
            {
                this["connectionStringName"] = value;
            }
        }

        [ConfigurationProperty("enableAutoUpdates",
            DefaultValue = "True", IsRequired = false)]
        public bool EnableAutoUpdates
        {
            get
            {
                bool enabled;
                bool.TryParse(this["enableAutoUpdates"].ToString(), out enabled);
                return enabled;
            }
            set
            {

```

```

        this["enableAutoUpdates"] = value;
    }
}
}
}

```

The `Chapter09Section` class accesses the `connectionStringName` and `enableAutoUpdates` attributes from the configuration. The properties that access these values are decorated with attributes that control how the property behaves. The `EnableAutoUpdates` property is not required, which allows the value to not be specified in the configuration. When it is not, the default value is used. The `ConnectionStringName` property is required and will cause an exception if the property is accessed while the attribute is not defined in the configuration.

To load this configuration, you will use a utility class named `Chapter09Configuration`, which is shown in Listing 9-33.

Listing 9-33. *Chapter09Configuration*

```

using System.Configuration;
using System.Web;
using System.Web.Configuration;

namespace Chapter09.Configuration
{
    public class Chapter09Configuration
    {
        public static Chapter09SectionGroup GetConfig()
        {
            System.Configuration.Configuration config;
            HttpContext context = HttpContext.Current;
            if (context != null)
            {
                string path = "~/";
                config = WebConfigurationManager.OpenWebConfiguration(path);
            }
            else
            {
                config = ConfigurationManager.OpenExeConfiguration ➤
(ConfigurationUserLevel.None);
            }
            Chapter09SectionGroup chapter09Config =
                (Chapter09SectionGroup)config.SectionGroups["chapter09Group"];
            return chapter09Config;
        }
    }
}

```

The `Chapter09Configuration` class has a single static method that returns the `Chapter09SectionGroup` class from the `GetConfig` method. This method will work for a

website and other applications such as console and desktop applications. As you can see, it uses the `WebConfigurationManager` or the `ConfigurationManager` based on the value from the `HttpContext`, which is defined when the method is run within a website. Once the configuration is loaded, it uses the `SectionGroups` collection to return the group named `chapter09Group`, which is lined up with the `Chapter09SectionGroup` class. In order to make these classes line up with the names, it is necessary to configure the custom sections in the configuration file. This definition is shown in Listing 9-34.

Listing 9-34. *Custom Section Definition*

```
<configSections>
  <sectionGroup name="chapter09Group"
    type="Chapter09.Configuration.Chapter09SectionGroup, Chapter09">
    <section name="chapter09"
      type="Chapter09.Configuration.Chapter09Section, Chapter09"/>
    </sectionGroup>
</configSections>
```

The name and the type are defined by the custom section definition with the section contained within the group. The rest of the custom configuration is shown in Listing 9-35.

Listing 9-35. *Custom Configuration*

```
<chapter09Group>
  <chapter09 connectionStringName="chpt09" enableAutoUpdates="True"/>
</chapter09Group>
```

Because the `enableAutoUpdates` attribute is optional, you can leave it off if you want to use the default value, which is set to `True`. Now you can configure as many connection strings as you like and point to the one you choose with this custom configuration section while enabling and disabling this automatic update feature.

A potential enhancement for the database update process is to define a separate connection string name for the update process, which is separate from the rest of the data access layer. The update scripts will need privileges that a normal web application does not need, so instead of escalating the rights for the entire application, you can define a special connection with the necessary permissions. This will improve the security of your application. To improve it further, you can enable the user account for this special connection just during the deployment process and disable it once you have completed. Doing so will ensure it is not possible to run the updates and block access to the account with the increased privileges.

Summary

In this chapter, I covered deploying websites and databases, and how to make use of custom configurations. By streamlining the deployment and configuration processes, you make it possible to push out more updates for your applications with a lower risk for each deployment because human error has been minimized by the scripts. Being able to push out timely updates will be very helpful to adapt to the performance needs of your production environment. The automation tasks covered in this chapter will also free up more of your time to spend on performance improvements instead of the busy work of pushing around files.



A Sample Application

Now you can put together several of the pieces covered throughout this book to create a flexible and high-performance application. Your goal will be to build a modular application that starts with a clean data access layer and has each business object clearly defined in terms of its data contained within each domain as well as the relationships between the objects.

This chapter covers the following:

- Understanding performance and scalability
- Creating the database
- Creating data access providers
- Managing relationships
- Using custom configuration
- Implementing a LINQ provider
- Implementing a WCF provider

In all of the previous chapters, the implicit rule for creating a high-performance application has been that good design and flexibility are more important than prematurely optimizing your application. Until you have a finished product that is put under an actual load, you will not really know where you will experience performance problems. And load can change over time in unexpected ways. If you start to optimize for performance too soon in the development cycle, you may lock yourself into an approach that just does not scale.

Understanding Performance and Scalability

It is important to distinguish between performance and scalability. The two are related but not one in the same. As your website becomes more heavily used, it will need to handle more and more requests, and it will need to do so without the cost of each request increasing. If you start out with 1,000 users, and each request is handled in under a second, you will want that level of service to be maintained when you reach 100,000 users. If you start out with a website that already has great performance, you may never have a scaling problem. But there could come a point when the cost per request does start to increase, and you will need to apply the approaches in this chapter to get you back to the level of service that kept those initial 1,000 users happy.

In Chapter 6, I mentioned how you can often be your own worst enemy by hitting the database so often that you essentially cause a traffic jam of disk reads against the database.

The frequent reads cause a traffic jam on the network as well, which doubles and triples the time to process a request. Such a high-load scenario causes the cost of each request to increase dramatically. For a computer, a second can be an eternity. And when many of the requests that come into your website are processed in well under a single second during normal load, you can imagine that setting a caching time-out to five seconds would affect the scalability of your application, even if a user may not see any change in performance. You eliminate calls to the database while also saving clock cycles on your processor. Setting a caching time-out is a great first step that will take you a long way.

A typical website can get anywhere between 5,000 and 100,000 page requests in a day. With 86,400 seconds in a 24-hour period, you can see in Table 10-1 how page requests can generally be spread apart.

Table 10-1. *Time Between Requests*

Requests per Day	Time Between Requests
5,000	16.78 seconds
100,000	0.86 seconds
500,000	0.17 seconds

Table 10-1 is an overly simplistic representation of website traffic, but it illustrates the difference between performance and scalability. When a website is getting only 5,000 hits in a day, it is not important to address scalability, especially when a single request takes less than a second. If a website gets 100,000 hits in a day, the average time between requests drops below one second, but the time between requests is still over half a second. You still have a little time between requests, so you will not have concurrent requests. The difference between 100,000 hits and 5,000 hits represents a huge window, and I assume most sites, especially intranet web applications, fall into this range. Now, when a website reaches 500,000 hits or more, scalability becomes a major issue. When traffic to your web application reaches this frequency, you will want to explore your options to spread the load somehow.

Concurrent Requests

The intersection where performance meets scalability occurs when the number of concurrent requests starts to increase the cost per request. If ten requests are currently being processed, and each of them locks a commonly used table in the database—even for just a moment—the requests may be delayed and their cost increased. But if each request can access the database efficiently, without interfering with the other requests, you will have the scalability that you want.

A frequently requested page on your website may not explicitly be locking a table. It may instead be indirectly locking an index during an update that you did not anticipate. We typically expect indexes to improve performance, but sometimes they can have the opposite effect. When you come across concurrency issues, you may be able to consult your DBA to determine what resource is being locked. If an index is involved, you may need to use a different type of index or drop the index entirely so that your concurrency problems go away. You may also be able leave the index in place and batch the updates to it so they are not happening so frequently.

Note More and more servers, and even personal computers, are coming out with multiple processors. It is not uncommon to purchase a new server with dual or quad core processors that can handle a great number of concurrent requests. These great new capabilities should be exploited as much as possible to the benefit of your application and your users. You also want to make sure you are not wasting processing power with unnecessary tasks by offloading ancillary work to other servers whenever the work can be handled out of process, such as processing an order after it has been submitted.

Bottlenecks

If you focus your efforts on the usual suspects when troubleshooting bottlenecks, you may be overlooking other areas that are the real bottlenecks, the true gatekeepers to scalability. Years ago a report that came out compared the scalability of the Apache web server on Linux to IIS on Windows NT. It showed that IIS could handle many more requests under high load. Many developers questioned the accuracy of the report because at the time the Apache web server was the most popular and most efficient web server around, and IIS was seen as a new kid on the block that had been plagued with various problems. It turns out that the bottleneck had little to do with the web server. The source of Apache's problem was that Linux internals used a locking mechanism allowing only a limited number of concurrent network connections into the Apache web server, while Windows NT used a threaded model allowing many more concurrent requests into IIS. So no matter how you configured the Apache web server, there was no way you could possibly make it faster than IIS. It simply was not the weakest link.

In the same way, you may find that your website is not the weakest link in an application. It could be the database or perhaps even the network. The quality of the networking equipment may be to blame or the quality of the network cables between the servers. I once had a computer that used a network card that I bought for \$15. I could have purchased the \$100 network card, but I did not see a difference at the time because they were both rated for the same speed. I used the cheap network card, which worked fine most of the time, but when I tried to transfer multiple large files over the network, the card failed and lost the connection. It turns out the more expensive card was more capable; because it had more buffers, it could handle the kind of load of many large files being copied to a remote computer. That experience taught me that sometimes I can speed up an application by changing out a \$15 piece of hardware instead of getting a new \$10,000 server that leaves the true bottleneck in place.

To find your bottlenecks, you will need to measure the performance of each part of your system. You may be pleasantly surprised that you can boost performance with the simplest of adjustments.

Traffic Spikes

The average time between requests shown in Table 10-1 does not truly represent traffic patterns. Nothing can ensure that the requests on your web server are so conveniently spaced apart. In reality, the vast majority of web traffic looks like a bell-shaped curve. With a 24-hour clock, you will surely get most of your traffic during the day if your site is primarily used by people in your nearby time zones (which is normally the case). In the United States, the traffic to your website will start to increase when it is 6 a.m. on the East Coast and start to drop off after it is 10 p.m. on the West Coast. And at noon you will typically see your most traffic,

depending on what your website provides. Sites such as news sites get dramatically high spikes of traffic during the lunch hour, when people check the headlines during their lunch breaks.

In light of these traffic spikes, you cannot assume that a website getting a total of 100,000 hits in a day will have 0.86 seconds to process each request before scalability becomes an issue. Your web server may be brought to its knees at a few key points in the day due to regular traffic spikes or in response to your actions. If you are running an informational site that sends out an e-mail newsletter to millions of readers, you may suddenly get a surge of traffic over the hours and days following the newsletter's release. Websites such as Digg (www.digg.com) and Slashdot (<http://slashdot.org>) have millions of readers who are all directed to the headlines placed on their home pages. This results in unsuspecting websites, the ones that Digg and Slashdot link to, getting a massive amount of traffic in a very short period of time. That effect has been dubbed the *Slashdot effect*, and it's known for taking sites offline that are not prepared for large surges in traffic. More recently, Digg has become extremely popular, so your website may also be hit by the *Digg effect*. You may simply have posted an interesting blog entry or photo that enough Digg users found worthwhile. There really is no warning that such an event is about to happen.

SURVIVING THE DIGG EFFECT

I have had my websites hit by Slashdot and Digg twice and was able to weather the storm by making some adjustments in response to the initial surge. In both instances, the content was hosted on an Apache web server, and I simply had to adjust the configuration for the newly increased load on my modestly powered server. The content was already ready to go because it was set up as static content that Apache could serve from memory, much like output caching for ASP.NET. In the case of the Apache web server, the bottleneck was created by the number of forked children I was allowing to be created to handle requests. With IIS this is not a concern because IIS is fully threaded, as is a modern Apache 2.0 installation, which can at least scale for the number of incoming requests. But if each request hits a database multiple times, you may see that the bottleneck is not the web server.

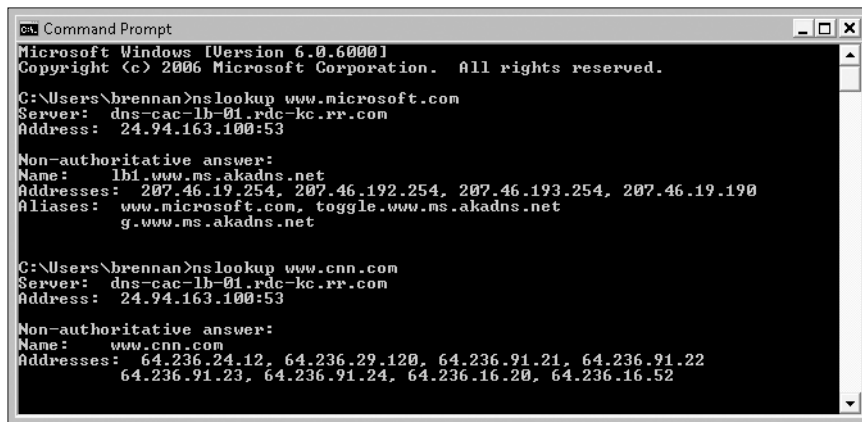
These times when you get a massive surge of traffic require an emergency response plan. I remember the tragic day of the 9/11 attacks and how everybody was desperately looking for any news they could find to tell them what was happening. One of the websites I was viewing was CNN.com. I watched as the website went from the full website we usually see with busy links and ads all over the home page in the first part of the day, to just a few links and one headline. They were clearly trying to handle the surge of traffic, which they surely had not experienced before. They successfully kept their website online and got that critical news to their readers.

Other times you know there will be a surge of traffic. During the Super Bowl, the NFL website is under a great deal of load. The people running the website can plan for this all year and typically set up a special website for the Super Bowl for that year, where all traffic is directed leading up to and during the game. The traffic on this website can be intense as football fans watch the game on their TV screens while also getting the latest stats from the website. Interactive applications show a diagram of the field and provide all the details about the game, updated continually. Each time the web application hits the server, it gets information about

what line the ball is on, who has possession, and what down it is. Often these are Adobe Flash applications, which can reach out to different sources other than just the web server serving up the web pages and can work with a data stream that is optimized to reduce the overall bandwidth requirements. Sending a 4 KB data stream is definitely preferable to having the user refresh a 50 KB page for each update. By managing the flow of data by an alternate, more efficient communication scheme, the high load is more easily managed.

Distributing Traffic

Traffic can be distributed so that while you are getting 500,000 hits per day, you could spread them across five servers to make it effectively 100,000 hits per day per server. This web farm approach is pretty common and is supported by load-balancing hardware. You can even leverage the fact that one hostname can be associated with multiple IP addresses through the Domain Name System (DNS). When a hostname has multiple IP addresses for a website, the web browser will circulate through each of the IP addresses. This technique is called *Round Robin DNS*. Sites such as CNN.com and Microsoft have multiple IP addresses configured for their website addresses to benefit from distributing the load. Figure 10-1 shows the result of running the utility `nslookup` on `www.cnn.com` and `www.microsoft.com`. You can see that there are multiple IP addresses for both websites.



```
Microsoft Windows [Version 6.0.6000]
Copyright (c) 2006 Microsoft Corporation. All rights reserved.

C:\Users\brennan>nslookup www.microsoft.com
Server: dns-cac-lb-01.rdc-kc.rr.com
Address: 24.94.163.100:53

Non-authoritative answer:
Name:    lb1.www.ms.akadns.net
Addresses: 207.46.19.254, 207.46.192.254, 207.46.193.254, 207.46.19.190
Aliases: www.microsoft.com, toggle.www.ms.akadns.net
          g.www.ms.akadns.net

C:\Users\brennan>nslookup www.cnn.com
Server: dns-cac-lb-01.rdc-kc.rr.com
Address: 24.94.163.100:53

Non-authoritative answer:
Name:    www.cnn.com
Addresses: 64.236.24.12, 64.236.29.120, 64.236.91.21, 64.236.91.22
          64.236.91.23, 64.236.91.24, 64.236.16.20, 64.236.16.52
```

Figure 10-1. *Round Robin DNS*

A web farm made up of multiple servers running off a common database server can effectively distribute the processing load, and with a good data-caching policy, a web farm can also reduce the load on the database.

Beyond using a logical distribution through DNS alone, you can also distribute the hardware to multiple physical networks. You may place servers at hosting facilities in a few major cities that have access to a major backbone of the Internet, which will guarantee reliable and fast access to your website for all of your users. I live in Milwaukee and remember that back in 1998 there were a few times that major lines between Milwaukee and Chicago were cut off as new lines were being installed. At the time, there were no redundant lines going to other locations such as Madison and Minneapolis as they do today. We got calls from customers saying they could not reach the websites we were hosting for them even though they were online and

even reachable from different locations within the city. If these websites were co-located at multiple hosting facilities, they may have been inaccessible to only a small portion of users or none at all.

Distributing Content

While distributing the traffic to a website offers speed of access and reliability, you can also achieve similar benefits by distributing the content that you are publishing. Much of the bandwidth on your website is likely due to images and other static media. There is no reason it all has to come from the same web server. You could publish your ASP.NET website with all dynamic pages on one server and all media on another, or perhaps distribute the media across many servers. The server producing the dynamic pages will then be freed up to handle personalized content.

For years Apple, Microsoft, Amazon.com, and Facebook have used services that spread their media across a vast network of web servers to allow them to cope with traffic spikes affecting them regularly as they announce new products and services.

HOSTING ON AKAMAI

The popular content distribution network used by the major websites for years has been Akamai (www.akamai.com). The primary service it offers has been image caching, although it has expanded to dynamic and personalized content. If you were to look up `images.apple.com`, you will see underneath the hostname that Akamai is hosting the content.

A useful aspect of distributing the content to separate websites is that you can fully leverage the delivery mechanisms provided by HTTP, such as the expiration header (one of many headers that precede the content of a response from your web server). When you use output caching, the expiration header is set. If the user returns to that page within the expiration period, that user's web browser may simply read the content from the local browser cache. Client-side caching is further enhanced when a large group of users are behind a caching proxy that uses the expiration headers to hold onto the content from your website for all users so that you do not have to serve up every request. The large Internet service providers (ISPs) such as AOL have used massive proxies to reduce the bandwidth on their networks to benefit their users and reduce their costs. Reducing bandwidth was a major concern for AOL when most of its users were on dial-up modems and bandwidth was limited. More recently, Google has offered their web accelerator service, which is essentially a massive proxy that caches a great deal of content. However, it cannot cache dynamic content, which includes every ASP.NET website that is generated without output caching enabled.

If your website has a great deal of static content, such as images, style sheets, and JavaScript, you could benefit greatly from client-side caching, although these benefits are not guaranteed. In fact, having this sort of content on an ASP.NET website can prevent it from being cached as much as it could be, because every piece of content on an ASP.NET website is associated with the cookies that make up the anonymous and authenticated sessions. A caching proxy will have to assume that the static content that a user pulls from your website that comes with a cookie must be unique to that user. The proxy will then not cache that same

35 KB header graphic used on every page of your website for all users behind the shared proxy. What you can do is place your static content onto a sub-domain that is not configured as an ASP.NET website, so that there are absolutely no cookies involved. You will need to ensure that if your main website is running at `www.acme.com` and your images are running at `images.acme.com`, your cookies are not set to be coming generically from `acme.com`, which would cause the cookies to be associated with the images.

After your static content meets the standards that most proxies follow, you will start to benefit from truly distributing your content by leveraging resources beyond your direct control. However, you will need to be careful with the static content. I learned about the problem that proxies have with cookies from a presentation by a developer from Yahoo! who explained that they have a one-time-use policy for static images. After an image has been published, it cannot change. Instead, they have to publish an updated image to a new URL and point their pages to it. The approach is meant to ensure that their users are getting the right image regardless of how aggressive the client-side caching is. The same could be done for JavaScript, which has again become a critical part of modern web applications.

DEVELOPING WITH STATIC FILES

Renaming static files as you work with them during development is not a realistic option. I tend to place a unique query string at the end of a script reference (`script.js?20070707`) to ensure that when it goes to the staging server to be tested, the Quality Assurance team gets the latest version of the script. It saves me from having to version-control an ever-changing filename while also keeping the application aligned with the latest version. When the website is released to the users, I can rename the script to include an arbitrary and unique version number to ensure that the day the new website goes live, users are not using the old version of the script. I find this is sometimes necessary for style sheets as well, especially with Internet Explorer, which very reluctantly replaces items that it has cached locally.

Distributing Services

You can also split the dynamic content of your website into multiple sub-domains. CNN.com could split their website into multiple sub-domains such as `weather.cnn.com`, `politics.cnn.com`, `entertainment.cnn.com`, and `sports.cnn.com`. Google already has their services split into sites such as `maps.google.com`, `mail.google.com`, and `news.google.com`. With each service set up to run in a more independent way, you will gain flexibility you would not have with a single large website. Each of the sub-domains would be dedicated to a specific service.

With the services split up onto separate servers, you will have the ability to independently manage each of the websites. If the development team working on the mapping website is ready to put out a new release, the team can do so without being concerned about how it will impact the mail and news websites (because they are decoupled from the mapping website). The mail and news websites at Google likely have their own servers and release schedules with a minimal number of dependencies between them. Each website can then act as one collective by using their single sign-on functionality, which unifies all Google websites, much like you would do with your own website using the ASP.NET Membership Provider.

If you could break your website into multiple sections and split them across multiple servers while also breaking up the development and release schedule, you could take an unwieldy website and shape it into multiple websites that are much easier to manage from

technical and project management points of view. As traffic increases on one of the services, you can add more web servers to the web farm that is hosting that service to improve overall performance. Your services could also overlap across servers. Table 10-2 shows an intranet website split across many sub-domains to service a large company. The services are grouped by Human Resources (HR), Timesheets (TS), Accounts Payable (A/P), Accounts Receivable (A/R), and Contracts (C).

Table 10-2. *Overlapping Services*

Server	HR	TS	A/P	A/R	C
Web01	X		X		
Web02		X	X		
Web03			X		X
Web04			X		X
Web05			X		X
Web06	X			X	
Web07		X		X	
Web08				X	X
Web09				X	X
Web10				X	X

With ten servers in the web farm, the services can be placed on each of them. To reduce the cost of maintenance, however, the services are placed on only as many servers as necessary to provide the desired level of service. The applications that get much more traffic and require more resources, such as A/P and A/R, are placed on five servers each. The services are also on separate servers so that no A/P service is hosted on an A/R server. Meanwhile, the lesser-used applications such as Contracts, Human Resources, and Timesheets are spread across the ten servers to ensure an even spread. Traffic to these ten servers could be managed by Round Robin DNS or a load-balancing router.

Distributing the Back End

The approaches in the previous section break up the front-end traffic, but because the bottle-neck will often be the database, you will want to consider breaking up the various groupings of data so you can run multiple database servers. For the intranet example in the previous section, the breakdown could easily go along the lines of the five services: Human Resources, Timesheets, Accounts Payable, Accounts Receivable, and Contracts. Initially, when you set out to create a distributed back end, you can do so logically long before you physically distribute the databases. Either SQL Server could have a single logical database with each of the tables and stored procedures carefully managed so they do not become interdependent over time, or you can run multiple logical databases on the same physical server. Later, as the company grows to the point that the single physical server is not enough, you can move one or more of the databases to a second server to physically distribute the back end. Table 10-3 shows the initial configuration for a small company with a single database server.

Table 10-3. *Small Company Configuration*

Hostname	Database Name	Service	Database
hrdb.acme.com	HR	Human Resources	DB1
tsdb.acme.com	TS	Timesheets	DB1
apdb.acme.com	AP	Accounts Payable	DB1
ardb.acme.com	AR	Accounts Receivable	DB1
contactsdb.acme.com	Contracts	Contracts	DB1

Table 10-3 shows the configuration of the five databases that handle all the services for the intranet website. A small company needs only a single database, so each of the hostnames point to the DB1 database, which may be at an internal IP address of 10.10.1.101. Each of the hostnames point to that IP address. Later, when the databases are distributed to separate physical servers, that data will be moved and the DNS records will be updated while the application configurations remain the same, as long as they are configured with these hostnames. Listing 10-1 shows the connection string for the AP database.

Listing 10-1. *Connection String for AP Database*

```
Data Source=apdb.acme.com;Initial Catalog=AP;Integrated Security=True
```

When the company grows and the application performance needs to improve, you could move the database that is under the most load to a new database server, perhaps with much more capable hardware than the initial database server created when the company was smaller and had a much smaller budget. Now instead of just a basic RAID and dual processor server, you may be stepping up to a SAN with a quad processor server, which could handle your top two databases. This updated configuration is shown in Table 10-4.

Table 10-4. *Growing Company Configuration*

Hostname	Database Name	Service	Database
hrdb.acme.com	HR	Human Resources	DB1
tsdb.acme.com	TS	Timesheets	DB1
apdb.acme.com	AP	Accounts Payable	DB2
ardb.acme.com	AR	Accounts Receivable	DB2
contactsdb.acme.com	Contracts	Contracts	DB1

After the databases for the A/P and A/R services have been moved, the DNS records for apdb.acme.com and ardb.acme.com could be pointed to the IP address of the new server, 10.10.1.102. The change to the new IP will not be instant, as the DNS record typically has a “time to live” setting of 30 minutes or an hour. To make the change immediate, you will need to flush the DNS on the application servers to force them to get the latest DNS information and begin routing the right requests to the databases. For a time, you may want to set the A/P and A/R databases to Read Only and leave them online for a while after the transition.

Planning for Scalability

With all of the suggestions covered to this point in the chapter, you should have several options to choose from when you start to experience growing pains. Unfortunately, there is a lot of space between knowing what you can do to improve the scalability of your application and being able to actually do it. And as much as you do not want to immediately set out to make your little website scalable to 1 billion users, you also should not ignore that your software should be flexible enough to adapt to your changing needs.

Throughout this book, you have looked into the techniques and features of the ASP.NET environment that assist us with building fast websites. Now you will work on piecing it all together into a flexible web application that is easy to build and easy to adapt to changing needs. As the load on the website increases, the software design will allow you to change the components that need a new approach without forcing the entire system to change. But you want to ensure that you can start out with the least amount of effort. You will never get a million users if you never get that first user, so you will start simple with a few baseline requirements.

For starters, you will break the business objects into groupings that are managed with custom provider models. Each business grouping will be managed independently while relationships between the business objects will be addressed by the design of the system. You will also create prototypes for future providers to ensure that when it becomes necessary to distribute the application in multiple ways, you will be ready to do so without a significant amount of rework.

Building a highly scalable system can be costly in terms of hardware and time, and because you cannot know what part of your application will need the attention down the road, it is best to simply be ready to adapt instead of investing all your time and energy on a problem that may never appear. Let the performance problems reveal themselves in time. You can make sure you are ready for them.

Note It is best to proactively gather metrics on everything that affects your application, from the number of requests per hour to the memory, disk, and processor usage of every server in your system. As you see the numbers change, you should be able to explain the changes and adapt as needed. The metrics should allow you to see trends so that you can get ahead of them and take the proper actions as necessary. And if the traffic to your website is seasonal, perhaps with a greater peak of traffic during a specific month, you should project that traffic increase by using the numbers for the month from the previous year in relation to the gradual changes you see happening throughout the year. Your goal should be to keep the capacity of your application well ahead of the projected use.

The Sample Application

The sample application for this chapter will be a website for a .NET user group. The website will display information about upcoming meetings, including details about the speakers and sponsors. It will also feature a job listing. There are relationships between some of these things, such as the location that will be associated with each event as well as each job and

sponsor. The relationship to the location objects is a dependency that must be managed so, for example, a location associated with a job is not deleted when the location is also associated with an event.

Creating the Database

We will create the database as we have done in previous chapters by building up a database project with scripts to define the tables, stored procedures, and constraint scripts. The objects in Table 10-5 will be represented in the database.

Table 10-5. *Objects in the Database*

Table	Object
dug_Events	Event
dug_Jobs	Job
dug_JobContacts	JobContact
dug_Locations	Location
dug_Speakers	Speaker
dug_Sponsors	Sponsor

Each of the tables will share the same basic structure with the primary key defined as a bigint named ID as well as DateTime values named Created and Modified, which are set as records are created and updated. The primary key is set as an identity value that will automatically be incremented as new records are created. Alternatively, the primary key could be named specifically for the table, such as EventID, which is a popular practice. Unfortunately, that practice makes it hard to work all records in a generic way, which we will be doing in the next section.

PARTITIONING BY DATE

With the Created and Modified dates set for each record in the database, it will be possible to partition the database on these values. The primary partition can include all records that have a modified date within the last three months, while everything else will belong to another partition that effectively represents an archive for the data. Initially, this partitioning scheme will not be necessary and may never become necessary. But for a high-traffic website that generates a great number of records to the database, this approach would be a helpful option.

A site such as Digg could use a few partitions. One could be for the current week, another for the last three months, and another for everything beyond the most recent three months. Because users generally read and interact on the newest content pages, the first partition will be accessed most frequently and will require the fastest physical access to the data. The other two partitions will have more modest requirements for speed but will need a great deal of space. With modest performance requirements, the archive partitions will cost significantly less, and the primary partition, with the space requirements minimized, will also cost less than a massive storage system with high-performance requirements.

Get, Save, and Delete

The data manipulation methods will follow the get, save, and delete approach covered in Chapter 7. Each stored procedure that gets the data for an Event will return the same columns, so an instance of an Event can be created consistently with each stored procedure. For an Event, we will be using the following stored procedures: `dug_GetEvent.sql`, `dug_GetAllEvents.sql`, and `dug_GetEventsByDate.sql`. Listing 10-2 shows `dug_GetEvent.sql`, which returns a single record that matches the given key.

Listing 10-2. *dug_GetEvent.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND name = 'dug_GetEvent')
    BEGIN
        DROP Procedure dug_GetEvent
    END
GO

CREATE Procedure dbo.dug_GetEvent
(
    @EventID bigint
)
AS

SELECT
    ID, Title, Description, MeetingDate, Created, Modified
FROM dug_Events
WHERE ID = @EventID
GO

GRANT EXEC ON dug_GetEvent TO PUBLIC
GO
```

Users who visit the website will mostly be interested in upcoming events, so instead of just providing access to a specific event or all events, we will create a stored procedure called `dug_EventsByDate` that returns all events between now and the target date that is passed in as a parameter. Using this simple stored procedure represents “performance by design” versus “performance by optimization” because the need is easily anticipated during the design phase. Clearly, if there are eventually 1,000 events in the database and most are for past events, there will be unnecessary overhead involved in getting all records and then displaying only the events for the coming months. This stored procedure is shown in Listing 10-3.

Listing 10-3. *dug_GetEventsByDate.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND name = 'dug_GetEvent')
    BEGIN
        DROP Procedure dug_GetEvent
    END
GO
```

```

CREATE Procedure dbo.dug_GetEvent
(
    @EventID bigint
)
AS

SELECT
    ID, Title, Description, MeetingDate, Created, Modified
FROM dug_Events
WHERE ID = @EventID
GO

GRANT EXEC ON dug_GetEvent TO PUBLIC
GO

```

And because the `dug_GetEventsByDate` stored procedure is using the `MeetingDate` column as the filtering criteria, it is a logical candidate for indexing if the table becomes large—but it is not strictly necessary when the database has just a few records. You may be able to get by easily for years without indexing the `MeetingDate` column. Keep in mind that maintaining indexes also adds load to your database, so the fewer indexes that you add to your tables, the less overhead you add to the system.

The script to save Event records, `dug_SaveEvent.sql`, takes in all the values associated with an Event along with the `OldEventID`, which is defined if the Event is already in the database and is being updated, along with an output parameter named `EventID`. The `EventID` is set whether an update or insert statement is executed during the save procedure. Listing 10-4 shows the save procedure.

Listing 10-4. *dug_SaveEvent.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND name = 'dug_SaveEvent')
    BEGIN
        DROP Procedure dug_SaveEvent
    END
GO

CREATE Procedure dbo.dug_SaveEvent
(
    @Title nvarchar(50),
    @Description text,
    @MeetingDate datetime,
    @SpeakerID bigint,
    @SponsorID bigint,
    @LocationID bigint,
    @OldEventID bigint,
    @EventID bigint OUTPUT
)
AS

```

```

IF (@OldEventID < 0)
    BEGIN
        INSERT INTO dug_Events
        (Title, Description, MeetingDate, SpeakerID,
         SponsorID, LocationID, Created, Modified)
        VALUES (
            @Title,
            @Description,
            @MeetingDate,
            @SpeakerID,
            @SponsorID,
            @LocationID,
            GETDATE(),
            GETDATE()
        )

        SELECT @EventID = @@IDENTITY
    END
ELSE
    BEGIN
        UPDATE dug_Events
        SET
            Title = @Title,
            Description = @Description,
            MeetingDate = @MeetingDate,
            SpeakerID = @SpeakerID,
            SponsorID = @SponsorID,
            LocationID = @LocationID,
            Modified = GETDATE()
        WHERE
            ID = @OldEventID

        SET @EventID = @OldEventID
    END

GO

GRANT EXEC ON dug_SaveEvent TO PUBLIC
GO

```

You can see that the Event record holds foreign key references to the Speaker, Sponsor, and Location tables. The specific data for the Speaker, Sponsor, and Location is not saved as a part of the Event save procedure beyond the foreign key reference with the respective ID value. The boundary between these sets of data is maintained to ensure ease of maintenance. The values stored in the Sponsor table could be changed completely as long as the ID value remains the same, which will preserve the relationship to the Event records.

Finally, the delete procedure is straightforward. All that is necessary is that the ID value for the event and the record are deleted, as shown in Listing 10-5.

Listing 10-5. *dug_DeleteEvent.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'dug_DeleteEvent')
BEGIN
    DROP Procedure dug_DeleteEvent
END
GO

CREATE Procedure dbo.dug_DeleteEvent
(
    @EventID bigint
)
AS

DELETE FROM dug_Events
WHERE ID = @EventID

GO

GRANT EXEC ON dug_DeleteEvent TO PUBLIC
GO

```

The other tables for Speaker, Sponsor, and Location are managed in a similar way, with a few differences due to the uniqueness of the data. Later I will go into how the Location table is used as a dependency by multiple other tables.

Creating Data Access Providers

The modular design of our example website encourages us to use the provider model, which is easy to implement as you saw in Chapter 5. We will create a SQL provider for each of the sets of data used by the .NET user group website, to be used when the initial website goes live. Later, as the website grows and needs to scale further, new providers can be created and deployed as needed. One provider we will create later uses LINQ, which could be faster than using the Enterprise Library and ADO.NET. We will also create a WCF provider, which would allow us to configure the database back end in a distributed way that would spread the database load.

EventProvider Object

For the moment, we will focus on the EventProvider, shown in Listing 10-6.

Listing 10-6. *EventProvider.cs*

```

namespace DotnetUserGroup.DataAccess.Events
{
    public abstract class EventProvider : ProviderBase, ILocationConsumer
    {

```

```

#region " Provider Methods "

public abstract Event GetNewEvent();

public abstract Event GetEvent(DomainKey key);

public abstract EventCollection GetAllEvents();

public abstract EventCollection GetEventsByDate(DateTime targetDate);

public abstract DomainKey SaveEvent(Event evt);

public abstract void DeleteEvent(Event evt);

public abstract bool IsUsingLocation(Location location);

#endregion

}
}

```

As covered in Chapter 5, the provider inherits from the `ProviderBase` class and declares all the operations as abstract methods. The `EventProvider` primarily works with the `Event` object, which holds onto each of the database values that are returned by the “get” procedures covered in the previous section. The `Event` class is shown in Listing 10-7.

Listing 10-7. *Event.cs*

```

namespace DotnetUserGroup.DataAccess.Events
{
    [DataContract]
    public class Event : DomainObject<Event>
    {
        protected internal Event() { }

        public Event(DataRow row)
        {
            Load(row);
        }

        public Event(IDataReader dr)
        {
            Load(dr);
        }

        public static Event CreateNewEvent()
        {

```

```
        Event evt = new Event();
        evt.ID.Value = (long) -1;
        return evt;
    }

    private string _title;

    [DataMember(Name = "Title",Order = 1)]
    public string Title
    {
        get { return _title; }
        set { _title = value; }
    }

    private string _description;

    [DataMember(Name = "Description",Order = 1)]
    public string Description
    {
        get { return _description; }
        set { _description = value; }
    }

    private DateTime _meetingDate = DefaultDateTime;

    [DataMember(Name = "MeetingDate",Order = 1)]
    public DateTime MeetingDate
    {
        get { return _meetingDate; }
        set { _meetingDate = value; }
    }

    private Speaker _speaker;

    [DataMember(Name = "Speaker",Order = 1)]
    public Speaker Speaker
    {
        get
        {
            if (_speaker == null)
            {
                // lazy load the speaker
                EventSection section = new EventSection();
                _speaker = SpeakerManager.
                    GetProvider(section.SpeakerProvider).GetSpeaker(this);
            }
            return _speaker;
        }
    }
}
```

```

        }
        set { _speaker = value; }
    }

    private Sponsor _sponsor;

    [DataMember(Name = "Sponsor", Order = 1)]
    public Sponsor Sponsor
    {
        get
        {
            if (_sponsor == null)
            {
                // lazy load the sponsor
                EventSection section = new EventSection();
                _sponsor = SponsorManager.
                    GetProvider(section.SponsorProvider).GetSponsor(this);
            }
            return _sponsor;
        }
        set { _sponsor = value; }
    }

    private Location _location;

    [DataMember(Name = "Location", Order = 1)]
    public Location Location
    {
        get
        {
            if (_location == null)
            {
                // lazy load the location
                EventSection section = new EventSection();
                _location = LocationManager.
                    GetProvider(section.LocationProvider).GetLocation(this);
            }
            return _location;
        }
        set { _location = value; }
    }
}

```

The Event object generally holds onto properties that come from the database. Because loading the data is typically a tedious task, this work has been set to work automatically via an updated version of the DomainObject covered in Chapter 7.

Revised DomainObject

The revised DomainObject used for the .NET user group website is a lot like the one used in Chapter 7, with a few differences. One difference in particular is how the primary key is always assumed to line up with a column named ID. In Chapter 7, the ID was simply a number, whereas the updated DomainObject uses a new type called DomainKey, which can hold onto primary key values that could be one of the following types: Int16, Int32, Int64, or GUID.

When a column named ID is pulled from the DataSet or IDDataReader, it is assumed to be the primary key value and is set on the DomainKey property. Listing 10-8 shows the DomainKey class.

Listing 10-8. *DomainKey.cs*

```
namespace DotnetUserGroup.DataAccess.Common
{
    [DataContract]
    public class DomainKey : IComparable<DomainKey>
    {
        public DomainKey(object keyValue)
        {
            Value = keyValue;
        }

        private object _keyValue;

        [DataMember(Name = "Value", Order = 1)]
        public object Value
        {
            get { return _keyValue; }
            set
            {
                // check in the order which is most likely
                if (value is Int32 ||
                    value is Guid ||
                    value is Int64 ||
                    value is Int16)
                {
                    _keyValue = value;
                }
                else
                {
                    throw new Exception("Type not supported as a DomainKey: " +
                        value.GetType());
                }
            }
        }

        public static DomainKey Default
        {

```

```

        get
        {
            DomainKey defaultKey = new DomainKey(-1);
            return defaultKey;
        }
    }

    #region " Comparison Methods "

    public int CompareTo(DomainKey other)
    {
        return Value.ToString().CompareTo(other.Value.ToString());
    }

    public override int GetHashCode()
    {
        return Value.GetHashCode();
    }

    public override bool Equals(object obj)
    {
        if (obj != null && obj is DomainKey)
        {
            DomainKey other = (DomainKey) obj;
            return Equals(other);
        }
        return false;
    }

    public bool Equals(DomainKey other)
    {
        if (other != null && Value.Equals(other.Value))
        {
            return true;
        }
        return false;
    }

    #endregion
}

```

The primary purpose of the `DomainKey` is to abstract away the type of the primary key. One implementation of the `EventProvider` could use the `long` value, which is used by the SQL implementation, while a future implementation may use a `Guid` value, which is more appropriate when data is spread across multiple databases and a sequential identity key cannot be maintained properly. The `EventProvider` will not have to change when the type of the primary key is changed from a `long` to a `Guid` in the future implementation. However, the `Event` class

does reference the associated Sponsor, Speaker, and Location, which are also set up with primary keys that are using long values. If one of these implementations is adjusted to use a Guid value, it will require some adjustments to the stored procedures and table structures.

Another feature of the DomainKey is the implementation of the IComparable<DomainKey> interface, which allows the key to be uniquely identified within a collection, among other uses. It can also uniquely identify one record to another by using the primary key value instead of the default comparison, which will not be accurate. The DomainObject uses the DomainKey type for the ID property then uses the comparison provided by the DomainKey to also compare instances of the DomainObject. The comparison methods used by the DomainObject class are shown in Listing 10-9.

Listing 10-9. *DomainObject Comparison Methods*

```
public int CompareTo(Object obj)
{
    int result = 0;
    DomainObject<T> otherDomainObject = obj as DomainObject<T>;
    if (otherDomainObject != null)
    {
        result = ID.CompareTo(otherDomainObject.ID);
        if (result == 0)
        {
            result = Created.CompareTo(otherDomainObject.Created);
            if (result == 0)
            {
                result = Modified.CompareTo(otherDomainObject.Modified);
            }
        }
    }
    return result;
}

public override int GetHashCode()
{
    return ID.GetHashCode();
}

public override bool Equals(object obj)
{
    DomainObject<T> domainObject = obj as DomainObject<T>;
    if (domainObject != null)
    {
        if (ID.Equals(domainObject.ID))
        {
            return true;
        }
    }
    return false;
}
```

Now any object that inherits from the `DomainObject` can be compared against another to determine whether they are the same object, such as the same `Event` or `Speaker`. The primary key value is automatically one part of the comparison, while the `Created` and `Modified` values are another part. An old copy of a `Speaker` object will not match a new copy, whereas a `Speaker` record loaded into two `Speaker` objects will match as they should. This feature is provided automatically for any class built by using the conventions defined for the data access layer used by this website. Because each table created for this website uses the `ID`, `Created`, and `Modified` columns, the loading procedure will populate their respective properties in the `DomainObject` instances as planned.

Meanwhile, the rest of the data-loading process happens as it did in Chapter 7. The load methods can take either a `DataRow` or an `IDataReader` to populate the `DomainObject`. Refer to the downloadable samples for this chapter for the full code listing.

DOMAINOBJECT AND GENERICS

Generics were used for the revised `DomainObject` because of a problem combining inheritance and reflection. A `TargetException` was thrown in certain scenarios even though the names and types for the data columns and object properties matched. I eventually discovered through Ayende Rahien, a lead developer on the `NHibernate` project, that he also had a similar problem with inheritance and reflection. I got passed the problem with the `TargetException` by making the `DomainObject` into a generic type that required all inheriting classes to be a `DomainObject<T>`, where `T` is a `DomainObject`. When `Location` inherits from `DomainObject<Location>`, it meets the generics criteria and overcomes the problem with reflection.

Managing Relationships

Back in Listing 10-7 for the `Event` class, you may have noticed that there are properties representing relationships. Specifically, the properties named `Speaker`, `Sponsor`, and `Location` are values that are not pulled in with the “get” stored procedures for the `Event` object. For the `Location` property, the actual value is left as null initially and loaded as needed. This is the “lazy loading” technique covered previously. When the `Location` property is accessed, the “getter” will check whether the variable is defined and if it is not, it will attempt to get the value from the `LocationProvider`, as shown in Listing 10-10.

Listing 10-10. *Getter for Location Property*

```
private Location _location;

[DataMember(Name = "Location", Order = 1)]
public Location Location
{
    get
    {
        if (_location == null)
        {
            // lazy load the location
            EventSection section = new EventSection();
```

```

        _location = LocationManager.
            GetProvider(section.LocationProvider).GetLocation(this);
    }
    return _location;
}
set { _location = value; }
}

```

When the `Location` property is used, the value is loaded by using the `LocationManager` class, but instead of just using the default provider to get a location, the `EventSection` class is used to determine the name of the provider to use. Later we will get into how custom configurations assist with the relationships between objects managed by different providers.

Using Locations

Another feature of the `EventProvider` that is used to manage relationships is the `ILocationConsumer` interface shown in Listing 10-11.

Listing 10-11. *ILocationConsumer* Interface

```

namespace DotnetUserGroup.DataAccess.Locations
{
    public interface ILocationConsumer
    {
        bool IsUsingLocation(Location location);
    }
}

```

The `ILocationConsumer` interface is used to protect a location from being deleted when it is still being used. A `Sponsor` can be associated with a `Location`, and so can an `Event`. If a `Sponsor` or an `Event` is using a `Location`, it should not be deleted. Each provider that has a relationship with locations implements the `ILocationConsumer` interface.

When the delete method for a `Location` is called, the `LocationManager` checks whether any provider implementation is using the `Location`. Listing 10-12 shows the `IsLocationInUse` method, which checks all the implementations that implement the `ILocationConsumer` interface.

Listing 10-12. *IsLocationInUse* Method

```

public static bool IsLocationInUse(Location location)
{
    foreach (ILocationConsumer locationConsumer in GetLocationConsumers())
    {
        if (locationConsumer.IsUsingLocation(location))
        {
            return true;
        }
    }
    return false;
}

```

The `IsLocationInUse` method calls the `IsUsingLocation` method, which is defined by the `ILocationConsumer` interface on each instance in the collection. The `GetLocationConsumers` method, shown in Listing 10-13, creates this collection.

Listing 10-13. *GetLocationConsumers Method*

```
private static ILocationConsumer[] GetLocationConsumers()
{
    List<ILocationConsumer> locationConsumers =
        new List<ILocationConsumer>();
    foreach (ProviderBase provider in
        AssemblyHelper.GetMatchedProviders(typeof(ILocationConsumer)))
    {
        ILocationConsumer locationConsumer = provider as ILocationConsumer;
        if (locationConsumer != null)
        {
            locationConsumers.Add(locationConsumer);
        }
    }
    return locationConsumers.ToArray();
}
```

The `GetLocationConsumers` method simply gets a collection from the `AssemblyHelper` class for all matched providers for the `ILocationConsumer` interface. The `AssemblyHelper` class is a utility class created to discover all configured providers that match a given type so that other interfaces, such as `ISpeakerConsumer` and `ISponsorConsumer`, can also be used in the same way. Listing 10-14 shows the `AssemblyHelper` class.

Listing 10-14. *AssemblyHelper.cs*

```
namespace DotnetUserGroup.DataAccess.Common
{
    public class AssemblyHelper
    {

        public static Type[] GetMatchedTypes(Assembly assembly, Type searchType)
        {
            List<Type> types = new List<Type>();
            Module[] modules = assembly.GetLoadedModules();
            foreach (Module module in modules)
            {
                foreach (Type type in module.GetTypes())
                {
                    if (type.IsClass &&
                        !type.IsAbstract &&
                        searchType.IsAssignableFrom(type))
                    {
                        types.Add(type);
                    }
                }
            }
        }
    }
}
```

```

        }
    }
}

return types.ToArray();
}

public static ProviderBase[] GetMatchedProviders(Type searchType)
{
    List<ProviderBase> providers = new List<ProviderBase>();

    DugConfiguration sectionGroup =
        DugConfiguration.GetConfiguration();

    foreach (ProviderConfigurationSection section in
        sectionGroup.ProviderSections)
    {
        foreach (ProviderSettings settings in section.Providers)
        {
            // The assembly should be in \bin or GAC
            Type providerType = Type.GetType(settings.Type, false);
            Assembly providerAssembly =
                providerType.Assembly;
            if (providerType.IsClass &&
                !providerType.IsAbstract &&
                searchType.IsAssignableFrom(providerType))
            {
                ProviderBase provider =
                    Activator.CreateInstance(providerType) as ProviderBase;
                if (provider != null)
                {
                    provider.Initialize(settings.Name, settings.Parameters);
                    providers.Add(provider);
                }
            }
        }
    }
    return providers.ToArray();
}
}
}

```

The `AssemblyHelper` class does not simply return the types that implement the `ILocationConsumer` interface. It also instantiates the configured providers and returns the instances so that the `IsUsingLocation` method can be called. The assemblies that are checked are all from the `DugConfiguration`, which is the root of all custom configurations for the providers. We briefly looked at custom configurations in Chapters 5 and 9 and will cover them in more depth next.

Custom Configuration

The glue that holds everything together is the configuration. Although each of the providers is designed to work independently, there are relationships between them that need to be maintained. The nature of these relationships can be changed with the configuration alone instead of rebuilding and redeploying your application. For example, you may want to use the SQL implementation for each provider initially but later configure a new *SpeakerProvider* implementation that integrates with an existing speaker database. All that is needed is the assembly that holds the class that inherits from the abstract *SpeakerProvider* class and an update to the configuration to point to the new class and assembly. The change is then configured with the respective provider section.

Each custom provider defines a configuration section that inherits from the standard class named *ConfigurationSection*. Listing 10-15 shows the *SpeakerSection* class.

Listing 10-15. *SpeakerSection.cs*

```
namespace DotnetUserGroup.DataAccess.Speakers
{
    public class SpeakerSection : ProviderConfigurationSection
    {
        [ConfigurationProperty("providers")]
        public override ProviderSettingsCollection Providers
        {
            get { return (ProviderSettingsCollection)base["providers"]; }
        }

        [StringValidator(MinLength = 1)]
        [ConfigurationProperty("defaultProvider",
            DefaultValue = "SqlSpeakerProvider")]
        public override string DefaultProvider
        {
            get { return (string)base["defaultProvider"]; }
            set { base["defaultProvider"] = value; }
        }
    }
}
```

The custom configuration in Chapter 9 simply defined a placeholder for a couple of configuration values. In Listing 10-15, the provider section defines a property named *Providers*, which leads to the *ProviderSettingsCollection*, as well as a property named *DefaultProvider*, which points to the provider to use when a name is not given. This should remind you of the standard ASP.NET providers because it is implemented in the same way.

The *Providers* property uses the *ConfigurationProperty* attribute to point to the related element in the configuration. The *ProviderSettingsCollection* is a standard type that will automatically handle the configuration within the providers section. Listing 10-16 shows a sample configuration for the speakers section.

Listing 10-16. Sample Configuration Section

```
<speakers defaultProvider="SqlSpeakerProvider">
  <providers>
    <clear />
    <add name="SqlSpeakerProvider"
        connectionStringName="dug"
        type="DotnetUserGroup.DataAccess.Speakers.SqlSpeakerProvider, ↗
DotnetUserGroup.DataAccess" />
  </providers>
</speakers>
```

You can see that the providers element lines up with the Providers property in the SpeakerSection class due to the ConfigurationProperty attribute. Within the providers section, you can clear, add, or remove provider definitions that must include the name and type attributes as well as any values the provider implementation needs.

Configuration Grouping

Each provider section looks much the same, and each will be a child element of the dotnetUserGroup element, which is associated with the DugConfiguration class that inherits from the standard ConfigurationSectionGroup class. This section group is what contains all the configurations that cleanly isolate them from the rest of the configuration. The hierarchy helps keep the configuration organized. Listing 10-17 shows the general structure of this hierarchy.

Listing 10-17. Configuration Hierarchy

```
<dotnetUserGroup>
  <events />
  <jobs />
  <jobContacts />
  <locations />
  <speakers />
  <sponsors />
</dotnetUserGroup>
```

In the AssemblyHelper class shown in Listing 10-14, the ProviderSections property on the DugConfiguration class was used to iterate over all the providers to get to the instances that match the target type. Listing 10-18 shows how the ProviderSections property is defined.

Listing 10-18. ProviderSections Property

```
private List<ProviderConfigurationSection> _providerSections = null;

public ProviderConfigurationSection[] ProviderSections
{
    get
    {
```

```

        if (_providerSections == null)
        {
            _providerSections = new List<ProviderConfigurationSection>();
            _providerSections.Add(EventSection);
            _providerSections.Add(JobContactSection);
            _providerSections.Add(JobSection);
            _providerSections.Add(LocationSection);
            _providerSections.Add(SpeakerSection);
            _providerSections.Add(SponsorSection);
        }
        return _providerSections.ToArray();
    }
}

```

The sections referenced in the `ProviderSections` property are also defined as properties in the `DugConfiguration`, as shown in Listing 10-19.

Listing 10-19. *Properties for Sections*

```

[ConfigurationProperty("events")]
public EventSection EventSection
{
    get {
        return Sections["events"] as EventSection;
    }
}

[ConfigurationProperty("jobContacts")]
public JobContactSection JobContactSection
{
    get {
        return Sections["jobContacts"] as JobContactSection;
    }
}

[ConfigurationProperty("jobs")]
public JobSection JobSection
{
    get {
        return Sections["jobs"] as JobSection;
    }
}

[ConfigurationProperty("locations")]
public LocationSection LocationSection
{
    get {
        return Sections["locations"] as LocationSection;
    }
}

```

```

    }
}

[ConfigurationProperty("speakers")]
public SpeakerSection SpeakerSection
{
    get {
        return Sections["speakers"] as SpeakerSection;
    }
}

[ConfigurationProperty("sponsors")]
public SponsorSection SponsorSection
{
    get {
        return Sections["sponsors"] as SponsorSection;
    }
}

```

You may have noticed that the `ProviderSections` property returns an array of `ProviderConfigurationSection` classes instead of `ConfigurationSection` classes. This is a custom class that makes it possible to group all the provider configuration sections together. The `ProviderConfigurationSection` class defines two abstract properties: `Providers` and `DefaultProvider`. The `SpeakerSection` class shown in Listing 10-15 inherits from this class. See Listing 10-20 for this abstract class.

Listing 10-20. *ProviderConfigurationSection.cs*

```

namespace DotnetUserGroup.DataAccess.Common
{
    public abstract class ProviderConfigurationSection : ConfigurationSection
    {
        public abstract ProviderSettingsCollection Providers { get; }
        public abstract string DefaultProvider { get; set; }
    }
}

```

Declaring the Custom Configuration

In order to use the custom configuration, it must be declared. Declaring custom configuration sections must be done at the beginning of a configuration file, right after the opening `Configuration` element. Listing 10-21 shows the declaration of the section group and sub-sections for the sample application.

Listing 10-21. *Configuration Declaration*

```

<configSections>
  <sectionGroup name="dotnetUserGroup"

```

```

        type="DotnetUserGroup.DataAccess.Common.DugConfiguration, ➡
DotnetUserGroup.DataAccess">
        <section name="events"
            type="DotnetUserGroup.DataAccess.Events.EventSection, ➡
DotnetUserGroup.DataAccess" />
        <section name="jobs"
            type="DotnetUserGroup.DataAccess.Jobs.JobSection, ➡
DotnetUserGroup.DataAccess" />
        <section name="jobContacts"
            type="DotnetUserGroup.DataAccess.JobContacts.JobContactSection, ➡
DotnetUserGroup.DataAccess" />
        <section name="locations"
            type="DotnetUserGroup.DataAccess.Locations.LocationSection, ➡
DotnetUserGroup.DataAccess" />
        <section name="speakers"
            type="DotnetUserGroup.DataAccess.Speakers.SpeakerSection, ➡
DotnetUserGroup.DataAccess" />
        <section name="sponsors"
            type="DotnetUserGroup.DataAccess.Sponsors.SponsorSection, ➡
DotnetUserGroup.DataAccess" />
    </sectionGroup>
</configSections>

```

Configuring the Providers

With the custom configuration classes built and the custom sections declared, the provider configuration can be added. Listing 10-22 shows the provider configuration that points to all the standard SQL providers for the sample application.

Listing 10-22. *Provider Configuration*

```

<dotnetUserGroup>
    <events defaultProvider="SqlEventProvider"
        locationProvider="SqlLocationProvider"
        speakerProvider="SqlSpeakerProvider"
        sponsorProvider="SqlSponsorProvider">
        <providers>
            <clear />
            <add name="SqlEventProvider"
                connectionStringName="dug"
                type="DotnetUserGroup.DataAccess.Events.SqlEventProvider, ➡
DotnetUserGroup.DataAccess" />
        </providers>
    </events>
    <jobs defaultProvider="SqlJobProvider">
        <providers>
            <clear />

```

```

        <add name="SqlJobProvider"
            connectionStringName="dug"
            type="DotnetUserGroup.DataAccess.Jobs.SqlJobProvider, ➡
DotnetUserGroup.DataAccess" />
    </providers>
</jobs>
<jobContacts defaultProvider="SqlJobContactProvider">
    <providers>
        <clear />
        <add name="SqlJobContactProvider"
            connectionStringName="dug"
            type="DotnetUserGroup.DataAccess.JobContacts.SqlJobContactProvider, ➡
DotnetUserGroup.DataAccess" />
    </providers>
</jobContacts>
<locations defaultProvider="SqlLocationProvider">
    <providers>
        <clear />
        <add name="SqlLocationProvider"
            connectionStringName="dug"
            type="DotnetUserGroup.DataAccess.Locations.SqlLocationProvider, ➡
DotnetUserGroup.DataAccess" />
    </providers>
</locations>
<speakers defaultProvider="SqlSpeakerProvider">
    <providers>
        <clear />
        <add name="SqlSpeakerProvider"
            connectionStringName="dug"
            type="DotnetUserGroup.DataAccess.Speakers.SqlSpeakerProvider, ➡
DotnetUserGroup.DataAccess" />
    </providers>
</speakers>
<sponsors defaultProvider="SqlSponsorProvider">
    <providers>
        <clear />
        <add name="SqlSponsorProvider"
            connectionStringName="dug"
            type="DotnetUserGroup.DataAccess.Sponsors.SqlSponsorProvider, ➡
DotnetUserGroup.DataAccess" />
    </providers>
</sponsors>
</dotnetUserGroup>

```

The provider settings added within the providers element are required because unlike the standard ASP.NET providers, these settings are not predefined in the `Machine.config` file (covered in Chapter 1, in Listing 1-4). But some values are optional. The `defaultProvider` attribute is optional in each case when the default is what you intend. In Listing 10-15, the

SpeakerSection defined the DefaultProvider property with an attribute for the DefaultValue set to SqlSpeakerProvider. The EventSection class, shown in Listing 10-23, defines additional properties for the providers it should use when populating the values for the Speaker, Sponsor, and Location properties in the Event class. The ConfigurationProperty attribute has the IsRequired value explicitly set to false, which happens to be the default value, so that you can leave this additional information out of the configuration if you want to use the default settings. When you access the properties from the EventSection class, it will return a value whether or not you have defined it in the configuration file.

Listing 10-23. *EventSection.cs*

```
namespace DotnetUserGroup.DataAccess.Events
{
    public class EventSection : ProviderConfigurationSection
    {
        [ConfigurationProperty("providers")]
        public override ProviderSettingsCollection Providers
        {
            get { return (ProviderSettingsCollection)base["providers"]; }
        }

        [StringValidator(MinLength = 1)]
        [ConfigurationProperty("defaultProvider",
            DefaultValue = "SqlEventProvider")]
        public override string DefaultProvider
        {
            get { return (string)base["defaultProvider"]; }
            set { base["defaultProvider"] = value; }
        }

        [StringValidator(MinLength = 1)]
        [ConfigurationProperty("speakerProvider",
            DefaultValue = "SqlSpeakerProvider",
            IsRequired = false)]
        public string SpeakerProvider
        {
            get { return (string)base["speakerProvider"]; }
            set { base["speakerProvider"] = value; }
        }

        [StringValidator(MinLength = 1)]
        [ConfigurationProperty("sponsorProvider",
            DefaultValue = "SqlSponsorProvider",
            IsRequired = false)]
        public string SponsorProvider
        {
            get { return (string)base["sponsorProvider"]; }
            set { base["sponsorProvider"] = value; }
        }
    }
}
```

```

    }

    [StringValidator(MinLength = 1)]
    [ConfigurationProperty("locationProvider",
        DefaultValue = "SqlLocationProvider",
        IsRequired = false)]
    public string LocationProvider
    {
        get { return (string)base["locationProvider"]; }
        set { base["locationProvider"] = value; }
    }
}
}
}

```

Creating New Providers

When you determine that the standard SQL providers are not sufficient to handle your needs, it is easy to create a new provider. All you need to do is inherit from the abstract provider class, such as `EventProvider`, and implement all of the abstract methods. Perhaps everything works as you like except for a few minor details in one provider. You can create a new implementation for that provider and configure your application to use it while leaving the rest of the standard providers in place. Because they all work off of the abstract interface defined by the base provider classes, the new implementation will work seamlessly with the existing providers. And with the source code that you can download for this book, you can use the existing classes as a starting point.

A new provider could be much like the standard SQL provider with a few differences, such as adding data-caching functionality that is not included in the standard implementations. The design for the .NET user group website is to use output caching on user controls to reduce the load on the database instead of placing the caching responsibilities into the data access layer. Content such as the event listings will not change frequently, so it is not necessary to always get the latest data from the provider to render the page. However, you may have different needs and may want to create a new provider that does put the caching functionality into the data access layer. You could create a new `EventProvider` implementation called `SqlCachedEventProvider` and use it in your application.

Implementing a LINQ Provider

Beyond using ADO.NET and the Enterprise Library, we can also now start using LINQ, which is currently out as a CTP. The final release of the .NET 3.5 runtime and Visual Studio 2008 is scheduled for February of 2008. When LINQ is finally released, it will change how we work with data. LINQ, or Language Integrated Query, is a language extension that introduces dynamic language features into C#. What it does is enable us to query objects by using a syntax similar to SQL. You may have a large collection of `Event` objects and want to get a list of the events that are going to be at a certain location in the next six months. A LINQ query can get you that result from C# without having the database involved.

The portion of LINQ that works with the database is called DLINQ. You can use DLINQ to query tables in a database or to get the result from a stored procedure call. For the purpose of implementing a provider, we will create an *EventProvider* implementation with LINQ that calls the same stored procedures as the standard SQL implementation.

To get a jump start on creating the LINQ implementation of the *EventProvider*, I generated a website with Blinq using the database with all the tables and stored procedures. You will recall Blinq from Chapter 8. I took what I needed from the generated website and adjusted a few of the details. For example, I know that each of the stored procedures that gets data from events returns the same set of columns, but Blinq generated a unique class to hold the result for each stored procedure. I took one of the generated result classes and renamed it to *EventResult*. Listing 10-24 shows the *EventResult* class.

Listing 10-24. *EventResult.cs*

```
namespace DotnetUserGroup.DataAccess.LinqProviders.Data
{
    public class EventResult
    {

        #region " Variables "

        private long _ID;

        private string _Title;

        private string _Description;

        private System.DateTime _MeetingDate;

        private System.DateTime _Created;

        private System.DateTime _Modified;

        #endregion

        #region " Constructors "

        public EventResult()
        {
        }

        #endregion

        #region " Properties "

        [Column(Name = "ID", Storage = "_ID", DbType = "BigInt")]
        [DataObjectField(false, false, false)]
        public long ID
```



```
{
    get
    {
        return this._ID;
    }
    set
    {
        if ((this._ID != value))
        {
            this._ID = value;
        }
    }
}

[Column(Name = "Title", Storage = "_Title", DBType = "NVarChar(50)")]
[DataObjectField(false, false, false, 50)]
public string Title
{
    get
    {
        return this._Title;
    }
    set
    {
        if ((this._Title != value))
        {
            this._Title = value;
        }
    }
}

[Column(Name = "Description", Storage = "_Description", DBType = "Text")]
[DataObjectField(false, false, false, 2147483647)]
public string Description
{
    get
    {
        return this._Description;
    }
    set
    {
        if ((this._Description != value))
        {
            this._Description = value;
        }
    }
}
```

```

        [Column(Name = "MeetingDate", Storage = "_MeetingDate",
DBType = "DateTime")]
        [DataObjectField(false, false, false)]
        public System.DateTime MeetingDate
        {
            get
            {
                return this._MeetingDate;
            }
            set
            {
                if ((this._MeetingDate != value))
                {
                    this._MeetingDate = value;
                }
            }
        }

        [Column(Name = "Created", Storage = "_Created", DBType = "DateTime")]
        [DataObjectField(false, false, false)]
        public System.DateTime Created
        {
            get
            {
                return this._Created;
            }
            set
            {
                if ((this._Created != value))
                {
                    this._Created = value;
                }
            }
        }

        [Column(Name = "Modified", Storage = "_Modified", DBType = "DateTime")]
        [DataObjectField(false, false, false)]
        public System.DateTime Modified
        {
            get
            {
                return this._Modified;
            }
            set
            {
                if ((this._Modified != value))
                {

```

```

        this._Modified = value;
    }
}

#endregion

}
}

```

You can see that the `EventResult` class does not inherit from any special LINQ base class but instead decorates each of the properties with attributes that line them up with the stored procedure results. Next the `DataContext` must be defined, which bridges the code to the database. I pulled the methods from the Blinq-generated code related to the events and adjusted them to create the `DugDataContext` class shown in Listing 10-25.

Listing 10-25. *DugDataContext.cs*

```

namespace DotnetUserGroup.DataAccess.LinqProviders.Data
{
    public class DugDataContext : DataContext
    {
        #region " Constructors "

        public DugDataContext(string connection)
            : base(connection)
        {
        }

        public DugDataContext(System.Data.IDbConnection connection)
            : base(connection)
        {
        }

        public DugDataContext(string connection, MappingSource mappingSource)
            : base(connection, mappingSource)
        {
        }

        public DugDataContext(
            IDbConnection connection, MappingSource mappingSource)
            : base(connection, mappingSource)
        {
        }

        #endregion

        #region " Methods "

```

```

[StoredProcedure(Name = "dug_GetEvent")]
public StoredProcedureResult<EventResult> GetEvent(
    [Parameter(Name = "EventID", DBType = "BigInt")]
    System.Nullable<long> eventID)
{
    return this.ExecuteStoredProcedure<EventResult>(
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), eventID);
}

[StoredProcedure(Name = "dug_GetEventsByDate")]
public StoredProcedureResult<EventResult> GetEventsByDate(
    [Parameter(Name = "TargetDate", DBType = "DateTime")]
    System.Nullable<System.DateTime> targetDate)
{
    return this.ExecuteStoredProcedure<EventResult>(
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), targetDate);
}

[StoredProcedure(Name = "dug_GetAllEvents")]
public StoredProcedureResult<EventResult> GetAllEvents()
{
    return this.ExecuteStoredProcedure<EventResult>(
        ((MethodInfo)(MethodInfo.GetCurrentMethod())));
}

[StoredProcedure(Name = "dug_SaveEvent")]
public int SaveEvent(
    [Parameter(Name = "Title", DBType = "NVarChar(50)")]
    string title,
    [Parameter(Name = "Description", DBType = "Text")]
    string description,
    [Parameter(Name = "MeetingDate", DBType = "DateTime")]
    System.Nullable<System.DateTime> meetingDate,
    [Parameter(Name = "SpeakerID", DBType = "BigInt")]
    System.Nullable<long> speakerID,
    [Parameter(Name = "SponsorID", DBType = "BigInt")]
    System.Nullable<long> sponsorID,
    [Parameter(Name = "LocationID", DBType = "BigInt")]
    System.Nullable<long> locationID,
    [Parameter(Name = "OldEventID", DBType = "BigInt")]
    System.Nullable<long> oldEventID,
    [Parameter(Name = "EventID", DBType = "BigInt")]
    ref System.Nullable<long> eventID)
{
    StoredProcedureResult result = this.ExecuteStoredProcedure(
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), title,

```

```

        description, meetingDate, speakerID, sponsorID, locationID,
        oldEventID, eventID);
    eventID = ((System.Nullable<long>)(result.GetParameterValue(7)));
    return result.ReturnValue.Value;
}

[StoredProcedure(Name = "dug_DeleteEvent")]
public int DeleteEvent(
    [Parameter(Name = "EventID", DbType = "BigInt")]
    System.Nullable<long> eventID)
{
    StoredProcedureResult result = this.ExecuteStoredProcedure(
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), eventID);
    return result.ReturnValue.Value;
}

[StoredProcedure(Name = "dug_IsEventUsingLocation")]
public StoredProcedureResult<CountResult> IsEventUsingLocation(
    [Parameter(Name = "LocationID", DbType = "BigInt")]
    System.Nullable<long> locationID)
{
    return this.ExecuteStoredProcedure<CountResult>(
        ((MethodInfo)(MethodInfo.GetCurrentMethod())), locationID);
}

#endregion
}

}

```

The attributes associated with each method declare the name of the stored procedure with the `StoredProcedure` attribute along with each of the parameters that are passed into the stored procedure. You can see that the return value from the get methods is `StoredProcedureResult<EventResult>`, which is not the same as `EventCollection` (as is necessary for the `EventProvider` implementation). A conversion will be necessary. The `EventResultConvert` class shown in Listing 10-26 will convert the LINQ type to the type we need.

Listing 10-26. *EventResultConverter.cs*

```

namespace DotnetUserGroup.DataAccess.LinqProviders.Data
{
    public static class EventResultConverter
    {
        public static EventCollection ToEventCollection(
            StoredProcedureResult<EventResult> eventResults)

```

```

    {
        EventCollection eventCollection = new EventCollection();
        foreach (EventResult eventResult in eventResults)
        {
            eventCollection.Add(Convert(eventResult));
        }
        return eventCollection;
    }

    public static Event Convert(EventResult eventResult)
    {
        Event evt = Event.CreateNewEvent();
        evt.ID.Value = (long) eventResult.ID;
        evt.Title = eventResult.Title;
        evt.MeetingDate = eventResult.MeetingDate;
        evt.Description = eventResult.Description;
        evt.Created = eventResult.Created;
        evt.Modified = eventResult.Modified;
        return evt;
    }
}

```

With the type conversion class in place, it is now possible to create the `LinqEventProvider` class that inherits the `EventProvider` base class. This new provider will include the `Initialize` method that providers have and it will use the configured connection string name to initialize the `DugDataContext` so it can be used by all of the implementation methods. Listing 10-27 shows the full `LinqEventProvider` class.

Listing 10-27. *LinqEventProvider.cs*

```

namespace DotnetUserGroup.DataAccess.LinqProviders.Events
{
    public class LinqEventProvider : EventProvider
    {

        #region " Variables "

        private string connStringName = String.Empty;
        private DugDataContext db;

        #endregion

        #region " Provider Methods "

        /// <summary>
        /// SQL Implementation

```

```

/// </summary>
public override void Initialize(string name,
    NameValueCollection config)
{
    if (config == null)
    {
        throw new ArgumentNullException("config");
    }

    if (String.IsNullOrEmpty(name))
    {
        name = "LinqEventProvider";
    }

    if (String.IsNullOrEmpty(config["description"]))
    {
        config.Remove("description");
        config.Add("description", "LINQ Events Provider");
    }

    base.Initialize(name, config);

    connStringName = config["connectionStringName"].ToString();
    config.Remove("connectionStringName");

    if (WebConfigurationManager.ConnectionStrings[connStringName] == null)
    {
        throw new ProviderException("Missing connection string");
    }

    string connString = ConfigurationManager.
        ConnectionStrings[connStringName].ConnectionString;
    db = new DugDataContext(connString);

    if (config.Count > 0)
    {
        string attr = config.GetKey(0);
        if (!String.IsNullOrEmpty(attr))
        {
            throw new ProviderException("Unrecognized attribute: " + attr);
        }
    }
}

#endregion

#region " Implementation Methods "

```

```

public override Event GetNewEvent()
{
    return Event.CreateNewEvent();
}

public override Event GetEvent(DomainKey key)
{
    StoredProcedureResult<EventResult> eventResults =
        db.GetEvent((long?) key.Value);
    EventCollection eventCollection =
        EventResultConverter.ToEventCollection(eventResults);
    if (eventCollection.Count > 0)
    {
        return eventCollection[0];
    }
    else
    {
        return null;
    }
}

public override EventCollection GetAllEvents()
{
    StoredProcedureResult<EventResult> eventResults = db.GetAllEvents();
    EventCollection eventCollection =
        EventResultConverter.ToEventCollection(eventResults);
    return eventCollection;
}

public override EventCollection GetEventsByDate(DateTime targetDate)
{
    StoredProcedureResult<EventResult> eventResults =
        db.GetEventsByDate(targetDate);
    EventCollection eventCollection =
        EventResultConverter.ToEventCollection(eventResults);
    return eventCollection;
}

public override DomainKey SaveEvent(Event evt)
{
    if (evt == null)
    {
        throw new ArgumentNullException("evt", "Event must be defined");
    }

    long? speakerId = null;
    if (evt.Speaker != null)

```



```
{
    speakerId = (long) evt.Speaker.ID.Value;
}

long? sponsorId = null;
if (evt.Sponsor != null)
{
    sponsorId = (long) evt.Sponsor.ID.Value;
}

long? locationId = null;
if (evt.Location != null)
{
    locationId = (long) evt.Location.ID.Value;
}

long? oldEventId = (long) evt.ID.Value;
long? newEventId = null;

db.SaveEvent(evt.Title, evt.Description, evt.MeetingDate,
    speakerId, sponsorId, locationId, oldEventId, ref newEventId);
evt.ID.Value = newEventId;

return evt.ID;
}

public override void DeleteEvent(Event evt)
{
    if (evt == null)
    {
        throw new ArgumentNullException("evt", "Event must be defined");
    }
    db.DeleteEvent((long)evt.ID.Value);
}

public override bool IsUsingLocation(Location location)
{
    if (location == null)
    {
        throw new ArgumentNullException("location");
    }

    StoredProcedureResult<CountResult> countResults =
        db.IsEventUsingLocation((long)location.ID.Value);
    foreach (CountResult countResult in countResults)
    {
        return countResult.Count > 0;
    }
}
```

```

        }
        return false;
    }

    #endregion

}
}

```

The LINQ implementation of the `EventProvider` was created as a separate assembly from the standard SQL implementations so that it can be deployed separately. The class library project for the standard SQL providers does not even need to include any of the references required by LINQ, such as `System.Data.Query`, because the new provider is loaded at runtime and the necessary namespaces for LINQ are accessible from the global assembly cache (GAC). It is convenient to isolate the LINQ-specific details to the LINQ provider class library, especially when LINQ is such a new technology. Even when Visual Studio 2008 is out, you can still target the .NET 2.0 or .NET 3.0 runtimes for the standard SQL providers, while a new LINQ provider can be built to target the .NET 3.5 runtime and fully leverage all the new dynamic language features that will be available with Visual Studio 2008.

Implementing a WCF Provider

Another provider implementation can be built by using the Windows Communication Foundation (WCF). WCF is an implementation of web services specifications that have matured dramatically over the last few years. The specifications for web services started out with SOAP and .asmx web services that offered some interoperability for multiple platforms such as .NET, Java, Python, and Ruby. But the industry set out to define many details about how web services would become a fully functional protocol for distributing applications as services with the WS-* specifications. Some of these specifications related to routing requests, while others addressed how security should be handled. Microsoft produced the Web Service Extensions (WSE), which added functionality that fell in line with these constantly changing specifications to the point that they became relatively stable. Then Microsoft released .NET 3.0 with WCF, which is a rich implementation of the WS-* specifications meant for web services, but the intention is not just for making .NET applications integrate well with other platforms. WCF will eventually replace .NET remoting and COM.

WCF will give us a great deal of flexibility that is not possible with other technologies. A WCF service could be self-hosted within that application that uses the services, or the service could be running outside of the application on the same server or a remote server. Making the change can be done entirely through the configuration. By creating a WCF implementation of a provider, we will have the option to move the data access layer to a service that runs locally or remotely on a separate server, which effectively distributes the load and traffic coming into the application.

Another benefit of using WCF is that you can provide a rich application layer that can be used by multiple applications simultaneously. Consider a provider that manages the payroll system, which needs to be accessible by multiple applications. If you had three distinct websites that used that provider, you would not automatically share the caching functionality from one website to the next because they are running in separate runtimes. But if you were to

implement the provider with WCF and point each of the three websites to the shared service, you would benefit from having a central point where data can be effectively cached, because you can control where data is saved and deleted and remove items from the cache when you determine they have changed.

You could actually be much more aggressive with your caching policy with a shared service because you do control every piece of data that is coming and going. It is much like using the stored procedures as the gatekeepers, but you have access to the full functionality of the .NET runtime. And instead of using ASP.NET caching, you could simply hold on to your own collections of data, keep them current as the data changes, and return the requested data from your internal collection without querying the database each time.

WCF Service Requirements

Running a WCF service requires a client and a service. It is a bit more work than just setting up a set of classes that can automatically talk to each other. In order for a client to work, the service must be hosted and ready. The service also needs to be at a known location that is set by using the configuration. For the WCF provider, we will create a new *EventProvider* and we will set the service to run inside of a console application during development on a local port by using the binding called *netTcpBinding*. The client configuration is shown in Listing 10-28.

Listing 10-28. WCF Client Configuration

```
<system.serviceModel>
  <client>
    <endpoint
      address = "net.tcp://localhost:8002/EventService/"
      binding = "netTcpBinding"
      contract = "DotnetUserGroup.DataAccess.WcfProviders.Events.IEventService"
    />
  </client>
</system.serviceModel>
```

The client will look for the service by using the *netTcpBinding* on port 8002 on the same machine. Then in the console application, the service will use the configuration in Listing 10-29.

Listing 10-29. WCF Service Configuration

```
<system.serviceModel>
  <services>
    <service name = "DotnetUserGroup.DataAccess.WcfProviders.Events.EventService">
      <endpoint
        address = "net.tcp://localhost:8002/EventService/"
        binding = "netTcpBinding"
        contract = "DotnetUserGroup.DataAccess.WcfProviders.Events.IEventService"
      />
    </service>
  </services>
</system.serviceModel>
```

The service element defines the concrete class, `EventService`, that implements the interface, `IEventService`, defined by the contract attribute. The interface is defined explicitly because it is used to generate the WSDL file that describes the web service. If it is not defined, the interface is not a part of the web service. It is clearly much easier to create an interface class than to create the WSDL file yourself, but you do need to decorate the interface with attributes to assist with generating the WSDL output. Listing 10-30 shows the `IEventService` interface.

Listing 10-30. *IEventService Interface*

```
[ServiceContract(Namespace = "http://dug/events/")]
[ServiceKnownType(typeof(DomainKey))]
[ServiceKnownType(typeof(Event))]
[ServiceKnownType(typeof(Location))]
[ServiceKnownType(typeof(Speaker))]
public interface IEventService
{

    [OperationContract]
    Event GetNewEvent();

    [OperationContract]
    Event GetEvent(DomainKey key);

    [OperationContract]
    EventCollection GetAllEvents();

    [OperationContract]
    EventCollection GetEventsByDate(DateTime targetDate);

    [OperationContract]
    DomainKey SaveEvent(Event evt);

    [OperationContract]
    void DeleteEvent(Event evt);

    [OperationContract]
    bool IsUsingLocation(Location location);

}
```

The first attribute that you should notice is the `ServiceKnownType`. By adding these attributes, the WSDL output will generate the data necessary to define these objects so they can be passed through the service. By explicitly declaring these types, the service will be able to pass these objects around safely. Then each method that is meant to be a part of the service is marked with the `OperationContract` attribute.

For the actual work of the WFC implementation, we will simply use the standard SQL provider, which will call the existing stored procedures. The WCF provider will act as a bridge to make the provider distributable to a remote server. But before we can call the service, we need to have the console application start up the service host.

Hosting the Service

Hosting a service can be done in any type of application. It could be a website, console application, or even a Windows service. It simply needs to start up a service host by using the concrete implementation of the service interface. For the event implementation, we will use the `EventServiceHost` class shown in Listing 10-31, which runs a singleton and provides methods to start and stop the service.

Listing 10-31. *EventServiceHost.cs*

```
namespace DotnetUserGroup.DataAccess.WcfProviders.Events
{
    public class EventServiceHost
    {
        private static readonly EventServiceHost _instance =
            new EventServiceHost();

        private ServiceHost host;

        public EventServiceHost()
        {
        }

        public static EventServiceHost Instance
        {
            get
            {
                return _instance;
            }
        }

        public void StartEventService()
        {
            Trace.WriteLine("Internal Hosting: StartEventService");
            host = new ServiceHost(typeof(EventService));
            host.Open();
        }

        public void StopEventService()
        {
            Trace.WriteLine("Internal Hosting: StopEventService");
            if (host != null)
            {
                host.Close();
                host = null;
            }
        }
    }
}
```

With the service configuration already in place, the console application can use the `EventServiceHost` class to run the service host. Listing 10-32 shows the console application.

Listing 10-32. *Program.cs*

```
namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            EventServiceHost.Instance.StartEventService();
            Console.WriteLine("Running...");
            Console.WriteLine("[ Press Enter to End ]");
            Console.ReadLine();
            EventServiceHost.Instance.StopEventService();
        }
    }
}
```

Using a console application to host a service during development is convenient because you can easily stop and start the console application as you make changes to the application. After you are ready to deploy your service to a production environment, you can run the service either through a website or a Windows service.

Defining the DataContracts

In addition to defining the operations that a service handles, it is also necessary to define the data objects that the service passes around. A `DataContract` is used to mark an object for use with a WCF service. In Listing 10-7, you may have noticed the `DataContract` and `DataMember` attributes on the `Event` class. Marking an object as a `DataContract` is much like marking it as `Serializable`. You can actually also pass around objects that are marked as `Serializable`, but it is preferable to now mark the classes as a `DataContract`.

Every bit of data that is used within a service must be prepared so that it can be safely passed around with the service. The `Event` class must be marked as a `DataContract` as well as each of the classes that it references, such as the `Location` and `DomainKey` classes.

Configuring the Provider

The configuration for the new WCF provider is now split into two parts: the client and the service. The client will be the application, such as the website, which will use the WCF provider to communicate with the service. Then the service will use the SQL implementation of the `EventProvider` to carry out the service operations. Listing 10-33 shows the client configuration, and Listing 10-34 shows the service configuration.

Listing 10-33. *Client Configuration*

```
<events defaultProvider="SqlEventProvider"
        locationProvider="SqlLocationProvider"
```

```

        speakerProvider="SqlSpeakerProvider"
        sponsorProvider="SqlSponsorProvider">
    <providers>
        <clear />
        <add name="WcfEventProvider"
            type="DotnetUserGroup.DataAccess.WcfProviders.Events.WcfEventProvider, ➤
DotnetUserGroup.DataAccess.WcfProviders" />
    </providers>
</events>

```

Listing 10-34. *Service Configuration*

```

<events defaultProvider="SqlEventProvider"
    locationProvider="SqlLocationProvider"
    speakerProvider="SqlSpeakerProvider"
    sponsorProvider="SqlSponsorProvider">
    <providers>
        <clear />
        <add name="SqlEventProvider" connectionStringName="dug"
            type="DotnetUserGroup.DataAccess.Events.SqlEventProvider, ➤
DotnetUserGroup.DataAccess" />
    </providers>
</events>

```

Using the Providers

Connecting the data to the website means connecting the providers to the pages and controls. As I have covered frequently, the `ObjectDataSource` will act as a bridge to bring these two parts together. But there is one additional step required because an `ObjectDataSource` requires a default constructor on the type it uses to populate the databound control with data items. Because the providers are initialized without a default constructor, we will need to create a proxy class.

What makes the most sense here is to create a tiny class that simply has the same methods as the provider and to mark that class as a `DataObject`. The `DataObject` is covered in greater detail in Chapter 7. Then the methods that return data can be marked as a `DataObjectMethod` of the type `Select`. Listing 10-35 shows the `EventDataObject`.

Listing 10-35. *EventDataObject.cs*

```

namespace DotnetUserGroup.DataAccess.Events
{
    [DataObject()]
    public class EventDataObject
    {

        #region " Data Methods "
    }
}

```

```

public Event GetNewEvent()
{
    return Provider.GetNewEvent();
}

[DataObjectMethod(DataObjectMethodType.Select)]
public Event GetEvent(DomainKey key)
{
    return Provider.GetEvent(key);
}

[DataObjectMethod(DataObjectMethodType.Select)]
public EventCollection GetAllEvents()
{
    return Provider.GetAllEvents();
}

[DataObjectMethod(DataObjectMethodType.Select)]
public EventCollection GetEventsByDate(DateTime targetDate)
{
    return Provider.GetEventsByDate(targetDate);
}

[DataObjectMethod(DataObjectMethodType.Update)]
public DomainKey SaveEvent(Event evt)
{
    return Provider.SaveEvent(evt);
}

[DataObjectMethod(DataObjectMethodType.Delete)]
public void DeleteEvent(Event evt)
{
    Provider.DeleteEvent(evt);
}

public bool IsUsingLocation(Location location)
{
    return Provider.IsUsingLocation(location);
}

#endregion

#region " Provider Properties "

public EventProvider Provider
{
    get
    {

```



```

        return EventManager.GetProvider(ProviderName);
    }
}

private string _providerName = String.Empty;

public string ProviderName
{
    get
    {
        return _providerName;
    }
    set
    {
        _providerName = value;
    }
}

#endregion
}
}

```

The `EventDataObject` uses a property called `Provider` that is simply a wrapper to get `EventManager`, which returns the named provider. Because the default constructor will be used with an `ObjectDataSource`, the value returned from the `ProviderName` property will always return a null value. When the `EventManager` is given a null value, it assumes the default provider is appropriate and the `EventDataObject` is able to function. If you wanted to use a different provider other than the default, you would have to programmatically instantiate either the `EventDataObject` and set the `ProviderName` property or directly use the `EventManager` to get access to the provider you want.

Finally, we can declare the `ObjectDataSource` and bind it to a databound control, such as a `Repeater`. Listing 10-36 shows the event listing in a user control.

Listing 10-36. *EventListingControl.ascx*

```

<%@ Control Language="C#" AutoEventWireup="true"
    CodeFile="EventListingControl.ascx.cs"
    Inherits="Controls_EventListingControl" %>

<h1>Event Listing</h1>

<asp:Repeater ID="rptEvents" runat="server" DataSourceID="odsEventsByDate" ➡
    OnItemDataBound="rptEvents_ItemDataBound">

    <HeaderTemplate>
    <hr />
    </HeaderTemplate>

```

```

<ItemTemplate>
<b><asp:Literal ID="ltTitle" runat="server" Text='<%# Bind("Title") %>'>
</asp:Literal></b><br />
<asp:Literal ID="ltMeetingDate" runat="server" Text="9/9/9999"></asp:Literal><br />
<p>
<asp:Literal ID="ltDescription" runat="server"
    Text='<%# Bind("Description") %>'>
</asp:Literal>
</p>
</ItemTemplate>

<SeparatorTemplate>
<hr />
</SeparatorTemplate>

<FooterTemplate>
<hr />
</FooterTemplate>

</asp:Repeater>

<asp:ObjectDataSource ID="odsEventsByDate" runat="server"
    OldValuesParameterFormatString="original_{0}"
    SelectMethod="GetEventsByDate"
    TypeName="DotnetUserGroup.DataAccess.Events.EventDataObject"
    OnSelecting="ObjectDataSource1_Selecting">
    <SelectParameters>
        <asp:Parameter
            Name="targetDate"
            Type="DateTime"
            DefaultValue="01/01/1754" />
    </SelectParameters>
</asp:ObjectDataSource>

```

With the event listing declared as a user control, it can be easily dropped into any page on your website. Most of the time you may not even need to add any code to the code-behind because you have safely isolated all of the work in the class library so that it is reusable across websites and other types of applications. In the case of the method used for the event listing, you will want to set the date to something more useful than the default of 1/1/1754. That default will show every event before today. The method itself ultimately calls a stored procedure that returns any event between now and the given date, so you can also go forward in time. The code-behind shown in Listing 10-37 sets that target date to 90 days into the future to show all upcoming meetings for the next three months. The code-behind also handles the `ItemDataBound` event for the `Repeater` so the meeting date can be set to the short data format.

Listing 10-37. *EventListingControl.ascx.cs*

```

public partial class Controls_EventListingControl : UserControl
{
    protected void ObjectDataSource1_Selecting(object sender,
        ObjectDataSourceSelectingEventArgs e)
    {
        e.InputParameters["targetDate"] = DateTime.Now.AddDays(90);
    }

    protected void rptEvents_ItemDataBound(
        object sender, RepeaterItemEventArgs e)
    {
        if (e.Item.DataItem != null)
        {
            Event evt = e.Item.DataItem as Event;
            if (evt != null)
            {
                Literal ltMeetingDate =
                    e.Item.FindControl("ltMeetingDate") as Literal;
                if (ltMeetingDate != null)
                {
                    ltMeetingDate.Text = evt.MeetingDate.ToShortDateString();
                }
            }
        }
    }
}

```

Summary

This chapter covered the big concerns, such as scalability and performance, that continue to drive the need to seek out techniques to make websites faster in high-load situations. I explained several scenarios where scalability intersects with performance and demands that you implement a plan to distribute the content and load in a way that is manageable from the perspectives of the network, web servers, and database servers. Having the ability to change the distribution of the data and the software with simple configuration changes will enable a highly agile environment that can adapt to your growing needs so you can address performance issues when they start to reveal themselves.



Photo Album

This appendix serves as a reference for the Photo Album provider covered in Chapter 5.

Photo Album Provider Configuration

Listing A-1. Configuration

```
<?xml version="1.0"?>
<configuration>
  <configSections>
    <section name="photoAlbumService" ➡
type="Chapter05.PhotoAlbumProvider.PhotoAlbumSection, ➡
Chapter05.PhotoAlbumProvider"/>
  </configSections>
  <appSettings>
    <!-- Flickr Settings http://www.flickr.com/services/api/ -->
    <add key="FlickrFeedUrlFormat" ➡
value="http://api.flickr.com/services/feeds/photos_public.gne?tags={0} ➡
&amp;format={1}"/>
  </appSettings>
  <connectionStrings>
    <add name="chpt5" connectionString="Data Source=.\SQLEXPRESS; ➡
Initial Catalog=Chapter05;Integrated Security=True" ➡
providerName="System.Data.SqlClient"/>
  </connectionStrings>
  <system.web>
    <!-- Web Settings-->
  </system.web>
  <photoAlbumService defaultProvider="SqlPhotoAlbumProvider">
    <providers>
      <clear/>
      <add name="SqlPhotoAlbumProvider" connectionStringName="chpt5" ➡
type="Chapter05.PhotoAlbumProvider.SqlPhotoAlbumProvider, ➡
Chapter05.PhotoAlbumProvider"/>
    </providers>
  </photoAlbumService>
</configuration>
```

```

    </providers>
  </photoAlbumService>
</configuration>

```

Classes

Listing A-2. *PhotoAlbumProvider.cs*

```

using System;
using System.Collections.Generic;
using System.Configuration.Provider;

namespace Chapter05.PhotoAlbumProvider
{
    /// <summary>
    /// Photo Album Provider
    /// </summary>
    public abstract class PhotoAlbumProvider : ProviderBase
    {

        #region " Abstract Methods "

        /// <summary>
        /// Gets albums for a user
        /// </summary>
        public abstract List<Album> GetAlbums(string userName);

        /// <summary>
        /// Gets photos for an album
        /// </summary>
        public abstract List<Photo> GetPhotosByAlbum(Album album);

        /// <summary>
        /// Creates an album
        /// </summary>
        public abstract Album AlbumInsert(string userName, string albumName,
            bool active, bool shared);

        /// <summary>
        /// Creates a photo
        /// </summary>
        public abstract Photo PhotoInsert(Album album, string photoName,
            DateTime photoDate, String regularUrl, int regularWidth,
            int regularHeight, String thumbnailUrl, int thumbnailWidth,
            int thumbnailHeight, bool active, bool shared);

        /// <summary>

```

```

    /// Updates an album
    /// </summary>
    public abstract void AlbumUpdate(Album album);

    /// <summary>
    /// Updates a photo
    /// </summary>
    public abstract void PhotoUpdate(Photo photo);

    /// <summary>
    /// Deletes an album
    /// </summary>
    public abstract void AlbumDeletePermanent(Album album);

    /// <summary>
    /// Deletes album permanently
    /// </summary>
    public abstract void PhotoDeletePermanent(Photo photo);

    /// <summary>
    /// Moves an album
    /// </summary>
    public abstract void AlbumMove(Album album,
        string sourceUserName, string destinationUserName);

    /// <summary>
    /// Moves a photo
    /// </summary>
    public abstract void PhotoMove(Photo photo, Album sourceAlbum,
        Album destinationAlbum);

    #endregion
}
}

```

Listing A-3. *SqlPhotoAlbumProvider.cs*

```

using System;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Configuration.Provider;
using System.Data;
using System.Data.Common;
using System.Web;
using System.Web.Configuration;
using Microsoft.Practices.EnterpriseLibrary.Data;

```

```

namespace Chapter05.PhotoAlbumProvider
{
    public class SqlPhotoAlbumProvider : PhotoAlbumProvider
    {

        #region " Variables "

        string connStringName = String.Empty;
        private Database db;

        #endregion

        #region " Implementation Methods "

        /// <summary>
        /// SQL Implementation
        /// </summary>
        public override void Initialize(string name,
        NameValueCollection config)
        {
            if (config == null)
            {
                throw new ArgumentNullException("config");
            }

            if (String.IsNullOrEmpty(name))
            {
                name = "SqlPhotoAlbumProvider";
            }

            if (String.IsNullOrEmpty(config["description"]))
            {
                config.Remove("description");
                config.Add("description", "SQL Photo Album Provider");
            }

            base.Initialize(name, config);

            if (config["connectionStringName"] == null)
            {
                throw new ProviderException(
                    "Required attribute missing: connectionStringName");
            }

            connStringName = config["connectionStringName"].ToString();
            config.Remove("connectionStringName");
        }
    }
}

```



```

        if (WebConfigurationManager.ConnectionStrings[connStringName] == null)
        {
            throw new ProviderException("Missing connection string");
        }

        db = DatabaseFactory.CreateDatabase(connStringName);

        if (config.Count > 0)
        {
            string attr = config.GetKey(0);
            if (!String.IsNullOrEmpty(attr))
            {
                throw new ProviderException("Unrecognized attribute: " + attr);
            }
        }
    }

    /// <summary>
    /// SQL Implementation
    /// </summary>
    public override List<Album> GetAlbums(string userName)
    {
        // use cache with a 5 second window
        String cacheKey = "PhotoAlbum::" + userName;
        object obj = HttpRuntime.Cache.Get(cacheKey);
        if (obj != null)
        {
            return (List<Album>)obj;
        }
        List<Album> albums = new List<Album>();

        try
        {
            using (DbCommand dbCmd =
                db.GetStoredProcCommand("pap_GetAlbumsByUserName"))
            {
                db.AddInParameter(dbCmd, "@UserName", DbType.String, userName);

                DataSet ds = db.ExecuteDataSet(dbCmd);

                // populate the album collection
                if (ds.Tables.Count > 0)
                {
                    foreach (DataRow row in ds.Tables[0].Rows)
                    {
                        Album album = new Album();
                        album.LoadDataRow(row);
                    }
                }
            }
        }
    }

```

```

        albums.Add(album);
    }
}
}
}
catch (Exception ex)
{
    HandleError("Exception with pap_GetAlbumsByUserName", ex);
}

// cache for 5 seconds
HttpRuntime.Cache.Insert(cacheKey, albums, null,
    DateTime.Now.AddSeconds(5), TimeSpan.Zero);

//return the results
return albums;
}

/// <summary>
/// SQL Implementation
/// </summary>
public override List<Photo> GetPhotosByAlbum(Album album)
{
    List<Photo> photos = new List<Photo>();

    try
    {
        using (DbCommand dbCmd =
            db.GetStoredProcCommand("pap_GetPhotosByAlbum"))
        {
            db.AddInParameter(dbCmd, "@AlbumID", DbType.Int64, album.ID);

            DataSet ds = db.ExecuteDataSet(dbCmd);

            // populate the photos collection
            if (ds.Tables.Count > 0)
            {
                foreach (DataRow row in ds.Tables[0].Rows)
                {
                    Photo photo = new Photo();
                    photo.LoadDataRow(row);
                    photo.Album = album;
                    photos.Add(photo);
                }
            }
        }
    }
}

```

```

        catch (Exception ex)
        {
            HandleError("Exception with pap_GetPhotosByAlbum", ex);
        }

        //return the results
        return photos;
    }

    /// <summary>
    /// SQL Implementation
    /// </summary>
    public override Album AlbumInsert(
        string userName, string albumName, bool active, bool shared)
    {
        try
        {
            using (DbCommand dbCmd =
                db.GetStoredProcCommand("pap_InsertAlbum"))
            {
                db.AddInParameter(dbCmd, "@UserName", DbType.String, userName);
                db.AddInParameter(dbCmd, "@Name", DbType.String, albumName);
                db.AddInParameter(dbCmd, "@IsActive", DbType.Boolean, active);
                db.AddInParameter(dbCmd, "@IsShared", DbType.Boolean, shared);

                db.AddOutParameter(dbCmd, "@AlbumID", DbType.Int64, 0);

                db.ExecuteNonQuery(dbCmd);
                long albumId = (long)db.GetParameterValue(dbCmd, "@AlbumID");
                ClearAlbumCache(userName);

                List<Album> albums = GetAlbums(userName);
                foreach (Album album in albums)
                {
                    if (album.ID == albumId)
                    {
                        return album;
                    }
                }
            }
        }
        catch (Exception ex)
        {
            HandleError("Exception with pap_InsertAlbum", ex);
        }
    }

```

```

        throw new ApplicationException("New album not found");
    }
    /// <summary>
    /// SQL Implementation
    /// </summary>
    public override Photo PhotoInsert(Album album, string photoName,
        DateTime photoDate, String regularUrl, int regularWidth,
        int regularHeight, String thumbnailUrl, int thumbnailWidth,
        int thumbnailHeight, bool active, bool shared)
    {
        try
        {
            using (DbCommand dbCmd =
                db.GetStoredProcCommand("pap_InsertPhoto"))
            {
                if (photoName.Length > 40)
                {
                    photoName = photoName.Substring(0, 40);
                }
                db.AddInParameter(dbCmd, "@AlbumID",
                    DbType.Int64, album.ID);
                db.AddInParameter(dbCmd, "@Name",
                    DbType.String, photoName);
                db.AddInParameter(dbCmd, "@PhotoDate",
                    DbType.DateTime, photoDate);
                db.AddInParameter(dbCmd, "@RegularUrl",
                    DbType.String, regularUrl);
                db.AddInParameter(dbCmd, "@RegularWidth",
                    DbType.Int32, regularWidth);
                db.AddInParameter(dbCmd, "@RegularHeight",
                    DbType.Int32, regularHeight);
                db.AddInParameter(dbCmd, "@ThumbnailUrl",
                    DbType.String, thumbnailUrl);
                db.AddInParameter(dbCmd, "@ThumbnailWidth",
                    DbType.Int32, thumbnailWidth);
                db.AddInParameter(dbCmd, "@ThumbnailHeight",
                    DbType.Int32, thumbnailHeight);
                db.AddInParameter(dbCmd, "@IsActive",
                    DbType.Boolean, active);
                db.AddInParameter(dbCmd, "@IsShared",
                    DbType.Boolean, shared);

                db.AddOutParameter(dbCmd, "@PhotoID", DbType.Int64, 0);

                db.ExecuteNonQuery(dbCmd);
                long photoId = (long)db.GetParameterValue(dbCmd, "@PhotoID");
            }
        }
    }

```

```

        List<Photo> photos = GetPhotosByAlbum(album);
        foreach (Photo photo in photos)
        {
            if (photo.ID == photoId)
            {
                return photo;
            }
        }
    }
}
catch (Exception ex)
{
    HandleError("Exception with pap_InsertPhoto", ex);
}

throw new ApplicationException("New photo not found");
}

/// <summary>
/// SQL Implementation
/// </summary>
public override void AlbumUpdate(Album album)
{
    try
    {
        using (DbCommand dbCmd =
            db.GetStoredProcCommand("pap_UpdateAlbum"))
        {
            db.AddInParameter(dbCmd, "@AlbumID",
                DbType.Int64, album.ID);
            db.AddInParameter(dbCmd, "@Name",
                DbType.String, album.Name);
            db.AddInParameter(dbCmd, "@IsActive",
                DbType.String, album.IsActive);
            db.AddInParameter(dbCmd, "@IsShared",
                DbType.String, album.IsShared);

            db.ExecuteNonQuery(dbCmd);
        }
    }
    catch (Exception ex)
    {
        HandleError("Exception with pap_UpdateAlbum", ex);
    }
}

/// <summary>

```

```

    /// SQL Implementation
    /// </summary>
    public override void PhotoUpdate(Photo photo)
    {
        try
        {
            using (DbCommand dbCmd =
                db.GetStoredProcCommand("pap_UpdatePhoto"))
            {
                db.AddInParameter(dbCmd, "@PhotoID",
                    DbType.Int64, photo.ID);
                db.AddInParameter(dbCmd, "@Name",
                    DbType.String, photo.Name);
                db.AddInParameter(dbCmd, "@PhotoDate",
                    DbType.DateTime, photo.PhotoDate);
                db.AddInParameter(dbCmd, "@RegularUrl",
                    DbType.String, photo.RegularUrl);
                db.AddInParameter(dbCmd, "@RegularWidth",
                    DbType.Int32, photo.RegularWidth);
                db.AddInParameter(dbCmd, "@RegularHeight",
                    DbType.Int32, photo.RegularHeight);
                db.AddInParameter(dbCmd, "@ThumbnailUrl",
                    DbType.String, photo.ThumbnailUrl);
                db.AddInParameter(dbCmd, "@ThumbnailWidth",
                    DbType.Int32, photo.ThumbnailWidth);
                db.AddInParameter(dbCmd, "@ThumbnailHeight",
                    DbType.Int32, photo.ThumbnailHeight);
                db.AddInParameter(dbCmd, "@IsActive",
                    DbType.Boolean, photo.IsActive);
                db.AddInParameter(dbCmd, "@IsShared",
                    DbType.Boolean, photo.IsShared);

                db.ExecuteNonQuery(dbCmd);
                ClearAlbumCache(photo.Album.UserName);
            }
        }
        catch (Exception ex)
        {
            HandleError("Exception with pap_UpdatePhoto", ex);
        }
    }

    /// <summary>
    /// SQL Implementation
    /// </summary>
    /// <param name="album"></param>
    public override void AlbumDeletePermanent(Album album)

```

```

{
    try
    {
        using (DbCommand dbCmd =
            db.GetStoredProcCommand("pap_DeleteAlbum"))
        {
            db.AddInParameter(dbCmd, "@AlbumID", DbType.Int64, album.ID);

            db.ExecuteNonQuery(dbCmd);
            ClearAlbumCache(album.UserName);
        }
    }
    catch (Exception ex)
    {
        HandleError("Exception with pap_DeleteAlbum", ex);
    }
}

/// <summary>
/// SQL Implementation
/// </summary>
public override void PhotoDeletePermanent(Photo photo)
{
    try
    {
        using (DbCommand dbCmd =
            db.GetStoredProcCommand("pap_DeletePhoto"))
        {
            db.AddInParameter(dbCmd, "@PhotoID", DbType.Int64, photo.ID);

            db.ExecuteNonQuery(dbCmd);
            ClearAlbumCache(photo.Album.UserName);
        }
    }
    catch (Exception ex)
    {
        HandleError("Exception with pap_DeletePhoto", ex);
    }
}

/// <summary>
/// SQL Implementation
/// </summary>
public override void AlbumMove(Album album, string sourceUserName,
string destinationUserName)
{
    try

```

```

    {
        using (DbCommand dbCmd = db.GetStoredProcCommand("pap_MoveAlbum"))
        {
            db.AddInParameter(dbCmd, "@AlbumID",
                DbType.Int64, album.ID);
            db.AddInParameter(dbCmd, "@SourceUserName",
                DbType.String, sourceUserName);
            db.AddInParameter(dbCmd, "@DestinationUserName",
                DbType.String, destinationUserName);

            db.ExecuteNonQuery(dbCmd);
            ClearAlbumCache(sourceUserName);
            ClearAlbumCache(destinationUserName);
        }
    }
    catch (Exception ex)
    {
        HandleError("Exception with pap_MoveAlbum", ex);
    }
}

/// <summary>
/// SQL Implementation
/// </summary>
public override void PhotoMove(
    Photo photo, Album sourceAlbum, Album destinationAlbum)
{
    try
    {
        using (DbCommand dbCmd = db.GetStoredProcCommand("pap_MovePhoto"))
        {
            db.AddInParameter(dbCmd, "@PhotoID",
                DbType.Int64, photo.ID);
            db.AddInParameter(dbCmd, "@SourceAlbumID",
                DbType.Int64, sourceAlbum.ID);
            db.AddInParameter(dbCmd, "@DestinationAlbumID",
                DbType.Int64, destinationAlbum.ID);

            db.ExecuteNonQuery(dbCmd);
            sourceAlbum.ClearPhotos();
            ClearAlbumCache(sourceAlbum.UserName);
            destinationAlbum.ClearPhotos();
            ClearAlbumCache(destinationAlbum.UserName);
        }
    }
    catch (Exception ex)
    {

```



```

        HandleError("Exception with pap_MoveAlbum", ex);
    }
}

#endregion

#region " Utility Methods "

private void HandleError(string message, Exception ex)
{
    //TODO log the error
    throw new ApplicationException(message, ex);
}

/// <summary>
/// SQL Implementation
/// </summary>
public void ClearAlbumCache(String userName)
{
    String cacheKey = "PhotoAlbum:." + userName;
    if (HttpRuntime.Cache.Get(cacheKey) != null)
    {
        HttpRuntime.Cache.Remove(cacheKey);
    }
}

#endregion

}
}

```

Listing A-4. *DataObject.cs*

```

using System;
using System.Data;
using System.Reflection;

namespace Chapter05.PhotoAlbumProvider
{
    /// <summary>
    /// DataObject
    /// </summary>
    public abstract class DataObject : IComparable
    {
        /// <summary>
        /// Default Datetime
        /// </summary>

```

```

public readonly static DateTime DefaultDatetime =
    DateTime.Parse("01/01/1754");

/// <summary>
/// Object ID
/// </summary>
public abstract long ID { get; set; }

/// <summary>
/// Loads the values from the DataRow and works to match up column names
/// with Property names. For example <code>row["Name"] = this.Name</code>
/// and <code>row["IsActive"] = this.IsActive</code> where String typed
/// properties are treated as strings and other types are treated properly.
/// Supported types include String, Boolean, Float, Int and DateTime.
/// The Property must also be set as public, not protected and also be
/// writeable.
/// </summary>
/// <param name="row"></param>
protected internal void LoadDataRow(DataRow row)
{
    Type type = GetType();
    foreach (PropertyInfo pi in type.GetProperties())
    {
        if (pi.CanWrite)
        {
            if (pi.PropertyType.Equals(typeof(DateTime)))
            {
                if (row[pi.Name] != null)
                {
                    pi.SetValue(this,
                        GetNotNullDateTime(row, pi.Name), null);
                }
            }
            else if (pi.PropertyType.Equals(typeof(Boolean)))
            {
                if (row[pi.Name] != null)
                {
                    if ("1".Equals(row[pi.Name]))
                    {
                        pi.SetValue(this, true, null);
                    }
                    else
                    {
                        pi.SetValue(this, false, null);
                    }
                }
            }
        }
    }
}

```

```

else if (pi.PropertyType.Equals(typeof(float)))
{
    if (row[pi.Name] != null)
    {
        pi.SetValue(this, GetNotNullFloat(row, pi.Name), null);
    }
}
else if (pi.PropertyType.Equals(typeof(String)))
{
    if (row[pi.Name] != null)
    {
        pi.SetValue(this,
            GetNotNullString(row, pi.Name), null);
    }
}
else if (pi.PropertyType.Equals(typeof(int)))
{
    if (row[pi.Name] != null)
    {
        pi.SetValue(this,
            GetNotNullInteger(row, pi.Name), null);
    }
}
else if (pi.PropertyType.Equals(typeof(Int64)))
{
    if (row[pi.Name] != null)
    {
        pi.SetValue(this, GetNotNullLong(row, pi.Name), null);
    }
}
}
}

}

/// <summary>
/// Utility Method
/// </summary>
protected internal DateTime GetNotNullDateTime(DataRow row, String name)
{
    Object obj = row[name];
    if (!DBNull.Value.Equals(obj))
    {
        return (DateTime)obj;
    }
    else
    {
        return DefaultDatetime;
    }
}

```

```

    }
}

/// <summary>
/// Utility Method
/// </summary>
protected internal String GetNotNullString(DataRow row, String name)
{
    Object obj = row[name];
    if (!DBNull.Value.Equals(obj))
    {
        return (String)obj;
    }
    else
    {
        return String.Empty;
    }
}

/// <summary>
/// Utility Method
/// </summary>
protected internal int GetNotNullInteger(DataRow row, String name)
{
    Object obj = row[name];
    if (!DBNull.Value.Equals(obj) && obj is Int32)
    {
        return (int)obj;
    }
    else
    {
        return 0;
    }
}

/// <summary>
/// Utility Method
/// </summary>
protected internal long GetNotNullLong(DataRow row, String name)
{
    Object obj = row[name];
    if (!DBNull.Value.Equals(obj) && obj is Int64)
    {
        return (long)obj;
    }
    else
    {

```

```
        return 0;
    }
}

/// <summary>
/// Utility Method
/// </summary>
protected internal float GetNotNullFloat(DataRow row, String name)
{
    Object obj = row[name];
    if (!DBNull.Value.Equals(obj) && obj is float)
    {
        return (float)obj;
    }
    else
    {
        return 0;
    }
}

private DateTime _created = DefaultDatetime;
/// <summary>
/// Object creation time
/// </summary>
public DateTime Created
{
    get
    {
        return _created;
    }
    set
    {
        _created = value;
    }
}

private DateTime _modified = DefaultDatetime;
/// <summary>
/// Object modified time
/// </summary>
public DateTime Modified
{
    get
    {
        return _modified;
    }
    set
```

```

        {
            _modified = value;
        }
    }

    /// <summary>
    /// Base override
    /// </summary>
    public int CompareTo(Object obj)
    {
        int result = 0;
        if (obj is DataObject)
        {
            DataObject other = (DataObject) obj;
            result = Created.CompareTo(other.Created) * (-1);
        }
        return result;
    }

    /// <summary>
    /// Base override
    /// </summary>
    public override int GetHashCode()
    {
        return ID.GetHashCode();
    }

    /// <summary>
    /// Base override
    /// </summary>
    public override bool Equals(object obj)
    {
        if (obj is DataObject) {
            if (ID.Equals(((DataObject)obj).ID))
            {
                return true;
            }
        }
        return false;
    }
}
}

```

Listing A-5. *Album.cs*

```

using System;
using System.Collections.Generic;

```

```

namespace Chapter05.PhotoAlbumProvider
{
    /// <summary>
    /// Album
    /// </summary>
    public class Album : DataObject
    {

        #region " Properties "

        private long _id = 0;
        /// <summary>
        /// Object ID
        /// </summary>
        public override long ID
        {
            get
            {
                return _id;
            }
            set
            {
                _id = value;
            }
        }

        private string _username;
        /// <summary>
        /// Owner's username
        /// </summary>
        public string UserName
        {
            get
            {
                return _username;
            }
            set
            {
                _username = value;
            }
        }

        private string _name;
        /// <summary>
        /// Album Name
        /// </summary>
        public string Name

```

```

    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    private bool _active = true;
    /// <summary>
    /// Indicates if an album is active
    /// </summary>
    public bool IsActive
    {
        get
        {
            return _active;
        }
        set
        {
            _active = value;
        }
    }

    private bool _shared = true;
    /// <summary>
    /// Indicates if an album is shared
    /// </summary>
    public bool IsShared
    {
        get
        {
            return _shared;
        }
        set
        {
            _shared = value;
        }
    }

    private List<Photo> _photos = null;
    /// <summary>
    /// Photos in album
    /// </summary>

```



```

    public List<Photo> Photos
    {
        get
        {
            if (_photos == null)
            {
                _photos = PhotoAlbumService.Instance.GetPhotosByAlbum(this);
            }
            return _photos;
        }
    }

#endregion

#region " Methods "

    /// <summary>
    /// Clear Photos
    /// </summary>
    protected internal void ClearPhotos()
    {
        _photos = null;
    }

    /// <summary>
    /// Base override
    /// </summary>
    public override String ToString()
    {
        return Name;
    }

#endregion

}
}

```

Listing A-6. *Photo.cs*

```

using System;

namespace Chapter05.PhotoAlbumProvider
{
    /// <summary>
    /// Photo
    /// </summary>
    public class Photo : DataObject
    {

```

```

#region " Properties "

private long _id = 0;
/// <summary>
/// Object ID
/// </summary>
public override long ID
{
    get
    {
        return _id;
    }
    set
    {
        _id = value;
    }
}

private string _name;
/// <summary>
/// Name of photo
/// </summary>
public string Name
{
    get
    {
        return _name;
    }
    set
    {
        _name = value;
    }
}

private DateTime _photoDate = DefaultDatetime;
/// <summary>
/// Date photo was taken
/// </summary>
public DateTime PhotoDate
{
    get
    {
        return _photoDate;
    }
    set
    {
        _photoDate = value;
    }
}

```

```
    }  
}  
  
private bool _active = true;  
/// <summary>  
/// Indicates if photo is active  
/// </summary>  
public bool IsActive  
{  
    get  
    {  
        return _active;  
    }  
    set  
    {  
        _active = value;  
    }  
}  
  
private bool _shared = true;  
/// <summary>  
/// Indicates if photo is shared  
/// </summary>  
public bool IsShared  
{  
    get  
    {  
        return _shared;  
    }  
    set  
    {  
        _shared = value;  
    }  
}  
  
private string _regularUrl = String.Empty;  
/// <summary>  
/// Url for photo  
/// </summary>  
public string RegularUrl  
{  
    get  
    {  
        return _regularUrl;  
    }  
    set  
    {
```

```

        _regularUrl = value;
    }
}

private int _regularWidth = 0;
/// <summary>
/// Width for photo
/// </summary>
public int RegularWidth
{
    get
    {
        return _regularWidth;
    }
    set
    {
        _regularWidth = value;
    }
}

private int _regularHeight = 0;
/// <summary>
/// Height for photo
/// </summary>
public int RegularHeight
{
    get
    {
        return _regularHeight;
    }
    set
    {
        _regularHeight = value;
    }
}

private string _thumbnailUrl = String.Empty;
/// <summary>
/// Thumbnail url for photo
/// </summary>
public string ThumbnailUrl
{
    get
    {
        return _thumbnailUrl;
    }
    set

```

```
        {
            _thumbnailUrl = value;
        }
    }

    private int _thumbnailWidth = 0;
    /// <summary>
    /// Thumbnail width for photo
    /// </summary>
    public int ThumbnailWidth
    {
        get
        {
            return _thumbnailWidth;
        }
        set
        {
            _thumbnailWidth = value;
        }
    }

    private int _thumbnailHeight = 0;
    /// <summary>
    /// Thumbnail height for photo
    /// </summary>
    public int ThumbnailHeight
    {
        get
        {
            return _thumbnailHeight;
        }
        set
        {
            _thumbnailHeight = value;
        }
    }

    private Album _album = null;
    /// <summary>
    /// Album which holds this photo
    /// </summary>
    public Album Album
    {
        get
        {
            return _album;
        }
    }
}
```

```

        set
        {
            // should be set when pulled by album
            _album = value;
        }
    }

    #endregion

}
}

```

Listing A-7. *PhotoAlbumService.cs*

```

using System.Configuration.Provider;
using System.Web.Configuration;

namespace Chapter05.PhotoAlbumProvider
{
    public class PhotoAlbumService
    {
        private static PhotoAlbumProvider _defaultProvider = null;
        private static PhotoAlbumProviderCollection _providers = null;
        private static object _lock = new object();

        private PhotoAlbumService() {}

        public PhotoAlbumProvider DefaultProvider
        {
            get { return _defaultProvider; }
        }

        public PhotoAlbumProvider GetProvider(string name)
        {
            return _providers[name];
        }

        public static PhotoAlbumProvider Instance
        {
            get
            {
                LoadProviders();
                return _defaultProvider;
            }
        }

        private static void LoadProviders()
        {

```



```

        get { return (ProviderSettingsCollection)base["providers"]; }
    }

    [StringValidator(MinLength = 1)]
    [ConfigurationProperty("defaultProvider",
        DefaultValue = "SqlPhotoAlbumProvider")]
    public string DefaultProvider
    {
        get { return (string)base["defaultProvider"]; }
        set { base["defaultProvider"] = value; }
    }
}
}

```

Listing A-9. *PhotoAlbumProviderCollection.cs*

```

using System;
using System.Configuration.Provider;

namespace Chapter05.PhotoAlbumProvider
{
    public class PhotoAlbumProviderCollection : ProviderCollection
    {
        public new PhotoAlbumProvider this[string name]
        {
            get { return (PhotoAlbumProvider)base[name]; }
        }

        public override void Add(ProviderBase provider)
        {
            if (provider == null)
                throw new ArgumentNullException("provider");

            if (!(provider is PhotoAlbumProvider))
                throw new ArgumentException
                    ("Invalid provider type", "provider");

            base.Add(provider);
        }
    }
}

```


Table Scripts

Listing A-10. *pap_Albums.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND name = 'pap_Albums')
    BEGIN
        PRINT 'Dropping Table pap_Albums'
        DROP Table pap_Albums
    END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON
GO
CREATE TABLE [dbo].[pap_Albums](
    [ID] [bigint] IDENTITY(1,1) NOT NULL,
    [UserName] [nvarchar](256) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Name] [varchar](50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [IsActive] [char](1) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [IsShared] [char](1) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Modified] [datetime] NOT NULL,
    [Created] [datetime] NOT NULL,
    CONSTRAINT [PK_pap_Albums] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF

```

Listing A-11. *pap_Photos.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND name = 'pap_Photos')
    BEGIN
        PRINT 'Dropping Table pap_Photos'
        DROP Table pap_Photos
    END
GO

SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_PADDING ON

```

```

GO
CREATE TABLE [dbo].[pap_Photos](
    [ID] [bigint] IDENTITY(1,1) NOT NULL,
    [AlbumID] [bigint] NOT NULL,
    [Name] [nvarchar](60) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [PhotoDate] [datetime] NULL,
    [RegularUrl] [nvarchar](200) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [RegularWidth] [int] NOT NULL,
    [RegularHeight] [int] NOT NULL,
    [ThumbnailUrl] [nvarchar](200) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [ThumbnailWidth] [int] NOT NULL,
    [ThumbnailHeight] [int] NOT NULL,
    [IsActive] [char](1) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [IsShared] [char](1) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Modified] [datetime] NOT NULL,
    [Created] [datetime] NOT NULL,
CONSTRAINT [PK_pap_Photos] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]

GO
SET ANSI_PADDING OFF
GO

```

Constraints Scripts

Listing A-12. *AddConstraints.sql*

```

IF EXISTS (select * from dbo.sysobjects where id =
    object_id(N'[dbo].[FK_pap_Photos_pap_Albums]') and
    OBJECTPROPERTY(id, N'IsForeignKey') = 1)
ALTER TABLE [dbo].[pap_Photos] DROP CONSTRAINT FK_pap_Photos_pap_Albums
GO

ALTER TABLE [dbo].[pap_Photos] WITH CHECK ADD
    CONSTRAINT [FK_pap_Photos_pap_Albums] FOREIGN KEY([AlbumID])
REFERENCES [dbo].[pap_Albums] ([id])
GO

```

Listing A-13. *DropConstraints.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'F'
    AND name = 'FK_pap_Photos_pap_Albums')
BEGIN
    print 'Dropping FK_pap_Photos_pap_Albums'

```

```
ALTER TABLE dbo.pap_Photos
    DROP CONSTRAINT FK_pap_Photos_pap_Albums
END
GO
```

Stored Procedure Scripts

Listing A-14. *pap_DeleteAlbum.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'pap_DeleteAlbum')
    BEGIN
        DROP Procedure pap_DeleteAlbum
    END
GO

CREATE Procedure dbo.pap_DeleteAlbum
(
    @AlbumID bigint
)
AS

/* Assume child dependencies are deleted by provider */
DELETE FROM pap_Albums WHERE ID = @AlbumID

GO

GRANT EXEC ON pap_DeleteAlbum TO PUBLIC
GO
```

Listing A-15. *pap_DeletePhoto.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'pap_DeletePhoto')
    BEGIN
        DROP Procedure pap_DeletePhoto
    END
GO

CREATE Procedure dbo.pap_DeletePhoto
(
    @PhotoID bigint
)
AS

/* assume child dependencies have been deleted by provider */
```

```
DELETE FROM pap_Photos WHERE ID = @PhotoID
```

```
GO
```

```
GRANT EXEC ON pap_DeletePhoto TO PUBLIC
```

```
GO
```

Listing A-16. *pap_GetAlbumsByUserName.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
  AND name = 'pap_GetAlbumsByUsername')
  BEGIN
    DROP Procedure pap_GetAlbumsByUsername
  END
```

```
GO
```

```
CREATE Procedure dbo.pap_GetAlbumsByUserName
(
  @UserName [nvarchar](256)
)
AS
```

```
SELECT * FROM pap_Albums
WHERE UserName = @UserName
ORDER BY Created DESC
```

```
GO
```

```
GRANT EXEC ON pap_GetAlbumsByUsername TO PUBLIC
GO
```

Listing A-17. *pap_GetPhotosByAlbum.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
  AND name = 'pap_GetPhotosByAlbum')
  BEGIN
    DROP Procedure pap_GetPhotosByAlbum
  END
GO
```

```
CREATE Procedure dbo.pap_GetPhotosByAlbum
(
  @AlbumID bigint
)
AS
```

```
SELECT *
```

```
FROM pap_Photos
WHERE AlbumID = @AlbumID
ORDER BY Created DESC

GO

GRANT EXEC ON pap_GetPhotosByAlbum TO PUBLIC
GO
```

Listing A-18. *pap_InsertAlbum.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'pap_InsertAlbum')
BEGIN
    DROP Procedure pap_InsertAlbum
END
GO

CREATE Procedure dbo.pap_InsertAlbum
(
    @AlbumID bigint OUTPUT,
    @UserName [nvarchar](256),
    @Name [varchar](50),
    @IsActive [char](1),
    @IsShared [char](1)
)
AS

INSERT INTO pap_Albums (UserName, [Name], IsActive, IsShared, Modified, Created)
VALUES (@UserName, @Name, @IsActive, @IsShared, GETDATE(), GETDATE())

SELECT @AlbumID = @@IDENTITY

GO

GRANT EXEC ON pap_InsertAlbum TO PUBLIC
GO
```

Listing A-19. *pap_InsertPhoto.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'pap_InsertPhoto')
BEGIN
    DROP Procedure pap_InsertPhoto
END
GO

CREATE Procedure dbo.pap_InsertPhoto
```

```

(
    @PhotoID bigint OUTPUT,
    @AlbumID bigint,
    @Name [nvarchar](60),
    @PhotoDate [datetime],
    @RegularUrl [nvarchar](200),
    @RegularWidth [int],
    @RegularHeight [int],
    @ThumbnailUrl [nvarchar](200),
    @ThumbnailWidth [int],
    @ThumbnailHeight [int],
    @IsActive [char](1),
    @IsShared [char](1)
)
AS

INSERT INTO pap_Photos (
    AlbumID, [Name], PhotoDate,
    RegularUrl, RegularWidth, RegularHeight,
    ThumbnailUrl, ThumbnailWidth, ThumbnailHeight,
    IsActive, IsShared, Modified, Created
) VALUES (
    @AlbumID, @Name, @PhotoDate,
    @RegularUrl, @RegularWidth, @RegularHeight,
    @ThumbnailUrl, @ThumbnailWidth, @ThumbnailHeight,
    @IsActive, @IsShared, GETDATE(), GETDATE()
)

SELECT @PhotoID = @@IDENTITY
GO

GRANT EXEC ON pap_InsertPhoto TO PUBLIC
GO

```

Listing A-20. *pap_MoveAlbum.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'pap_MoveAlbum')
BEGIN
    DROP Procedure pap_MoveAlbum
END
GO

CREATE Procedure dbo.pap_MoveAlbum
(
    @AlbumID bigint,
    @SourceUserName [nvarchar](256),
    @DestinationUserName [nvarchar](256)

```

```
)
AS

UPDATE pap_Albums SET UserName = @DestinationUserName
WHERE UserName = @SourceUserName AND ID = @AlbumID

GO

GRANT EXEC ON pap_MoveAlbum TO PUBLIC
GO
```

Listing A-21. *pap_MovePhoto.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'pap_MovePhoto')
BEGIN
    DROP Procedure pap_MovePhoto
END
GO

CREATE Procedure dbo.pap_MovePhoto
(
    @PhotoID bigint,
    @SourceAlbumID bigint,
    @DestinationAlbumID bigint
)
AS

UPDATE pap_Photos SET AlbumID = @DestinationAlbumID
WHERE AlbumID = @SourceAlbumID AND ID = @PhotoID

GO

GRANT EXEC ON pap_MovePhoto TO PUBLIC
GO
```

Listing A-22. *pap_UpdateAlbum.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'pap_UpdateAlbum')
BEGIN
    DROP Procedure pap_UpdateAlbum
END
GO

CREATE Procedure dbo.pap_UpdateAlbum
(
    @AlbumID bigint,
```

```

        @Name [varchar](50),
        @IsActive [char](1),
        @IsShared [char](1)
    )
AS

UPDATE pap_Albums SET
    [Name] = @Name,
    IsActive = @IsActive,
    IsShared = @IsShared,
    Modified = GETDATE()
WHERE ID = @AlbumID

GO

GRANT EXEC ON pap_UpdateAlbum TO PUBLIC
GO

```

Listing A-23. *pap_UpdatePhoto.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
    AND name = 'pap_UpdatePhoto')
    BEGIN
        DROP Procedure pap_UpdatePhoto
    END
GO

CREATE Procedure dbo.pap_UpdatePhoto
(
    @PhotoID bigint,
    @Name [varchar](50),
    @PhotoDate [datetime],
    @RegularUrl [nvarchar](200),
    @RegularWidth [int],
    @RegularHeight [int],
    @ThumbnailUrl [nvarchar](200),
    @ThumbnailWidth [int],
    @ThumbnailHeight [int],
    @IsActive [char](1),
    @IsShared [char](1)
)
AS

UPDATE pap_Photos SET
    [Name] = @Name,
    PhotoDate = @PhotoDate,
    RegularUrl = @RegularUrl,
    RegularWidth = @RegularWidth,

```



```

RegularHeight = @RegularHeight,
ThumbnailUrl = @ThumbnailUrl,
ThumbnailWidth = @ThumbnailWidth,
ThumbnailHeight = @ThumbnailHeight,
IsActive = @IsActive,
IsShared = @IsShared,
Modified = GETDATE()
WHERE ID = @PhotoID

GO

GRANT EXEC ON pap_UpdatePhoto TO PUBLIC
GO

```

Website Classes

Listing A-24. *FlickrHelper.cs*

```

using System;
using System.Collections.Generic;
using System.Configuration;
using System.IO;
using System.Net;
using System.Web.Configuration;
using System.Xml;
using Chapter05.CustomSiteMapProvider;
using Chapter05.PhotoAlbumProvider;

/// <summary>
/// Summary description for FlickrHelper
/// </summary>
public class FlickrHelper
{

    public static void ImportFlickrPhotosToAlbum(Album album, string tag)
    {
        WebClient client = new WebClient();
        string url = String.Format(
            SiteConfiguration.FlickFeedUrlFormat, tag, "rss2");
        byte[] data = client.DownloadData(url);
        MemoryStream stream = new MemoryStream(data);

        XmlDocument document = new XmlDocument();
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(document.NameTable);
        nsmgr.AddNamespace("media", "http://search.yahoo.com/mrss/");
        nsmgr.AddNamespace("dc", "http://purl.org/dc/elements/1.1/");
    }
}

```

```

document.Load(stream);
document.Normalize();

int max = 10;
int count = 1;

XmlNode channelNode = document.SelectSingleNode("/rss/channel");
if (channelNode != null)
{
    XmlNodeList itemNodes = channelNode.SelectNodes("item");
    foreach (XmlNode itemNode in itemNodes)
    {
        XmlNode titleNode = itemNode.SelectSingleNode("title");
        XmlNode dateTakenNode = itemNode.SelectSingleNode(
            "dc:date.Taken", nsmgr);

        XmlNode regularUrlNode = itemNode.SelectSingleNode(
            "media:content/@url", nsmgr);
        XmlNode regularWidthNode = itemNode.SelectSingleNode(
            "media:content/@width", nsmgr);
        XmlNode regularHeightNode = itemNode.SelectSingleNode(
            "media:content/@height", nsmgr);

        XmlNode thumbnailUrlNode = itemNode.SelectSingleNode(
            "media:thumbnail/@url", nsmgr);
        XmlNode thumbnailWidthNode = itemNode.SelectSingleNode(
            "media:thumbnail/@width", nsmgr);
        XmlNode thumbnailHeightNode = itemNode.SelectSingleNode(
            "media:thumbnail/@height", nsmgr);

        try
        {
            string title = "No Title";
            DateTime dateTaken;
            string regularUrl;
            string thumbnailUrl;
            int regularWidth, regularHeight,
                thumbnailWidth, thumbnailHeight;

            if (titleNode != null && titleNode.FirstChild != null)
            {
                title = titleNode.FirstChild.Value;
            }
            DateTime.TryParse(dateTakenNode.FirstChild.Value,
                out dateTaken);
            regularUrl = regularUrlNode.Value;
            int.TryParse(regularWidthNode.Value,

```

```

        out regularWidth);
    int.TryParse(regularHeightNode.Value,
        out regularHeight);

    thumbnailUrl = thumbnailUrlNode.Value;
    int.TryParse(thumbnailWidthNode.Value,
        out thumbnailWidth);
    int.TryParse(thumbnailHeightNode.Value,
        out thumbnailHeight);

    PhotoAlbumProvider provider =
        PhotoAlbumService.Instance;
    provider.PhotoInsert(album, title,
        dateTaken, regularUrl, regularWidth,
        regularHeight, thumbnailUrl,
        thumbnailWidth, thumbnailHeight,
        true, true);
}
catch (Exception ex)
{
    Utility.LogError("Error reading RSS item", ex);
}
count++;
if (count > max) {
    break;
}
}
}
}

public static void ExportFlickrPhotos(string tag)
{
    WebClient client = new WebClient();
    string feedUrlFormat = "http://api.flickr.com/services/feeds/" +
        "photos_public.gne?tags={0}&format={1}";
    string url = String.Format(feedUrlFormat, tag, "rss2");
    byte[] data = client.DownloadData(url);
    MemoryStream stream = new MemoryStream(data);

    XmlDocument document = new XmlDocument();
    XmlNamespaceManager nsmgr = new XmlNamespaceManager(document.NameTable);
    nsmgr.AddNamespace("media", "http://search.yahoo.com/mrss/");
    nsmgr.AddNamespace("dc", "http://purl.org/dc/elements/1.1/");

    document.Load(stream);
    document.Normalize();

```

```

XmlNode channelNode = document.SelectSingleNode("/rss/channel");
if (channelNode != null)
{
    XmlNodeList itemNodes = channelNode.SelectNodes("item");
    foreach (XmlNode itemNode in itemNodes)
    {
        XmlNode titleNode =
            itemNode.SelectSingleNode("title");
        XmlNode dateTakenNode =
            itemNode.SelectSingleNode(
                "dc:date.Taken", nsmgr);
        XmlNode urlNode = itemNode.SelectSingleNode(
            "media:content/@url", nsmgr);
        XmlNode heightNode = itemNode.SelectSingleNode(
            "media:content/@height", nsmgr);
        XmlNode widthNode = itemNode.SelectSingleNode(
            "media:content/@width", nsmgr);
        XmlNode tagsNode = itemNode.SelectSingleNode(
            "media:category", nsmgr);

        string title = titleNode.FirstChild.Value;
        DateTime dateTaken;
        DateTime.TryParse(dateTakenNode.FirstChild.Value, out dateTaken);
        int width;
        int height;
        string photoUrl = urlNode.Value;
        string tags = tagsNode.InnerText;
        int.TryParse(widthNode.Value, out width);
        int.TryParse(heightNode.Value, out height);

        Console.WriteLine("title: " + title);
        Console.WriteLine("dateTaken: " + dateTaken);
        Console.WriteLine("photoUrl: " + photoUrl);
        Console.WriteLine("tags: " + tags);
        Console.WriteLine("width: " + width);
        Console.WriteLine("height: " + height);
        Console.WriteLine("title: " + title);

        //TODO use these values to download and save the file

    }
}

public static void CreateFlickrAlbums(string username)
{
    PhotoAlbumProvider provider = PhotoAlbumService.Instance;

```

```

Dictionary<String, String> defaultAlbums =
    new Dictionary<string, string>();
defaultAlbums.Add("Forest Album", "forest");
defaultAlbums.Add("Summer Album", "summer");
defaultAlbums.Add("Water Album", "water");

List<Album> albums = provider.GetAlbums(username);
foreach (Album album in albums)
{
    // start from scratch
    if (defaultAlbums.ContainsKey(album.Name))
    {
        // delete photos first
        foreach (Photo photo in album.Photos)
        {
            provider.PhotoDeletePermanent(photo);
        }
        provider.AlbumDeletePermanent(album);
    }
}

foreach (string albumName in defaultAlbums.Keys)
{
    CreateFlickrAlbums(username, albumName, defaultAlbums[albumName]);
}

RepopulateSiteMap();
}

private static void CreateFlickrAlbums(string username,
    string albumName, string flickrTag)
{
    PhotoAlbumProvider provider = PhotoAlbumService.Instance;
    Album album = provider.AlbumInsert(username, albumName, true, true);
    ImportFlickrPhotosToAlbum(album, flickrTag);
}

private static void DeleteFlickAlbum(string username,
    string albumName)
{
    PhotoAlbumProvider provider = PhotoAlbumService.Instance;
    List<Album> albums = provider.GetAlbums(username);
    foreach (Album album in albums)
    {
        // start from scratch
        if (album.Name.Equals(albumName))
        {

```

```

        // delete photos first
        foreach (Photo photo in album.Photos)
        {
            provider.PhotoDeletePermanent(photo);
        }
        provider.AlbumDeletePermanent(album);
    }
}

public static void RemoveFlickAlbum(string username,
    string albumName)
{
    DeleteFlickAlbum(username, albumName);
    RepopulateSiteMap();
}

public static void AddFlickrAlbum(string username,
    string albumName, string flickrTag)
{
    CreateFlickrAlbums(username, albumName, flickrTag);
    RepopulateSiteMap();
}

private static void RepopulateSiteMap()
{
    string connStringName = GetSqlSiteMapConnectionString();
    SqlSiteMapHelper helper = new SqlSiteMapHelper(connStringName);
    helper.RepopulateSiteMapNodes();
}

public static string GetSqlSiteMapConnectionString()
{
    string connStringName = String.Empty;
    SiteMapSection siteMapSection =
        ConfigurationManager.GetSection("system.web/siteMap")
        as SiteMapSection;
    if (siteMapSection != null)
    {
        string defaultProvider = siteMapSection.DefaultProvider;
        connStringName =
            siteMapSection.Providers[defaultProvider].
            Parameters["connectionStringName"];
    }
    return connStringName;
}
}

```

SQL SiteMap Provider

Classes

Listing A-25. *SqlSiteMapProvider.cs*

```
// Derived from the PlainTextSiteMapProvider example on MSDN
// http://msdn2.microsoft.com/en-us/library/system.web.sitemapprovider.aspx
```

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Collections.Specialized;
using System.Data;
using System.Data.Common;
using System.Security.Permissions;
using System.Web;
using System.Web.Caching;
using Microsoft.Practices.EnterpriseLibrary.Data;

namespace Chapter05.CustomSiteMapProvider
{
    /// <summary>
    /// Summary description for SqlSiteMapProvider
    /// </summary>
    [AspNetHostingPermission(SecurityAction.Demand, Level =
    AspNetHostingPermissionLevel.Minimal)]
    public class SqlSiteMapProvider : SiteMapProvider
    {

        #region " Variables "

        private string connStringName;
        private Database db;

        private SiteMapProvider _parentSiteMapProvider = null;
        private SiteMapNode rootNode = null;
        private List<DictionaryEntry> siteMapNodes = null;
        private List<DictionaryEntry> childParentRelationship = null;

        #endregion

        #region " Implementation Methods "

        public override SiteMapNode CurrentNode
        {

```

```

    get
    {
        EnsureSiteMapLoaded();
        string currentUrl = FindCurrentUrl();
        // Find the SiteMapNode that represents the current page.
        SiteMapNode currentNode = FindSiteMapNode(currentUrl);
        return currentNode;
    }
}

public override SiteMapNode RootNode
{
    get
    {
        EnsureSiteMapLoaded();
        return rootNode;
    }
}

public override SiteMapProvider ParentProvider
{
    get
    {
        return _parentSiteMapProvider;
    }
    set
    {
        _parentSiteMapProvider = value;
    }
}

public override SiteMapProvider RootProvider
{
    get
    {
        // If the current instance belongs to a provider hierarchy, it
        // cannot be the RootProvider. Rely on the ParentProvider.
        if (ParentProvider != null)
        {
            return ParentProvider.RootProvider;
        }
        // If the current instance does not have a ParentProvider,
        // it is not a child in a hierarchy and can be the
        // RootProvider.
        else
        {
            return this;
        }
    }
}

```



```

    }
  }
}

public override SiteMapNode FindSiteMapNode(string rawUrl)
{
    EnsureSiteMapLoaded();

    // Does the root node match the URL?
    if (RootNode.Url == rawUrl)
    {
        return RootNode;
    }
    else
    {
        SiteMapNode candidate;
        // Retrieve the SiteMapNode that matches the URL.
        lock (this)
        {
            candidate = GetNode(siteMapNodes, rawUrl);
        }
        return candidate;
    }
}

public override SiteMapNodeCollection GetChildNodes(SiteMapNode node)
{
    EnsureSiteMapLoaded();

    SiteMapNodeCollection children = new SiteMapNodeCollection();
    // Iterate through the ArrayList and find all nodes that have the
    // specified node as a parent.
    lock (this)
    {
        for (int i = 0; i < childParentRelationship.Count; i++)
        {
            string nodeUrl = childParentRelationship[i].Key as string;

            SiteMapNode parent = GetNode(childParentRelationship, nodeUrl);

            if (parent != null && node.Url == parent.Url)
            {
                // The SiteMapNode with the Url that corresponds to
                // nodeUrl is a child of the specified node. Get the
                // SiteMapNode for the nodeUrl.
                SiteMapNode child = FindSiteMapNode(nodeUrl);
                if (child != null)

```

```

        {
            children.Add(child);
        }
        else
        {
            throw new Exception("ArrayLists not in sync.");
        }
    }
}
}
return children;
}

protected override SiteMapNode GetRootNodeCore()
{
    EnsureSiteMapLoaded();
    return RootNode;
}

public override SiteMapNode GetParentNode(SiteMapNode node)
{
    // Check the childParentRelationship table and find the parent
    // of the current node. If there is no parent, the current node
    // is the RootNode.
    SiteMapNode parent;
    lock (this)
    {
        // Get the Value of the node in childParentRelationship
        EnsureSiteMapLoaded();
        parent = GetNode(childParentRelationship, node.Url);
    }
    return parent;
}

public override void Initialize(string name, NameValueCollection attributes)
{
    lock (this)
    {
        base.Initialize(name, attributes);

        connStringName = attributes["connectionStringName"].ToString();
        //SqlCacheDependencyAdmin.EnableNotifications(connString);
        db = DatabaseFactory.CreateDatabase(connStringName);
        siteMapNodes = new List<DictionaryEntry>();
        childParentRelationship = new List<DictionaryEntry>();
        EnsureSiteMapLoaded();
    }
}

```

```

}

#endregion

#region " Private helper methods "

private SiteMapNode GetNode(List<DictionaryEntry> list, string url)
{
    for (int i = 0; i < list.Count; i++)
    {
        DictionaryEntry item = list[i];
        if ( ((string)item.Key).ToLower().Equals(url.ToLower()))
            return item.Value as SiteMapNode;
    }
    return null;
}

private string FindCurrentUrl()
{
    try
    {
        // The current HttpContext.
        HttpContext currentContext = HttpContext.Current;
        if (currentContext != null)
        {
            return currentContext.Request.RawUrl;
        }
        else
        {
            throw new Exception("HttpContext.Current is Invalid");
        }
    }
    catch (Exception e)
    {
        throw new NotSupportedException(
            "This provider requires a valid context.", e);
    }
}

private void EnsureSiteMapLoaded()
{
    if (rootNode == null)
    {
        // Build the site map in memory.
        LoadSiteMapFromDatabase();
    }
}

```

```

protected virtual void LoadSiteMapFromDatabase()
{
    lock (this)
    {
        // If a root node exists, LoadSiteMapFromDatabase has already
        // been called, and the method can return.
        if (rootNode != null)
        {
            return;
        }
        else
        {
            // Clear the state of the collections and rootNode
            Clear();
            SiteMapNode temp = null;

            DataSet nodes = LoadSiteMapNodes();
            if (nodes != null && nodes.Tables.Count > 0)
            {
                string baseUrl = HttpRuntime.AppDomainAppVirtualPath + "/";
                foreach (DataRow node in nodes.Tables[0].Rows)
                {
                    long parentNodeId = node["ParentID"] is long ?
                        (long)node["ParentID"] : 0L;
                    String url = node["Url"] as String;
                    String parentUrl = node["ParentUrl"] as String;
                    String title = node["Title"] as String;

                    temp = new SiteMapNode(this, baseUrl + url,
                        baseUrl + url, title);

                    // Is this a root node yet?
                    if (null == rootNode && parentNodeId < 0)
                    {
                        rootNode = temp;
                    }
                    // If not the root node, add the node to the
                    // various collections.
                    else if (parentUrl != null)
                    {
                        siteMapNodes.Add(
                            new DictionaryEntry(temp.Url, temp));
                        // The parent node has already been added
                        // to the collection.
                        SiteMapNode parentNode = FindSiteMapNode(
                            baseUrl + parentUrl);
                        if (parentNode != null)

```

```

        {
            childParentRelationship.Add(
                new DictionaryEntry(temp.Url, parentNode));
        }
        else
        {
            throw new Exception(
                "Parent node not found for current node.");
        }
    }
}
}
}
}
return;
}

private void Clear()
{
    rootNode = null;
    siteMapNodes.Clear();
    childParentRelationship.Clear();
}

/// <summary>
/// Get SiteMap Nodes from the database
/// </summary>
/// <returns></returns>
internal DataSet LoadSiteMapNodes()
{
    String cacheKey = SqlSiteMapHelper.CACHE_KEY;
    object obj = HttpRuntime.Cache.Get(cacheKey);
    if (obj != null)
    {
        return obj as DataSet;
    }
    DataSet ds = null;

    try
    {
        using (DbCommand dbCmd =
            db.GetStoredProcCommand("sm_GetSiteMapNodes"))
        {
            ds = db.ExecuteDataSet(dbCmd);
        }
    }
    catch (Exception ex)

```

```

        {
            HandleError("Exception with LoadSiteMapNodes", ex);
        }

        //SqlCacheDependency tableDependency =
        // new SqlCacheDependency(connStringName, "sm_SiteMapNodes");
        HttpRuntime.Cache.Insert(cacheKey, ds, null, DateTime.Now.AddHours(1),
            TimeSpan.Zero, CacheItemPriority.NotRemovable, OnRemoveCallback);

        //return the results
        return ds;
    }

    private void OnRemoveCallback(string key, object value,
        CacheItemRemovedReason reason)
    {
        if (CacheItemRemovedReason.DependencyChanged == reason ||
            CacheItemRemovedReason.Removed == reason)
        {
            Clear();
            LoadSiteMapFromDatabase();
        }
        else
        {
            Clear();
        }
    }

    private void HandleError(string message, Exception ex)
    {
        //TODO log error
        throw new ApplicationException(message, ex);
    }

    #endregion
}
}

```

Listing A-26. *SqlSiteMapHelper.cs*

```

using System;
using System.Data.Common;
using System.Web;
using Microsoft.Practices.EnterpriseLibrary.Data;

namespace Chapter05.CustomSiteMapProvider

```

```
{
    public class SqlSiteMapHelper
    {
        public const string CACHE_KEY = "SqlSiteMapNodes";

        private Database db;

        public SqlSiteMapHelper(string connStringName)
        {
            db = DatabaseFactory.CreateDatabase(connStringName);
        }

        public void RepopulateSiteMapNodes()
        {
            try
            {
                using (DbCommand dbCmd =
                    db.GetStoredProcCommand("sm_RepopulateSiteMapNodes"))
                {
                    db.ExecuteNonQuery(dbCmd);
                    InvalidateSiteMapCache();
                }
            }
            catch (Exception ex)
            {
                HandleError("Exception with RepopulateSiteMapNodes", ex);
            }
        }

        internal void InvalidateSiteMapCache()
        {
            HttpRuntime.Cache.Remove(CACHE_KEY);
        }

        private void HandleError(string message, Exception ex)
        {
            //TODO log error
            throw new ApplicationException(message, ex);
        }
    }
}
```

Table Scripts

Listing A-27. *sm_SiteMapNodes.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'U' AND
    name = 'sm_SiteMapNodes')
    BEGIN
        DROP Table sm_SiteMapNodes
    END
GO

CREATE TABLE [dbo].[sm_SiteMapNodes](
    [ID] [bigint] IDENTITY(1,1) NOT NULL,
    [ParentID] [bigint] NULL,
    [Url] [nvarchar](150) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Title] [nvarchar](50) COLLATE SQL_Latin1_General_CP1_CI_AS NOT NULL,
    [Depth] [int] NOT NULL CONSTRAINT [DF_sm_SiteMapNodes_Depth] DEFAULT ((0)),
    [Creation] [datetime] NOT NULL,
    [Modified] [datetime] NOT NULL,
    CONSTRAINT [PK_sm_SiteMapNodes] PRIMARY KEY CLUSTERED
(
    [ID] ASC
)WITH (PAD_INDEX = OFF, IGNORE_DUP_KEY = OFF) ON [PRIMARY]
) ON [PRIMARY]

GO

GRANT SELECT ON sm_SiteMapNodes TO PUBLIC
GO
```

Stored Procedure Scripts

Listing A-28. *sm_GetSiteMapNodes.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND ➡
    name = 'sm_GetSiteMapNodes')
    BEGIN
        DROP Procedure sm_GetSiteMapNodes
    END
GO

CREATE Procedure dbo.sm_GetSiteMapNodes
AS

SELECT c.ID, c.ParentID, c.Url, c.Title, c.Depth, p.Url AS ParentUrl
FROM sm_SiteMapNodes AS c
LEFT OUTER JOIN sm_SiteMapNodes AS p ON p.ID = c.ParentID
```



```
ORDER BY c.Depth, c.ParentID
```

```
GO
```

```
GRANT EXEC ON sm_GetSiteMapNodes TO PUBLIC
```

```
GO
```

Listing A-29. *sm_InsertSiteMapNode.sql*

```
IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND  
    name = 'sm_InsertSiteMapNode')
```

```
    BEGIN
```

```
        DROP Procedure sm_InsertSiteMapNode
```

```
    END
```

```
GO
```

```
CREATE Procedure dbo.sm_InsertSiteMapNode
```

```
(
```

```
    @ParentID bigint,
```

```
    @Url nvarchar(150),
```

```
    @Title nvarchar(50),
```

```
    @Depth int,
```

```
    @ID bigint OUTPUT
```

```
)
```

```
AS
```

```
IF NOT EXISTS (SELECT * FROM sm_SiteMapNodes WHERE Url = @Url)
```

```
BEGIN
```

```
    INSERT INTO sm_SiteMapNodes (ParentID, Url, Title, Depth, Creation, Modified)
```

```
    VALUES (
```

```
        @ParentID,
```

```
        @Url,
```

```
        @Title,
```

```
        @Depth,
```

```
        GETDATE(),
```

```
        GETDATE())
```

```
    )
```

```
    SET @ID = @@IDENTITY
```

```
END
```

```
ELSE
```

```
BEGIN
```

```
    SET @ID = ( SELECT ID FROM sm_SiteMapNodes WHERE Url = @Url )
```

```
END
```

```
GO
```

```
GRANT EXEC ON sm_InsertSiteMapNode TO PUBLIC
```

```
GO
```

Listing A-30. *sm_RepopulateSiteMapNodes.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P'
AND name = 'sm_RepopulateSiteMapNodes')
BEGIN
    DROP Procedure sm_RepopulateSiteMapNodes
END

GO

CREATE Procedure dbo.sm_RepopulateSiteMapNodes

AS

SET NOCOUNT ON

DECLARE @RootNodeID bigint
DECLARE @AlbumsNodeID bigint

-- reset the table
--TRUNCATE TABLE sm_SiteMapNodes
--DBCC CHECKIDENT (sm_SiteMapNodes, RESEED, 0)
DELETE FROM sm_SiteMapNodes

EXEC sm_InsertSiteMapNode -1, 'Default.aspx', 'Home', 0,
    @RootNodeID OUTPUT
EXEC sm_InsertSiteMapNode @RootNodeID, 'Albums/Default.aspx',
    'Albums', 1, @AlbumsNodeID OUTPUT

DECLARE @Albums TABLE
(
    ID int IDENTITY,
    AlbumID bigint,
    [Name] varchar(50),
    UserName nvarchar(256)
)

INSERT INTO @Albums (AlbumID, [Name], UserName)
SELECT ID, [Name], UserName
FROM pap_Albums
WHERE IsActive = 1

DECLARE @CurID int
DECLARE @MaxID int
DECLARE @AlbumID bigint
DECLARE @AlbumNodeID bigint
DECLARE @Name varchar(50)
DECLARE @UserName nvarchar(256)

```

```

DECLARE @Url nvarchar(150)

SET @MaxID = ( SELECT MAX(ID) FROM @Albums )
SET @CurID = 1

WHILE (@CurID <= @MaxID)
BEGIN
    SET @AlbumID = ( SELECT AlbumID FROM @Albums WHERE ID = @CurID )
    SET @Name = ( SELECT Name FROM @Albums WHERE ID = @CurID )
    SET @UserName = ( SELECT UserName FROM @Albums WHERE ID = @CurID )

    SET @Url = ( 'Albums/Album.aspx?AlbumID=' +
        CONVERT(varchar(10), @AlbumID) +
        '&UserName=' + @UserName )

    -- PRINT 'Name = ' + @Name
    -- PRINT 'UserName = ' + @UserName
    -- PRINT 'Url = ' + @Url

    EXEC sm_InsertSiteMapNode @AlbumsNodeID, @Url, @Name, 2,
        @AlbumNodeID OUTPUT

    SET @CurID = @CurID + 1
END

SET NOCOUNT OFF

GO

GRANT EXEC ON sm_RepopulateSiteMapNodes TO PUBLIC
GO

```

Listing A-31. *sm_UpdateSiteMapNode.sql*

```

IF EXISTS (SELECT * FROM sysobjects WHERE type = 'P' AND
    name = 'sm_UpdateSiteMapNode')
BEGIN
    DROP Procedure sm_UpdateSiteMapNode
END

GO

CREATE Procedure dbo.sm_UpdateSiteMapNode
(
    @ID bigint,
    @ParentID bigint,
    @Url nvarchar(150),
    @Title nvarchar(50),

```

```
@Depth int
)
AS

IF EXISTS (SELECT * FROM sm_SiteMapNodes WHERE ID = @ID)
BEGIN
    UPDATE sm_SiteMapNodes SET
        ParentID = @ParentID,
        Url = @Url,
        Title = @Title,
        Depth = @Depth,
        Creation = GETDATE(),
        Modified = GETDATE()
END

GO

GRANT EXEC ON sm_UpdateSiteMapNode TO PUBLIC
GO
```

Index

■ Numbers and symbols

- @maximumRows input parameter, 67
- @OldFavoriteLinkId, checking value for, 211–212
- @sortExpression input parameter, 67
- @startRowIndex input parameter, 67

■ A

- A switch, 8
- .abc files, adding build provider for, 237
- AbcBuildProvider, for .abc files, 237–241
- absolute expiration, setting for cached data, 164
- abstract properties, 317
- abstract property and method declarations, 132–133
- abstract provider class, 132–133
- Active Directory
 - attributeMapEmail attribute, 113
 - released with ASP.NET 2.0, 112
- Add method, 160
- Add Provider Services.cmd script, 8
- Add SQL Cache Dependencies.cmd, 170–171
- AddConstraints.sql, 65
 - Photo Album provider, 372
- AddIndexing.sql, 63–64
- Admin role, restricting Admin section access with, 117–118
- Admin section, securing, 34–35
- Admin user, creating, 35–36
- AdminHyperlink, setting Visibility for, 117
- Akamai, hosting on, 294
- alter script, writing statements manually with, 272–273
- alter statements, updating database structure with, 272–273
- Always Show Solution option, setting, 3
- anonymous method, with delegate, 166–167
- anonymous profiles
 - vs. authenticated profiles, 121
 - configuring, 119–120
 - deleting inactive with T-SQL, 120
 - enabling, 17, 119
 - migrating, 121–122
- Apache Web server vs. Linux to IIS on Windows NT, 291
- App_Code folder, Utility class in, 17
- App_Data folder, DATA_DIRECTORY in, 12

- application state, as alternative to caching, 149–150
- Application_Start method, 284–285
- ASP.NET, caching in, 147–148
- ASP.NET 2.0 application
 - code and database separation, 7
 - common folders, 4–5
 - datasource configuration, 5–7
 - DATA_DIRECTORY for, 12
 - managing provider services, 7–11
 - preparing your environment, 1–7
 - SqlMembershipProvider released with, 112
- ASP.NET 2.0 data model. *See* data model
- ASP.NET controls, using provider-powered, 126–128
- ASP.NET machineKey Generator, 129
- ASP.NET Membership User accounts, integration of website to, 203
- ASP.NET providers
 - configuring, 11–18
 - introduced in ASP.NET 2.0, 7
 - managing services, 7–11
 - mixing and matching, 11
 - removing services, 8–10
- ASP.NET runtime, accessing values through profiles with, 125
- aspnet_regsql.exe utility, 8
- AspNet_SqlCacheRegisterTableStored-Procedure, updated, 168–170
- AssemblyHelper class, 312–313
- Assert statements, checking return values with, 71
- AttachedProperty method, 239–240
- attributeMapEmail attribute, for Active Directory implementation, 113
- authenticated profiles, migrating from anonymous to, 121–122
- authentication system, security of in ASP.NET, 126–128
- authentication token, 126
- AutoScaffold page, SubSonic code generator, 248–249

■ B

- back end, distributing to, 296–297
- banking website, securing access to, 126–128
- BeforeBuild and AfterBuild targets, 258

- binding
 - input parameters, 81–84
 - user controls, 83–84
 - BirthDate column, using COALESCE
 - function on, 207
 - Blinq code generator, for ASP.NET, 249–252
 - Blinq project, prototype on ASP.NET website, 249–250
 - BLOB storage mechanism, used by
 - SqlProfileProvider, 125
 - BLOBs, dynamic profiles and profiles as, 124–125
 - bottlenecks, effect on scalability, 291
 - BoundField, replacing with TemplateField, 78
 - breadcrumb trail, implementation of, 139
 - Browsable attribute, 226
 - Build Action, setting to Embedded Resource, 279
 - build providers, 234–236
 - build script, creating at solution level, 258–260
 - build.proj script
 - called by Build target, 269–270
 - creating at solution level, 258–260
 - BuildProvider class, in
 - System.Web.Compilation namespace, 235–236
 - buildProviders configuration element, 234
 - business logic, encapsulating in stored procedures, 198–199
 - business objects, creating, 222
- C**
- cache
 - adding and getting items with index, 160–161
 - adding dependencies, 168
 - removing items from, 165
 - Cache object, 147
 - cached data, 164–165
 - CacheItemPriority parameter, values, 161
 - CacheItemRemovedCallBack parameter, 162–163
 - CacheItemRemovedReason parameter, values, 162
 - caching
 - adding to enable polling, 160
 - alternatives to, 148
 - enumerating over the cache, 161
 - improving performance with, 147
 - methods, 160
 - options, 153–163
 - parameters, 161–163
 - performance strategies, 187–189
 - problems with, 186
 - proxy, 294–295
 - purging cached items by type, 161
 - simulation, 163
 - time-out, effect on scalability, 290
 - CaptureSettings method, 103–104
 - Category attribute, 226
 - change scripts, generating, 272–273
 - Change Scripts folder, for update scripts, 273
 - Chapter09Configuration class, loading
 - configuration with, 287–288
 - Chapter09Section class, 286–287
 - Chapter09SectionGroup class, building, 285–286
 - chpt02_GetAllPeople.sql stored procedure, 47
 - chpt03_GetPeopleRowCount.sql stored procedure, 68–69
 - chpt03_GetPeopleSubSetSorted.sql stored procedure, 66–67
 - chpt03_SaveLocation.sql script, 60–61
 - chpt03_SavePerson.sql script, 59
 - chpt03_SavePersonWithLocation.sql script, 61
 - chpt07_GetFavoriteLinksByProfileID.sql stored procedure, 209
 - chpt07_GetFavoriteLinksByTag.sql stored procedure, 209–210
 - chpt07_PurgeFavoriteLinksByProfileID stored procedure, 216–217
 - chpt07_PurgeLinkTagsByProfileID stored procedure, 215–216
 - chpt07_SaveFavoriteLink stored procedure, parameters, 211
 - chpt09_GetSchemaVersion.sql script, 278–279
 - chpt09_SchemaVersions.sql script, 277
 - chpt09_SetSchemaVersion.sql script, 277–278
 - CityListingControl.ascx, 85–86
 - class library, structure of, 280
 - ClassLibrary, creating, 218
 - ClearCachedItem method, 247
 - client-side caching, 294–295
 - COALESCE function, 207–208
 - code generation, 233–234
 - Code Snippet Manager, 45
 - Code Snippet selection menu, 46
 - code snippets. *See* data access code snippets
 - CodeAssignStatement class, 240
 - CodeCompileUnit, 237–238
 - CodeDom namespace, 237–242
 - CodeFieldReferenceExpression class, 240
 - CodeMemberProperty type, 240
 - CodeMethodReturnStatement class, 240
 - CodeNamespace type, 237–238
 - CodeNamespaceImport type, 237–238
 - CodePlex
 - SubSonic partially hosted on, 242–243
 - website, 242

- CodeSnippetCompileUnit class, 236
 - CodeTypeDeclaration type, 237–238
 - command line, adding provider services from, 8–10
 - Common folder
 - adding SubSonic and Blinq tools to, 250
 - adding custom databound control to, 109
 - additions for SQL notifications, 175
 - common folders. *See* folders
 - Comma Separated Values (CSV) export file, generating, 125
 - CompositeDataBoundControl,
 - CreateChildControls for, 92
 - configSource attribute option, Web Deployment Projects, 264
 - configuration
 - of anonymous profiles, 119–120
 - creating custom sections, 285–288
 - custom, 314–321
 - custom for Chapter09Group, 288
 - of datasource, 5–7
 - general structure of hierarchy, 315
 - grouping, 315–317
 - section class for Photo Album Provider, 130–131
 - settings for Membership provider, 14
 - settings for Profile provider, 16
 - Configuration property, setting in MSBuild script, 260
 - ConfigurationSectionGroup class, 315
 - connection string, for AP database, 297
 - connectionStringName attribute, 113
 - in SqlMembershipProvider configuration, 114
 - connectionStrings, in Machine.config file, 12
 - console applications, for hosting a service, 335–336
 - constraints and indexes, managing, 62–65
 - content distribution network, Akamai, 294
 - content-disposition header, 125
 - continuous integration, tool for, 71
 - controls collection, creating, 104
 - ControlState, 89–91
 - CountryListingControl.ascx, 84–85
 - CreateChildControls method
 - for CompositeDataBoundControl, 92
 - getting data passed into, 94–97
 - implementing, 93–94
 - Created value, adding to tables, 205–206
 - CreatedUser event handler, 124
 - CreatePagerControls method, 98–99
 - CreateUserWizard, 122–124
 - CRUD, 233
 - methods, generating, 47
 - procedures, 59
 - CruiseControl.NET, 71
 - current context, holding data in, 150–153
 - custom configuration sections, *See* configuration
 - custom section definition, 288
- ## D
- dacreation shortcut, 219
 - dagetdr shortcut, 220
 - dagetds shortcut, 219
 - danonquery shortcut, 220
 - data
 - deleting from database, 213–217
 - loading, 222–224
 - unit tests for, 68–71
 - Data Access Application Block, 37–39
 - data access layer (DAL), 191–231
 - creating to work with data binding, 217–225
 - generated, 233–253
 - separation of concerns, 218–219
 - suggested default values in, 207
 - data access methods, creating, 219–221
 - data access providers, creating, 303–310
 - Data Access Types, code snippet, 40–41
 - data binding
 - code, 102–103
 - warning about recursive, 86
 - data caching, 159–163. *See also* caching
 - Data Contract, 201–203
 - data examples
 - nontrivial, 49
 - trivial, 47–49
 - data load, tuning, 224–225
 - data manipulation methods, get, save, and delete, 300–303
 - data model, choices, 37–54
 - data serialization, profile properties, 124
 - data transfer object (DTO), DataSets used as, 192
 - data value, casting to a property, 224
 - data warehousing, as performance strategy, 187–188
 - database
 - automating updates, 285–288
 - building, 203–225
 - configuration, 219
 - deleting data from, 213
 - deploying, 271–288
 - initialization file, 51
 - management, 55–71
 - partitioning by date, 299
 - sample for data model, 46–47
 - updating, 279–280
 - Database Creation, 41
 - Database object, 38
 - Database Projects, 55–57
 - constructing database with, 208–217

- database server
 - small company configuration for single, 296–297
 - updated growing company configuration, 297
 - web server communications to, 148
 - DatabaseManager class, loading and running scripts with, 280–285
 - DataBinder, loading data values with, 102–103
 - databound controls
 - creating, 91–110
 - embedding, 84
 - DataContracts, defining, 336
 - DataFirstNameField property, 103
 - DataObject, 53–54
 - DataObjectMethod attribute, 52
 - DataObjectMethodType attributes, 199–200
 - DataObjects, 199–203
 - DataObjects.cs, 355
 - DataReader method, 43–44, 53
 - DataRow, loading, 223–224
 - DataRowView, viewing in debugger, 194
 - DataSet method, 41–43
 - DataSets
 - debugging support, 194–195
 - inline SQL, and stored procedures, 191–199
 - populating FavoriteLinkCollection object with, 223–224
 - refactoring, 193
 - types of, 192
 - datasource configuration, 5–7
 - DataSourceSelectArguments method, adjusting, 97
 - DataSourceViewSelectCallback, callback delegate, 95
 - DateEditor user control, 79–80
 - DateTime type, casting value of, 206–207
 - datypes shortcut, 219
 - db.proj script, for database updates, 274–276
 - DBNull.value, 206–207
 - debugging
 - placing breakpoints, 109
 - support with DataSets, 194–195
 - without ViewState, 109
 - defaultProvider attribute, 113
 - DefaultValue attribute, 226
 - deploy.proj script, 266–268
 - deployment, 255–288
 - deployment files, managing creation of, 266–268
 - Design Contract, 200–201, 203
 - DetailsView control, 73–74
 - differencing virtual hard disk, creating, 2
 - Digg, 292
 - distributed back end. *See* back end
 - distributing services, 295–296
 - DLINQ, querying tables with, 322
 - Do Nothing Button, 107
 - DomainKey type, 307–309
 - DomainObject class
 - comparison methods, 309–310
 - revised, 307–310, 341
 - dotnetUserGroup element, 315
 - Dropconstraints.sql, Photo Album provider, 372
 - DTO. *See* data transfer object (DTO)
 - DugConfiguration class, 315–317
 - DugDataContext class, creating, 325–327
 - dug_DeleteEvent.sql stored procedure, 302–303
 - dug_GetEvent.sql stored procedure, 300
 - dug_GetEventsByDate.sql stored procedure, 300–301
 - dug_SaveEvent.sql stored procedure, 301–302
 - dynamic language runtime (DLR), .NET platform, 236
 - dynamic profiles and profiles as BLOBs, 124–125
- ## E
- EdifTemplate, DateEditor placed in, 80–81
 - embedded scripts, using, 276–285
 - EndDaysBack property, 226
 - endpoint, creating Service Broker, 173
 - Enterprise Library, 38–39
 - Data Access Application Block as part of, 37
 - SubSonic support for, 242
 - Event class, .NET user group website, 304–306
 - event handlers
 - CreatedUser, 124
 - Login Authenticate, 126–127
 - ObjectDataSource1_Selecting, 227
 - OnSelecting, 82
 - PageButton_Click, 100
 - Selecting event, 82
 - tagCloud_OnTagSelected, 231
 - TagSelected, 228–229
 - Web Form button, 120
 - EventDataObject, 337–339
 - EventListingControl.ascx, 339–341
 - EventProvider class, 303–304
 - LINQ implementation of, 322
 - EventResult class, creating, 322–325
 - EventResultConvert class, 327–328
 - events, relaying information with, 228–229
 - EventSection class, 320–321
 - EventServiceHost class, for event implementation, 335–336

exceptions, handling in Data Access Layer, 222

ExecuteDDL task, for database updates, 274–276

expiration header, 294

F

/f switch, Blinq code generator, 250

Favorite Link record

checking existing, 211–212

saving with INSERT or UPDATE, 212–213

Favorite Links website

central business object, 222

stored procedure for getting data, 209–210

FavoriteLink object

testing performance of, 224–225

viewing in debugger, 194–195

FavoriteLinkCollection object, 223–224

FavoriteLinkDomain class, creating, 219–221

FavoriteLinks database

deleting data from, 213–217

managing scripts, 208–209

Membership User record in, 205

relationships in, 205

preventing null values in, 208

FetchByID method, with caching, 246–247

FetchPersonsByLocationID method, 247–248

fields, editing and validating, 77–81

FirstName data field property, 103

Flickr photo website, building provider that uses, 129

FlickrHelper.cs, Website class, 379

folders, creating common, 4–5

foreign key constraints

handling of, 62

removing and adding, 64–65

FormatException, when saving a date value, 77

Forms Authentication token, assignment of, 117

FormsAuthenticationTicket, inserting remote address into, 126–128

FormView control, 74–76

fragment caching, 155

of user control output, 153

with postcache substitution, 156–159

FullName property, creating partial class to add, 245–246

G

Generate Change Script button, 62, 272

GenerateCode method, BuildProvider class, 235

GeoIP services, 128

GetAllPeopleDataSet method, 51–52

GetAllPeopleReader, adding to PersonDomain, 53

GetClassName method, 235–236

GetContents method, 235

GetData method, 94

GetFieldName method, 240–241

GetGeneratedCode method, 236–237

GetLocationConsumers method, 312

GetLoginTimeout method, 127

GetManifestResourceStream method, 283

GetNamespace method, 240

GetPeopleRowCount method, 69

GetPeopleSubSetSortedDataSet method, 67–68

GetPerson method, from Person class, 252

GetPersonsByLocation method, 252

GetProduct method, in Utility class, 158–159

GetRecentFavoriteLinks method, 219

GetSqlCommand method, 282–283

GetTotalRowCount, 97

GridView control, 73, 76–77

binding data to, 200–201

with TemplateField, 78–79

Guidance Automation Extensions for VS, 39

H

Hao Kung, 125

Home.aspx page, creating, 229–230

hosting, by Akamai, 294

I

IDataItemContainer members, 101–102

IDataReader, loading data with, 224

IEventService interface, WCF provider, 334

ILocationConsumer interface, 311

indexes

adding to a table, 273

and constraints, managing, 62–65

defragging, 64

improving performance with, 65

index creation script, 273

init.sql script, as embedded resource, 279

InitializeDatabase method, 281

InitializePagerControls method, 99

inline SQL

maintenance considerations, 196

security considerations, 196–198

input parameters, binding, 81–84

InputParameterExample.aspx, 81

InputParameterExample2.aspx, 82

Insert method, 160

IntelliSense support, for generated class, 234

Interactive property, setting in MSBuild script, 260

InvalidCastException, 192

IP address, physical location by, 128

IsLocationInUse method, method called by, 311–312

ISNULL function vs. COALESCE function, 208

IsRemoteAddressMatched method, 127–128
 IsUsingLocation method, 311–312
 ItemGroup element, MSBuild scripts, 257

J

JetBrains, 179

K

Kung, Hao, 125

L

lazy loading, 188–189
 LinksControl user control, creating, 226–228
 LINQ provider, implementing, 321–332
 LinqEventProvider class, creating, 328–332
 Linux to IIS on Windows NT vs. Apache Web server, 291
 LiteralControls, used by PersonRow, 104–105
 Load event, effect on data loading, 152
 load time and page size, 87
 load-balancing hardware, for web farms, 293–294
 LoadControlState method, 90, 108
 LoadViewState method, 106
 Location property, getter for, 310–311
 LocationManager class, 311
 locations
 saving, 60–61
 using, 311–313
 logging, handling with stored procedures, 198
 Login Authenticate event handler, 126–127
 Login controls, 112
 login page, configuration of, 116–117

M

Machine.config file, 11–12
 Management Studio. *See* SQL Server Management Studio
 many-to-many relationships, 205
 MarkDirty method, 244–245
 markup code, creating efficient, 104
 master key, defining, 173
 MaxIndex property, 98
 MembersControl.ascx, controls held in, 18
 Membership API, creating user with, 122–124
 Membership provider, configuration of, 13–14. *See also* ASP.NET providers
 membership system, implementing with forms authentication, 118
 MembershipProvider class, 112–113
 Microsoft Patterns & Practices group
 Guidance Automation Extensions for VS, 39
 modules provided by, 37
 Microsoft Virtual PC environments, preparing, 1–2

Microsoft.WebDeployment.Tasks.dll task, 264–265

mixed mode,
 authentication, 6
 select arguments in, 95

Modified value, adding to tables, 205–206

MSBuild, automation with, 256–260

MSBuild Community Tasks, 257

MSBuild scripts

 BeforeBuild and AfterBuild targets, 258
 Common folder additions, 271
 for database updates, 274–276
 parameters for, 260
 primary elements of, 257
 rules for working with, 256
 skeleton of basic, 257

MSDN, content available on, 140

N

Name property, breaking up, 201–202
 /namespace switch, Blinq code generator, 250

NamesUpdate-01.sql script, 280

.NET 3.0 with WCF, released by Microsoft, 332

.NET user group website

 creating data providers for, 303–310
 creating EventProvider object for, 303–306
 sample application, 298–303

netTcpBinding, 333

networking equipment, effect on network
 bottlenecks, 291

NHibernate, O/R mapping tool, 222

Nonquery method, 44–45

nontrivial data examples, 49

Nontyped DataSets, 51–53, 192–193

notifications system, granting permissions to
 use, 174–175

null values, preventing, 207–208

nulls

 converting default date to, 207–208
 handling of, 206–207

NUnit, 179

 unit-testing framework for .NET, 70

O

Object/Relational (O/R) mapping, 222–224

ObjectDataSource, 199–203

 adding a second to a Web Form, 52
 information given by, 67–70

ObjectDataSource declaration, for Person
 table, 250–252

ObjectDataSource1

 for use by databound control, 69–70
 LinksControls, 227

ObjectDataSource1_Selecting event handler, 227

OnChange handler method, 166

- one-to-many relationships, 205
- one-to-one relationships, 205
- OnPreRender method, setting text properties in, 104–105
- OnSelecting event handler, 82
- output caching, 153
 - enabled, 154
 - problems with, 155
 - programmatically setting a page for, 154–155
 - settings when language is a concern, 186

P

- page caching, 154–155
- page size and load time, 87
- PageIndex property, 100–101
- pager events, wiring, 98–101
- PagerButton_Click event handler, 100
- pages, creating, 229–231
- PageSize property, calculating MaxIndex property with, 98
- PageStatePersister property, 87–88
- Page_Load method, 151
- paging, reducing ViewState size with, 88
- pap_Albums.sql, Photo Album table script, 371
- pap_DeleteAlbum.sql, 373
- pap_DeletePhoto.sql, 373
- pap_GetAlbumsByUserName.sql, 374
- pap_GetPhotosByAlbum.sql, 374
- pap_InsertAlbum.sql, 375
- pap_InsertPhoto.sql, 375
- pap_MoveAlbum.sql, 376
- pap_MovePhoto.sql, 377
- pap_Photos.sql, Photo Album table script, 371
- pap_UpdateAlbum.sql, 377
- pap_UpdatePhoto.sql, 378
- partial classes, generated by SubSonic generator, 245–247
- partitioning, 206
 - database by date, 299
- password policy, creating for
 - SqlMembershipProvider, 114–115
- passwords, regular expressions for, 114–115
- performance and scalability, understanding, 289–297
- performance tuning, 152–153
- PerformSelect method, 95–97
- permissions, granting to use notifications system, 174–175
- Person and Location tables, 46
- Person class, object browser with generated, 241
- person record, inserting new, 59
- Person.abc file, 237
- Personalization provider. *See* ASP.NET providers
- PersonID parameter, setting, 59
- PersonListing.aspx.cs Code-Behind, 83–84
- PersonListingControl.ascx, 83
- PersonRow, creating, 101–105
- Photo Album provider, 343–398
 - building, 129–138
 - classes, 344–371
 - configuration, 343
 - constraints scripts, 372–373
 - implementing, 134–135
 - requirements for, 130
 - stored procedure scripts, 373–379
 - table scripts, 371–372
 - unit testing, 137–138
- PhotoAlbumProvider, Web.config for, 138
- PhotoAlbumProvider.cs class, 132–133, 344
- PhotoAlbumProviderCollection.cs class, 131, 370
- PhotoAlbumSection.cs class, 130–131, 369
- PhotoAlbumService.cs, 135–137, 368
- polling
 - configuring SqlCacheDependency for, 171–172
 - enabling for a table, 168–171
 - problems with, 172
 - vs. query notifications, 172–173
- postcache substitution, 155–159
- PreBuildEvent and PostBuildEvent properties, 258
- PreferredCustomer role, adding users to, 117
- PreLoad event, 99–100
- ProcessFieldNodes method, 238–239
- profile data, trickiness of getting, 124–125
- Profile_MigrateAnonymous event, 121
- Profile_onMigrateAnonymous script, 17–18
- Profile provider, configuration, 15–18. *See also* ASP.NET providers
- profiles exporter, for exporting profile data, 125
- PropertyGroup element, MSBuild scripts, 257
- provider collection class, for building Photo Album provider, 131
- provider service class, 135
- provider services. *See also* ASP.NET providers
 - managing, 7–11
 - Membership configuration, 13–14
 - mixing and matching, 11
 - removing, 8–10
- ProviderBase class, 113
 - EventProvider.cs inherits from, 304
- ProviderConfigurationSection class, 317
- providers. *See also* ASP.NET providers
 - creating new, 321
 - using, 337–341

Providers property, defined in
 SpeakerSection class, 314
 ProviderSections property, 315–316

■ Q

Query class, used by SubSonic query tool,
 247
 query notifications
 functions, operators, and expressions not
 allowed, 175–176
 vs. polling, 172–173
 removing items from cache with, 172–176
 requirements for, 175–176
 troubleshooting, 176–186
 query tools, 247–248

■ R

refactoring, 193
 relationships, managing, 204–205, 310–313
 release branching, 256
 remote address, inserting, 126–128
 Remove Indexing.sql, 62–63
 Remove method, 160
 Remove Provider Services.cmd script, 10
 Remove SQL Cache Dependencies.cmd, 171
 RemoveConstraints.sql, 64–65
 RemoveLinkTag stored procedure, 213–214
 Repeater control, 227–228
 ReplaceConfigSections task, for PostBuild
 deployments, 264–265
 rigid Typed Datasets, 50
 RoleManager.ascx.cs script, 29–34
 RoleProvider, enabling, 118
 roles and users, adding, 35–36
 Roles provider, configuration, 14–15. *See also*
 ASP.NET providers
 Roles.Enabled property, 14–15
 RolesManager.ascx control, 18
 Round Robin DNS, 293–294
 RowFilter property, using with ProductID,
 187–188
 rows count, getting total, 97–98
 Row_Number function, 67
 Run On process, silent feature built into, 57
 RunBuild.cmd script, 270–271
 running MSBuild script with, 260
 RunDb.cmd script, running db.proj script
 with, 276
 RunDeploy.cmd script, 268–269
 RunSqlCommand method, 283–284
 runtime errors, 192–193

■ S

sample application, 289–341
 SaveControlState method, 89–90, 107–108
 SaveFavoriteLink method, 220–221

SaveViewState method, 105
 saving data, 211–213
 scaffolding, 248–249
 scalability
 effect of concurrent requests on, 290
 effect of traffic spikes on, 291–293
 planning for, 298
 scalability and performance. *See*
 performance and scalability
 script templates, using, 273
 scripts
 creating common folders for, 4–5
 using embedded, 276–285
 ScriptsPrefix Constant script, 280–281
 security considerations, in inline SQL,
 196–198
 Select method, calling, 94–95
 SelectCountMethod property, 68
 Selecting event handler, 82
 Service Broker, enabling, 173–174
 services. *See* distributing services
 Session
 as alternative to caching, 150
 and ViewState, 87–88
 shopping basket, setting up, 16–18
 shortcuts
 dcreation, 219
 dagetdr, 220
 dagetds, 219
 danonquery, 220
 datypes, 219
 SiteMap provider. *See* SQL SiteMap
 provider.cs class
 Slashdot, 292
 sliding expiration, setting for cached data,
 164–165
 sm_GetSiteMapNodes.sql, 394
 sm_InsertSiteMapNode.sql, 142–143, 395
 sm_RepopulateSiteMapNodes.sql
 loading SiteMap with, 140–142
 stored procedure script, 395
 sm_SiteMapNodes table, database
 requirements, 140
 sm_SiteMapNodes.sql, table script, 393
 sm_UpdateSiteMapNode.sql, 397
 social bookmarking website, building,
 203–231
 source-control system, keeping broken code
 out of, 71
 SpeakerSection class, 314
 Spring Framework, 245
 SQL cache dependencies, 165–186
 SQL injection attacks, 196–198
 SQL Photo Album provider. *See* Photo Album
 provider
 SQL Providers, 111–145
 SQL queries, 196

SQL Server, mixed-mode authentication, 6

SQL Server Management Studio

- managing table creation and modifications, 57
- removing indexing, 62–63
- selecting people by first and last name, 57–58

SQL Server Surface Area Configuration for

- Features utility, 173

SQL SiteMap provider

- building, 139–145
- classes, 385
- stored procedure scripts, 394

SQL Web event provider. *See* ASP.NET providers

SqlCacheDependency

- configuring for polling, 171–172
- constructors, 167

SqlCommand object, 167

SqlDependency object, 165–167

SqlMembershipProvider, 112–115

SqlPhotoAlbumProvider.cs, 345

- initialize method for, 134–135

SqlProfileProvider, 118–125

SqlRoleProvider, grouping users with, 115–118

SqlSiteMapHelper.cs class, 392

SqlSiteMapProvider.cs class, 385

SqlTableProfileProvider, 125

StartDaysBack property, 226

static files, developing with, 295

stored procedures. *See also individual stored procedure names*

- chpt03_GetPeopleRowCount.sql, 68–69
- chpt03_GetPeopleSubSetSorted.sql, 66–67
- chpt03_SavePersonWithLocation.sql, 61
- extending applications with, 198–199
- managing, 57–61
- Photo Album provider, 373–379
- planning for Favorite Links website, 209
- purging to improve performance, 214
- requesting a subset of items, 66–67
- returning a range of items, 66–67
- saving data, 211–213
- selecting people by first and last name, 58
- sm_InsertSiteMapNode.sql, 142–143

StringBuilder, 153

sub-domains, splitting services into, 295–296

SubSonic code generator, 242–243

- adjusted template for table properties, 244–245

- AutoScaffold page, 248–249
- built-in query tool, 247–248
- class template, 243–244
- command-line tool, 243
- running, 243
- templating system, 243–245

- Tools directory, 242–243
- website, 243

Substitution control, 155–156

SubstitutionFragment class, 156

switches, adding services with, 8

system environment variables, adding to

- folders, 5

system tray application, CruiseControl.NET, 71

System.CodeDom namespace. *See* CodeDom namespace

system.web section, in Machine.config file, 11–12

T

/t switch, Blinq code generator, 250

table scripts, SQL SiteMap Provider, 393

tables, denormalized vs. using join, 187–188

TagCloudControl, 228–229

TagCloudEventArgs, 229

tagCloud_OnTagSelected event handler, 231

TagSelected event handler, 228–229

Target element, MSBuild scripts, 257

Template Explorer, 273

TemplateField, replacing BoundField with, 78

templates, creating common folders for, 4–5

templating vs. CodeDom namespace, 241–242

testing, Design and Data Contracts, 203

token purge list, assembling, 215

tools, creating common folders for, 4–5

TotalRowCount property, 97–98

traffic spikes, effect on scalability, 291–293

trivial data examples, in ASP.NET, 47–49

TrivialExample.aspx with SqlDataSource, 47–48

Typed DataSet Designer, generating CRUD methods in, 47

Typed DataSets, 192

- building customized, 49–50
- rigid, 50

U

unit tests

- for data, 70–71
- DomainTests.cs, 179–180
- most popular framework for .NET, 70
- MSBuild with, 186
- PhotoAlbumProvider, 137–138
- Test101_Caching_Off_Test Method, 181
- Test102_Caching_AbsoluteExpiration_Test Method, 181–182
- Test103_Caching_Polling_Test Method, 182–183
- Test104_Caching_Notification_Test Method, 183–184

unit tests (*continued*)

- Test105_Caching_SqlDependency_
 - Test Method, 184–185
- troubleshooting query notifications with, 178
- UnitRun, with NUnit tests, 179
- UpdateDatabase method, 281–282
- Url property, setting in FavoriteLink object, 195
- user controls
 - binding, 83–84
 - configuration, 230
 - declaring, 229
 - markup, 159
 - SubstitutionFragment, 157–158
- UserManager.ascx
 - control, 18–23
 - script, 23–29
- usernames and passwords, storing in a
 - Web.config file, 118
- users and roles, creating, 18–34
- Utility class
 - GetProduct method in, 158–159
 - in App_Code folder, 17
- Utility methods script, 17

V

- VaryByParam attribute, effect of wildcard use on, 154
- version numbers, changing, 202
- ViewState
 - as alternative to caching, 150
 - and databinding, 87
 - vs. ControlState, 89–91
 - debugging without, 109
 - disabling, 89
 - persisting manually, 105–106
 - Session and, 87–88
 - working with and without, 89, 107–108
- virtual environment, 1–4
- virtual hard drive, creating, 1–2
- VirtualPath property, 235–236
- Visible property, setting for AdminHyperlink, 117
- Visual Studio (VS) 2005, adding code snippets to, 45
- Visual Studio Professional Edition, Database Projects in, 56–57

W

- WCF provider
 - benefits of, 332–333
 - client configuration, 333, 336
 - configuring, 336–337
 - hosting the service, 335–336
 - IEventService interface, 334

- implementing, 332–337
- service configuration, 333–334, 337
- Web Deployment Projects
 - automating configuration changes, 262–264
 - configSource attribute option, 264
 - creating, 261–262
 - multiple replacement sections, 265–266
 - PostBuild deployments, 264–271
 - website for, 261
- web farms
 - distributing traffic with, 293–294
 - using shared machine key for, 129
- Web Form, adding ObjectDataSource to, 52
- web pages
 - performance considerations, 65–70
 - securing, 116
- web server, to database server
 - communications, 148
- Web Service Extensions (WSE), released by Microsoft, 332
- Web.config file, 234
 - custom, 13
 - datasource configuration in, 6–7
 - for PhotoAlbumProvider, 138
 - securing Admin section in, 34–35
 - storing usernames and passwords in, 118
- website
 - code and database separation, 7
 - CodePlex information, 242
 - controlling access by role, 115–117
 - controlling behavior by role, 117–118
 - copying files from source to destination directory, 261
 - customizing content to suit users, 119
 - deploying, 261–271
 - distributing content and traffic to multiple servers, 293–295
 - Web Development Projects, 261
 - Windows Live, 125
- Website classes, Photo Album provider, 379
- Website Project, creating class library, 218
- Windows Communication Foundation provider. *See* WCF provider
- WipeProviderData.sql script, 9–10

X

- xcopy command, 261
- XML implementations, using, 113
- XPath constants, 239
- XsdBuildProvider, .xsd mapped to, 234

Z

- zip file, defining name of in PropertyGroup, 271
- Zip task, declaration, 271