Microsoft

Programming Microsoft ASP.NET 3.5

Dino Esposito

Table of Contents

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Chapter 7

ADO.NET Data Providers

In this chapter:

ADO.NET is a data-access subsystem in the Microsoft .NET Framework. It was heavily inspired by ActiveX Data Objects (ADO), which has been for years a very successful object model for writing data-aware applications. The key design criteria for ADO.NET are simplicity and performance. Those criteria typically work against each other, but with ADO.NET you get the power and performance of a low-level interface combined with the simplicity of a modern object model. Unlike ADO, though, ADO.NET has been purposely designed to observe general, rather than database-oriented, guidelines.

Several syntactical differences exist between the object models of ADO and ADO.NET. In spite of this, the functionalities of ADO and ADO.NET look much the same. This is because Microsoft put a lot of effort into aligning some programming aspects of the ADO.NET object model with ADO. In this way, seasoned data developers new to .NET don't need to become familiar with too many new concepts and can work with a relatively short learning curve. With ADO.NET, you probably won't be able to reuse much of your existing code. You'll certainly be able, though, to reuse all your skills. At the same time, novice developers face a relatively simple and easy-to-understand model, with a consistent design and a powerful set of features.

The ADO.NET framework is made of two distinct, but closely related, sets of classes—data providers and data containers. We tackle providers in this chapter and reserve containers for the next.

.NET Data Access Infrastructure

ADO.NET is the latest in a long line of database access technologies that began with the Open Database Connectivity (ODBC) API several years ago. Written as a C-style library, ODBC was designed to provide a uniform API to issue SQL calls to various database servers. In the ODBC model, database-specific drivers hide any difference and discrepancy between the SQL

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

language used at the application level and the internal query engine. Next, COM landed in the database territory and started a colonization process that culminated with OLE DB.

OLE DB has evolved from ODBC and, in fact, the open database connectivity principle emerges somewhat intact in it. OLE DB is a COM-based API aimed at building a common layer of code for applications to access any data source that can be exposed as a tabular rowset of data. The OLE DB architecture is composed of two elements—a consumer and a provider. The consumer is incorporated in the client and is responsible for setting up COM-based communication with the data provider. The OLE DB data provider, in turn, receives calls from the consumer and executes commands on the data source. Whatever the data format and storage medium are, an OLE DB provider returns data formatted in a tabular layout—that is, with rows and columns. OLE DB uses COM to make client applications and data sources communicate.

Because it isn't especially easy to use and is primarily designed for coding from within C++ applications, OLE DB never captured the heart of programmers, even though it could guarantee a remarkable mix of performance and flexibility. Next came ADO—roughly, a COM automation version of OLE DB—just to make the OLE DB technology accessible from Microsoft Visual Basic and classic Active Server Pages (ASP) applications. When used, ADO acts as the real OLE DB consumer embedded in the host applications. ADO was invented in the age of connected, 2-tier applications, and the object model design reflects that. ADO makes a point of programming redundancy: it usually provides more than just one way of accomplishing key tasks, and it contains a lot of housekeeping code. For all these reasons, although it's incredibly easy to use, an ADO-based application doesn't perform as efficiently as a pure OLE DB application.

Note Using ADO in .NET applications is still possible, but for performance and consistency reasons its use should be limited to a few very special cases. For example, ADO is the only way you have to work with server cursors. In addition, ADO provides a schema management API to .NET Framework 1.x applications. On the other hand, ADO recordsets can't be directly bound to ASP.NET or Microsoft Windows Forms data-bound controls. We'll cover ASP.NET data binding in Chapter 9.

.NET Managed Data Providers

A key architectural element in the ADO.NET infrastructure is the *managed provider*, which can be considered as the .NET counterpart of the OLE DB provider. A managed data provider enables you to connect to a data source and retrieve and modify data. Compared to the OLE DB provider, a .NET managed provider has a simplified data access architecture made of a smaller set of interfaces and based on .NET Framework data types.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Note The .NET Framework 3.5 introduces new language features fully supported by Visual Studio 2008. One of the most important is the .NET Language Integrated Query, or LINQ for short. Essentially, LINQ helps make querying data a first-class programming concept. Developers can use LINQ with any data source and can express query behavior natively in the language of choice, optionally transforming query results into whatever format is desired. LINQ-enabled languages, such as the newest versions of C# and Visual Basic .NET, provide full type-safety and compile-time checking of query expressions. The .NET Framework 3.5 ships with built-in libraries that enable LINQ queries against collections, XML, and databases. In particular, LINQ-to-SQL is an object/relational mapping (O/RM) implementation that allows you to model a relational database (for example, a SQL Server database) using .NET classes. As a result, you can query the database tables and views using the LINQ syntax, and you can update, insert, and delete data from it. In addition, LINQ-to-SQL fully supports transactions and stored procedures. I'll cover LINQ-to-SQL in Chapter 10. However, for a wider and deeper coverage of the whole LINQ platform, you might want to take a look at the book *Introducing Microsoft LINQ* (Microsoft Press, 2007).

Building Blocks of a .NET Data Provider

The classes in the managed provider interact with the specific data source and return data to the application using the data types defined in the .NET Framework. The logical components implemented in a managed provider are those graphically featured in Figure 7-1.

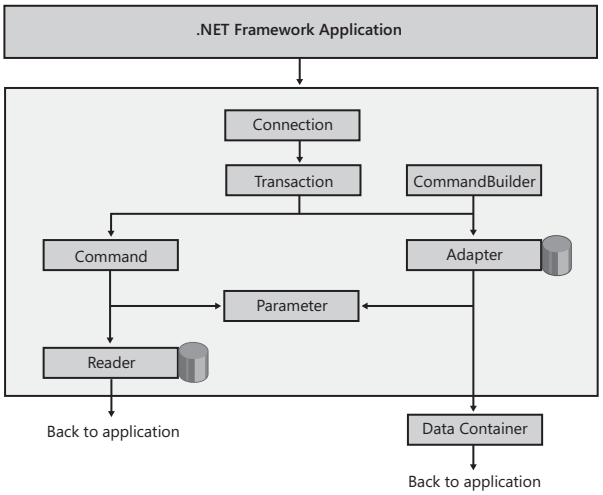

FIGURE 7-1 The .NET Framework classes that form a typical managed provider and their interconnections.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The functionalities supplied by a .NET data provider fall into a couple of categories:

- Support for disconnected data—that is, the capability of populating ADO.NET container classes with fresh data
- Support for connected data access, which includes the capability of setting up a connection and executing a command

Table 7-1 details the principal components of a .NET data provider.

TABLE 7-1 Principal Components of a .NET Data Provider

Component	Description
Connection	Creates a connection with the specified data source, including SQL Server, Oracle, and any data source for which you can indicate either an OLE DB provider or an ODBC driver
Transaction	Represents a transaction to be made in the source database server
Command	Represents a command that hits the underlying database server
Parameter	Represents a parameter you can pass to the command object
DataAdapter	Represents a database command that executes on the specified database server and returns a disconnected set of records
CommandBuilder	Represents a helper object that automatically generates commands and parameters for a *DataAdapter*
DataReader	Represents a read-only, forward-only cursor created on the underlying database server

Each managed provider that wraps a real-world database server implements all the objects in Table 7-1 in a way that is specific to the data source.

Caution You won't find in the .NET Framework any class named *Connection*. You'll find instead several connection-like classes, one for each supported .NET managed provider (for example, *SqlConnection* and *OracleConnection*). The same holds true for the other objects listed in the table.

Interfaces of a .NET Data Provider

The components listed in Table 7-1 are implemented based on methods and properties defined by the interfaces you see in Table 7-2.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

TABLE 7-2 Interfaces of .NET Data Providers

Interface	Description
IDbConnection	Represents a unique session with a data source
IDbTransaction	Represents a local, nondistributed transaction
IDbCommand	Represents a command that executes when connected to a data source
IDataParameter	Allows implementation of a parameter to a command
IDataReader	Reads a forward-only, read-only stream of data created after the execution of a command
IDataAdapter	Populates a *DataSet* object, and resolves changes in the *DataSet* object back to the data source
IDbDataAdapter	Supplies methods to execute typical operations on relational databases (such as insert, update, select, and delete)

Note that all these interfaces except *IDataAdapter* are officially considered to be optional. However, any realistic data provider that manages a database server would implement them all.

Note Individual managed providers are in no way limited to implementing all and only the interfaces listed in Table 7-2. Based on the capabilities of the underlying data source and its own level of abstraction, each managed provider can expose more components. A good example of this is the data provider for Microsoft SQL Server that you get in .NET Framework 2.0 and newer versions. It adds several additional classes to handle special operations, such as bulk copy, data dependency, and connection string building.

Managed Providers vs. OLE DB Providers

OLE DB providers and managed data providers are radically different types of components that share a common goal—to provide a unique and uniform programming interface for data access. The differences between OLE DB providers and .NET data providers can be summarized in the following points:

- **Component Technology** OLE DB providers are in-process COM servers that expose a suite of COM interfaces to consumer modules. The dialog between consumers and providers takes place through COM and involves a number of interfaces. A .NET data provider is a suite of managed classes whose overall design looks into one *particular* data source rather than blinking at an abstract and universal data source, as is the case with OLE DB.
- **Internal Implementation** Both types of providers end up making calls into the data-source programming API. In doing so, though, they provide a dense layer of code that separates the data source from the calling application. Learning from the OLE DB experience, Microsoft designed .NET data providers to be more agile and simple. Fewer

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

interfaces are involved, and the conversation between the caller and the callee is more direct and as informal as possible.

- **Application Integration** Another aspect of .NET that makes the conversation between caller and callee more informal is the fact that managed providers return data using the same data structures that the application would use to store it. In OLE DB, the data-retrieval process is more flexible, but it's also more complex because the provider packs data in flat memory buffers and leaves the consumer responsible for mapping that data into usable data structures.

Calling into an OLE DB provider from within a .NET application is more expensive because of the data conversion necessary to make the transition from the managed environment of the common language runtime (CLR) to the COM world. Calling a COM object from within a .NET application is possible through the COM interop layer, but doing so comes at a cost. In general, to access a data source from within a .NET application, you should always use a managed provider instead of OLE DB providers or ODBC drivers. You should be doing this primarily because of the transition costs, but also because managed providers are normally more modern tools based on an optimized architecture.

Some data sources, though, might not have a .NET data provider available. In these cases, resorting to old-fashioned OLE DB providers or ODBC drivers is a pure necessity. For this reason, the .NET Framework encapsulates in managed wrapper classes the logic needed to call into a COM-style OLE DB provider or a C-style ODBC driver.

Data Sources You Access Through ADO.NET

The .NET data provider is the managed component of choice for database vendors to expose their data in the most effective way. Ideally, each database vendor should provide a .NET-compatible API that is seamlessly callable from within managed applications. Unfortunately, this is not always the case. However, at least for the major database management systems (DBMS), a managed data provider can be obtained from either Microsoft or third-party vendors.

As of version 3.5, the .NET Framework supports the data providers listed in Table 7-3.

TABLE 7-3 Managed Data Providers in .NET

Data Source	Namespace	Description
SQL Server	*System.Data.SqlClient*	Targets various versions of SQL Server, including SQL Server 7.0, SQL Server 2000, and the newest SQL Server 2005
SQL Server CE	*Microsoft.SqlServerCe.Client*	Targets SQL Server 2005 Mobile Edition
OLE DB providers	*System.Data.OleDb*	Targets OLE DB providers, including SQLOLEDB, MSDAORA, and the Jet engine

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Data Source	Namespace	Description
ODBC drivers	*System.Data.Odbc*	Targets several ODBC drivers, including those for SQL Server, Oracle, and the Jet engine
Oracle	*System.Data.OracleClient*	Targets Oracle 9i, and supports all of its data types

The OLE DB and ODBC managed providers listed in Table 7-3 are not specific to a physical database server, but rather they serve as a bridge that gives instant access to a large number of existing OLE DB providers and ODBC drivers. When you call into OLE DB providers, your .NET applications jumps out of the managed environment and issues COM calls through the COM interop layer.

Accessing SQL Server

As mentioned, Microsoft supplies a managed provider for SQL Server 7.0 and newer versions. Using the classes contained in this provider is by far the most effective way of accessing SQL Server. Figure 7-2 shows how SQL Server is accessed by .NET and COM clients.

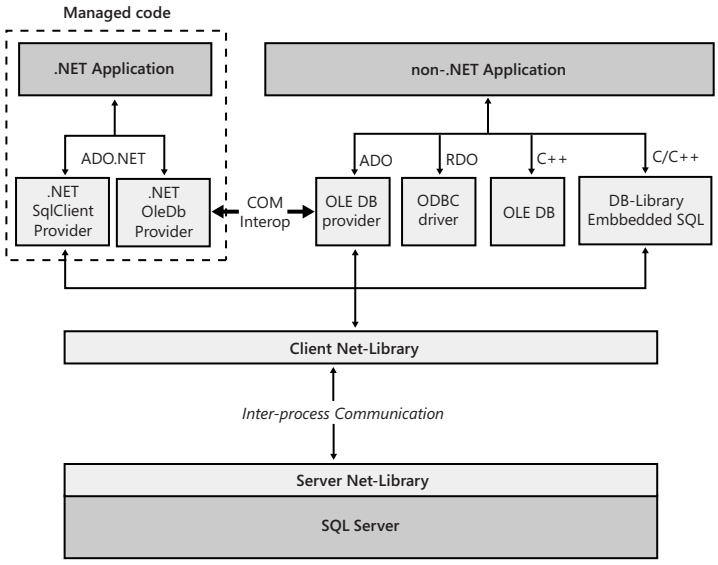

FIGURE 7-2 Accessing SQL Server by using the managed provider for OLE DB adds overhead because the objects called must pass through the COM interop layer.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

A .NET application should always access a SQL Server database using the native data provider. Although it's possible to do so, you should have a good reason to opt for an alternative approach like using the OLE DB provider for SQL Server (named SQLOLEDB). A possible good reason is the need to use ADO rather than ADO.NET as the data-access library. The SQL Server native provider not only avoids paying the performance tax of going down to COM, it also implements some little optimizations when preparing the command for SQL Server.

Accessing Oracle Databases

The .NET Framework includes a managed provider for Oracle databases. The classes are located in the *System.Data.OracleClient* namespace in the *System.Data.OracleClient* assembly. Instead of using the managed provider, you can resort to the COM-based OLE DB provider (named MSDAORA) or the ODBC driver. Note, though, that the Microsoft OLE DB provider for Oracle does not support Oracle 9i and its specific data types. In contrast, Oracle 9i data types are fully supported by the .NET managed provider. So by using the .NET component to connect to Oracle, you not only get a performance boost but also increased programming power.

Note The .NET data provider for Oracle requires Oracle client software (version 8.1.7 or later) to be installed on the system before you can use it to connect to an Oracle data source.

Microsoft is not the only company to develop a .NET data provider for Oracle databases. Data Direct, Core Lab, and Oracle itself also ship one. Each provider has its own set of features; for example, the Oracle provider (named ODP.NET) has many optimizations for retrieving and manipulating Oracle native types, such as any flavor of large objects (LOBs) and REF cursors. ODP.NET can participate in transactional applications, with the Oracle database acting as the resource manager and the Microsoft Distributed Transaction Coordinator (DTC) coordinating transactions.

Using OLE DB Providers

The .NET data provider for OLE DB providers is a data-access bridge that allows .NET applications to call into data sources for which a COM OLE DB provider exists. While this approach is architecturally less effective than using native providers, it is the only way to access those data sources when no managed providers are available.

The classes in the *System.Data.OleDb* namespace, though, don't support all types of OLE DB providers and have been optimized to work with only a few of them, as listed in Table 7-4.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

TABLE 7-4 OLE DB Providers Tested

Name	Description
Microsoft.Jet.OLEDB.4.0	The OLE DB provider for the Jet engine implemented in Microsoft Access
MSDAORA	The Microsoft OLE DB provider for Oracle 7 that partially supports some features in Oracle 8
SQLOLEDB	The OLE DB provider for SQL Server 6.5 and newer

The preceding list does not include all the OLE DB providers that really work through the OLE DB .NET data provider. However, only the components in Table 7-4 are guaranteed to work well in .NET. In particular, the classes in the *System.Data.OleDb* namespace don't support OLE DB providers that implement any of the OLE DB 2.5 interfaces for semistructured and hierarchical rowsets. This includes the OLE DB providers for Exchange (EXOLEDB) and for Internet Publishing (MSDAIPP).

In general, what really prevents existing OLE DB providers from working properly within the .NET data provider for OLE DB is the set of interfaces they really implement. Some OLE DB providers—for example, those written using the Active Template Library (ATL) or with Visual Basic and the OLE DB Simple Provider Toolkit—are likely to miss one or more COM interfaces that the .NET wrapper requires.

Using ODBC Drivers

The .NET data provider for ODBC lets you access ODBC drivers from managed, ADO.NET-driven applications. Although the ODBC .NET data provider is intended to work with all compliant ODBC drivers, it is guaranteed to work well only with the drivers for SQL Server, Oracle, and Jet. Although ODBC might appear to now be an obsolete technology, it is still used in several production environments, and for some vendors it is still the only way to connect to their products.

You can't access an ODBC driver through an OLE DB provider. There's no technical reason behind this limitation—it's just a matter of common sense. In fact, calling the MSDASQL OLE DB provider from within a .NET application would drive your client through a double data-access bridge—one going from .NET to the OLE DB provider, and one going one level down to the actual ODBC driver.

The Provider Factory Model

ADO.NET takes into careful account the particularity of each DBMS and provides a programming model tailor-made for each one. All .NET data providers share a limited set of common features, but each has unique capabilities. The communication between the user code and the DBMS takes place more directly using ADO.NET. This model works better and faster and is probably clearer to most programmers.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

But until version 2.0 of the .NET Framework, ADO.NET has one key snag. Developers must know in advance the data source they're going to access. Generic programming—that is, programming in which the same code targets different data sources at different times—is hard (but not impossible) to do. You can create a generic command object and a generic data reader, but not a generic data adapter and certainly not a generic connection. However, through the *IDbConnection* interface, you can work with a connection object without knowing the underlying data source. But you can never create a connection object in a weakly typed manner—that is, without the help of the *new* operator.

Instantiating Providers Programmatically

Starting with version 2.0, ADO.NET enhances the provider architecture and introduces the factory class. Each .NET data provider encompasses a factory class derived from the base class *DbProviderFactory*. A factory class represents a common entry point for a variety of services specific to the provider. Table 7-5 lists the main methods of a factory class.

TABLE 7-5 Principal Methods of a Factory Class

Method	Description
CreateCommand	Returns a provider-specific command object
CreateCommandBuilder	Returns a provider-specific command builder object
CreateConnection	Returns a provider-specific connection object
CreateDataAdapter	Returns a provider-specific data adapter object
CreateParameter	Returns a provider-specific parameter object

How do you get the factory of a particular provider? By using a new class, *DbProviderFactories*, that has a few static methods. The following code demonstrates how to obtain a factory object for the SQL Server provider:

```
DbProviderFactory fact;
fact = DbProviderFactories.GetFactory("System.Data.SqlClient");
```

The *GetFactory* method takes a string that represents the invariant name of the provider. This name is hard-coded for each provider in the configuration file where it is registered. By convention, the provider name equals its unique namespace.

GetFactory enumerates all the registered providers and gets assembly and class name information for the matching invariant name. The factory class is not instantiated directly. Instead, the method uses reflection to retrieve the value of the static *Instance* property of the factory class. The property returns the instance of the factory class to use. Once you hold a factory object, you can call any of the methods listed earlier in Table 7-5.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following pseudocode gives you an idea of the internal implementation of the *CreateConnection* method for the *SqlClientFactory* class—the factory class for the SQL Server .NET data provider:

```
public DbConnection CreateConnection()
{
    return new SqlConnection();
}
```

Enumerating Installed Data Providers

You can use all .NET data providers registered in the configuration file. The following excerpt is from the *machine.config* file:

```
<system.data>
  <DbProviderFactories>
    <add name="SqlClient Data Provider"
         invariant="System.Data.SqlClient"
         description=".Net Framework Data Provider for SqlServer"
         type="System.Data.SqlClient.SqlClientFactory, System.Data "/>
    <add name="OracleClient Data Provider"
         invariant="System.Data.OracleClient"
         description=".Net Framework Data Provider for Oracle"
         type="System.Data.OracleClient.OracleFactory,
               System.Data.OracleClient" />
    ...
  </DbProviderFactories>
</system.data>
```

Each provider is characterized by an invariant name, a description, and a type that contains assembly and class information. The *GetFactoryClasses* method on the *DbProviderFactories* class returns this information packed in an easy-to-use *DataTable* object. The following sample page demonstrates how to get a quick list of the installed providers:

```
<%@ page language="C#" %>
<%@ import namespace="System.Data" %>
<%@ import namespace="System.Data.Common" %>

<script runat="server">
    void Page_Load (object sender, EventArgs e)
    {
        DataTable providers = DbProviderFactories.GetFactoryClasses();
        provList.DataSource = providers;
        provList.DataBind();
    }
</script>
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
<html>
<head runat="server"><title>List Factory Objects</title></head>
<body>
    <form runat="server">
        <asp:datagrid runat="server" id="provList" />
    </form>
</body>
</html>
```

The final page is shown in Figure 7-3.

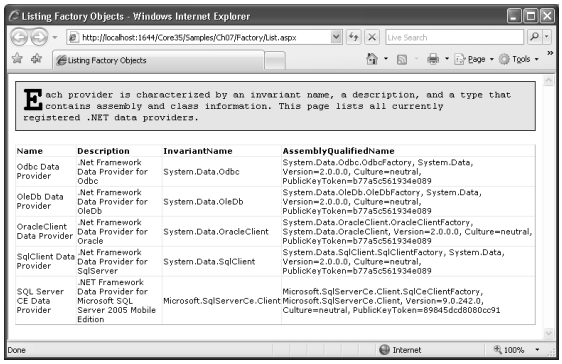

FIGURE 7-3 The list of the installed .NET data providers.

Database-Agnostic Pages

Let's write out some sample code to demonstrate how to craft database-agnostic pages. The sample page will contain three text boxes to collect the name of the provider, connection string, and command text.

```
protected void RunButton_Click(object sender, EventArgs e)
{
    string provider = ProviderNameBox.Text;
    string connString = ConnectionStringBox.Text;
    string commandText = CommandTextBox.Text;

    // Get the provider
    DbProviderFactory fact = DbProviderFactories.GetFactory(provider);

    // Create the connection
    DbConnection conn = fact.CreateConnection();
    conn.ConnectionString = connString;
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    // Create the data adapter
    DbDataAdapter adapter = fact.CreateDataAdapter();
    adapter.SelectCommand = conn.CreateCommand();
    adapter.SelectCommand.CommandText = commandText;

    // Run the query
    DataTable table = new DataTable();
    adapter.Fill(table);

    // Shows the results
    Results.DataSource = table;
    Results.DataBind();
}
```

By changing the provider name and properly adapting the connection string and command, the same core code can now be used to work on other database servers.

Caution Nothing presented here is invented; no magic and no tricks apply. This said, though, don't be fooled by the apparent simplicity of this approach. Be aware that in real-world applications data access is normally insulated in the boundaries of the Data Access Layer (DAL), and that practice suggests you have one DAL per supported data source. This is because the complexity of real problems need to be addressed by getting the most out of each data server. In the end, you need optimized data access code to take advantage of all the features of a given DBMS rather than generic code that you write once and which queries everywhere and everything.

Connecting to Data Sources

The ADO.NET programming model is based on a relatively standard and database-independent sequence of steps. You first create a connection, then prepare and execute a command, and finally process the data retrieved. As far as basic operations and data types are involved, this model works for most providers. Some exceptions are BLOB fields management for Oracle databases and perhaps bulk copy and XML data management for SQL Server databases.

In the rest of the chapter, we'll mostly discuss how ADO.NET data classes work with SQL Server (from version 7.0 onward). However, we'll promptly point out any aspect that is significantly different than other .NET data providers. To start out, let's see how connections take place.

More Info For in-depth coverage of ADO.NET 2.0, see *Programming Microsoft ADO.NET 2.0 Applications: Advanced Topics* by Glenn Johnson (Microsoft Press, 2005) and *Programming ADO.NET 2.0 Core Reference* by David Sceppa (Microsoft Press, 2005)

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273 Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
Publisher: Microsoft Press
Print Publication Date: 2008/02/23 User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The *SqlConnection* Class

The first step in working with an ADO.NET-based application is setting up the connection with the data source. The class that represents a physical connection to SQL Server is *SqlConnection*, and it is located in the *System.Data.SqlClient* namespace. The class is sealed (that is, not inheritable) and implements the *IDbConnection* interface. In ADO.NET, the interface is implemented through the intermediate base class *DbConnection*, which also provides additional features shared by all providers. (In fact, adding new members to the interface would have broken existing code.)

The *SqlConnection* class features two constructors, one of which is the default parameterless constructor. The second class constructor, on the other hand, takes a string containing the connection string:

```
public SqlConnection();
public SqlConnection(string);
```

The following code snippet shows the typical way to set up and open a SQL Server connection:

```
string connString = "SERVER=...;DATABASE=...;UID=...;PWD=...";
SqlConnection conn = new SqlConnection(connString);
conn.Open();
...
conn.Close();
```

Properties of the *SqlConnection* Class

Table 7-6 details the public properties defined on the *SqlConnection* class.

TABLE 7-6 Properties of the *SqlConnection* Class

Property	*IDbConnection* Interface Property?	Description
ConnectionString	Yes	Gets or sets the string used to open the database.
ConnectionTimeout	Yes	Gets the number of seconds to wait while trying to establish a connection.
Database	Yes	Gets the name of the database to be used.
DataSource		Gets the name of the instance of SQL Server to connect to. It corresponds to the *Server* connection string attribute.
PacketSize	No	Gets the size in bytes of network packets used to communicate with SQL Server. Set to 8192, it can be any value in the range from 512 through 32,767.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Property	*IDbConnection* Interface Property?	Description
ServerVersion	No	Gets a string containing the version of the current instance of SQL Server. The version string is in the form of *major.minor.release*.
State	Yes	Gets the current state of the connection: open or closed. Closed is the default.
StatisticsEnabled	No	Enables the retrieval of statistical information over the current connection. *Not available in ADO.NET 1.x.*
WorkStationId	No	Gets the network name of the client, which normally corresponds to the *WorkStation ID* connection string attribute.

An important characteristic to note about the properties of the connection classes is that they are all read-only except *ConnectionString*. In other words, you can configure the connection only through the tokens of the connection string, but you can read attributes back through handy properties.

Note This characteristic of connection class properties in ADO.NET is significantly different than what you find in ADO, where many of the connection properties—for example, *ConnectionTimeout* and *Database*—were read/write.

Methods of the *SqlConnection* Class

Table 7-7 shows the methods available in the *SqlConnection* class.

TABLE 7-7 Methods of the *SqlConnection* Class

Method	*IDbConnection* Interface Method?	Description
BeginTransaction	Yes	Begins a database transaction. Allows you to specify a name and an isolation level.
ChangeDatabase	Yes	Changes the current database on the connection. Requires a valid database name.
Close	Yes	Closes the connection to the database. Use this method to close an open connection.
CreateCommand	Yes	Creates and returns a *SqlCommand* object associated with the connection.
Dispose	No	Calls *Close*.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Method	*IDbConnection* Interface Method?	Description
EnlistDistributedTransaction	No	If auto-enlistment is disabled, enlists the connection in the specified distributed Enterprise Services DTC transaction. *Not available in ADO.NET 1.0.*
EnlistTransaction	No, but defined on *DbConnection*	Enlists the connection on the specified local or distributed transaction. *Not available in ADO.NET 1.x.*
GetSchema	No, but defined on *DbConnection*	Retrieve schema information for the specified scope (that is, tables, databases). *Not available in ADO.NET 1.x.*
ResetStatistics	No	Resets the statistics service. *Not available in ADO.NET 1.x.*
RetrieveStatistics	No	Gets a hash table filled with the information about the connection, such as data transferred, user details, and transactions. *Not available in ADO.NET 1.x.*
Open	Yes	Opens a database connection.

Note that if the connection goes out of scope, it is not automatically closed. The garbage collector will eventually pick up the object instance, but the connection won't be closed because the garbage collector can't recognize the peculiarity of the object and handle it properly. Therefore, you must explicitly close the connection by calling *Close* or *Dispose* before the object goes out of scope.

Note Like many other disposable objects, connection classes implement the *IDisposable* interface, thus providing a programming interface for developers to dispose of the object. The dispose pattern entails the sole *Dispose* method; *Close* is not officially part of the pattern, but most classes implement it as well.

Changing Passwords

The *SqlConnection* class provides a static method named *ChangePassword* to let developers change the SQL Server password for the user indicated in the supplied connection string:

```
public static void ChangePassword(
    string connectionString, string newPassword)
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

An exception will be thrown if the connection string requires integrated security (that is, *Integrated Security=True* or an equivalent setting). The method opens a new connection to the server, requests the password change, and closes the connection once it has completed. The connection used to change the password is not taken out of the connection pool. The new password must comply with any password security policy set on the server, including minimum length and requirements for specific characters.

Note that *ChangePassword* works only on SQL Server 2005.

Accessing Schema Information

In ADO.NET 2.0 and beyond, all managed providers are expected to implement a *GetSchema* method for retrieving schema information. The standard providers offer the following overloads of the method:

```
public override DataTable GetSchema();
public override DataTable GetSchema(string collection);
public override DataTable GetSchema(string collection, string[] filterVal)
```

The schema information you can retrieve is specific to the back-end system. For the full list of valid values, call *GetSchema* with no parameters. The following code shows how to retrieve all available collections and bind the results to a drop-down list:

```
// Get schema collections
SqlConnection conn = new SqlConnection(connString);
conn.Open();
DataTable table = conn.GetSchema();
conn.Close();

// Display their names
CollectionNames.DataSource = table;
CollectionNames.DataTextField = "collectionname";
CollectionNames.DataBind();
```

Figure 7-4 shows the available schema collections for a SQL Server 2000 machine. (For SQL Server 2005, it adds only a *UserDefinedTypes* collection.) Call *GetSchema* on, say, the *Databases* collection and you will get the list of all databases for the instance of SQL Server you are connected to. Likewise, if you call it on *Tables*, you will see the tables in the connected database.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

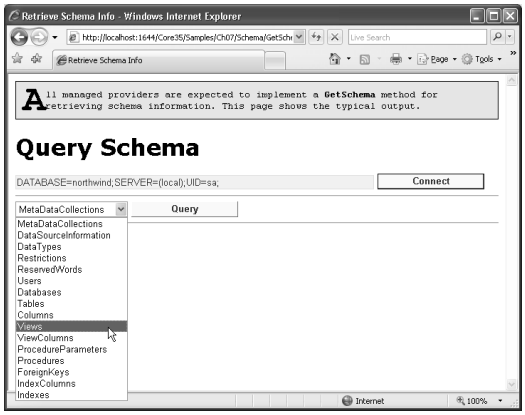

FIGURE 7-4 The list of available schema collections for a SQL Server database

Note The preceding code snippet introduces the *DataTable* class as well as data binding. We will cover the *DataTable* class—one of the most important ADO.NET container classes—in the next chapter. Data binding, on the other hand, will be the subject of Chapter 10.

The list of schema collections is expressed as a *DataTable* object with three columns—*CollectionName* is the column with names. The following code shows how to retrieve schemainformation regarding the collection name currently selected in the drop-down list—the *Views*:

```
string coll = CollectionNames.SelectedValue;
string connString = ConnStringBox.Text;
SqlConnection conn = new SqlConnection(connString);
conn.Open();
DataTable table = conn.GetSchema(coll);
conn.Close();
GridView1.DataSource = table;
GridView1.DataBind();
```

As Figure 7-5 demonstrates, the data is then bound to a grid for display.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

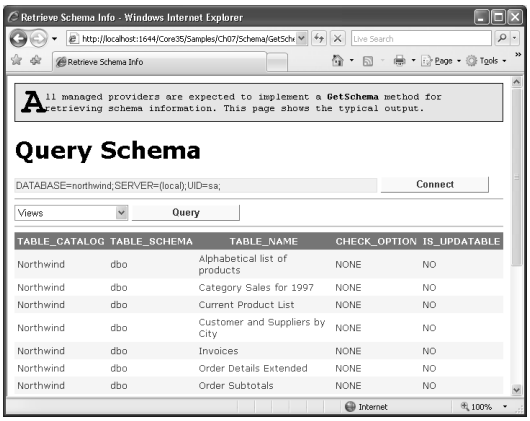

FIGURE 7-5 The list of views found in the Northwind database.

In ADO.NET 2.0 and beyond, all connection objects support *GetSchema* methods, as they are part of the new intermediate *DbConnection* class. In ADO.NET 1.x, you have different approaches depending on the target source. If you work with OLE DB, you get schema information through the OLE DB native provider calling the *GetOleDbSchemaTable* method. The following code shows how to get table information:

```
OleDbConnection conn = new OleDbConnection(connString);
conn.Open();
DataTable schema = cConn.GetOleDbSchemaTable(
    OleDbSchemaGuid.Tables,
    new object[] {null, null, null, "TABLE"});
conn.Close();
```

GetOleDbSchemaTable takes an *OleDbSchemaGuid* argument that identifies the schema information to return. In addition, it takes an array of values to restrict the returned columns. *GetOleDbSchemaTable* returns a *DataTable* populated with the schema information. Alternately, you can get information on available databases, tables, views, constraints, and so on through any functionality provided by the specific data source, such as stored procedures and views.

```
SqlConnection conn = new SqlConnection(connString);
SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT * FROM INFORMATION_SCHEMA.TABLES " +
            "WHERE TABLE_TYPE = 'BASE TABLE' " +
            "ORDER BY TABLE_TYPE", conn);
DataTable schema = new DataTable();
adapter.Fill(schema);
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The *GetSchema* method unifies the approach for retrieving schema information. The *SqlDataAdapter* class that appears in the preceding code snippet is a special type of command that we'll explore in depth in the next chapter. One of its key characteristics is that it returns disconnected data packed in a *DataTable* or *DataSet*.

Connection Strings

To connect to a data source, you need a connection string. Typically made of semicolon-separated pairs of names and values, a connection string specifies settings for the database runtime. Typical information contained in a connection string includes the name of the database, location of the server, and user credentials. Other more operational information, such as connection timeout and connection pooling settings, can be specified too.

In many enterprise applications, the usage of connection strings is related to a couple of issues: how to store and protect them, and how to build and manage them. The .NET Framework provides excellent solutions to both issues as we'll see in a moment.

Needless to say, connection strings are database specific, although huge differences don't exist between, say, a connection string for SQL Server and Oracle databases. In this chapter, we mainly focus on SQL Server but point out significant differences.

Configuring Connection Properties

The *ConnectionString* property of the connection class can be set only when the connection is closed. Many connection string values have corresponding read-only properties in the connection class. These properties are updated when the connection string is set. The contents of the connection string are checked and parsed immediately after the *ConnectionString* property is set. Attribute names in a connection string are not case sensitive, and if a given name appears multiple times, the value of the last occurrence is used. Table 7-8 lists the keywords that are supported.

TABLE 7-8 Connection String Keywords for SQL Server

Keyword	Description
Application Name	Name of the client application as it appears in the SQL Profiler. Defaults to *.Net SqlClient Data Provider.*
Async	When *true*, enables asynchronous operation support. *Not supported in ADO.NET 1.x.*
AttachDBFileName or Initial File Name	The full path name of the file (.mdf) to use as an attachable database file.
Connection Timeout	The number of seconds to wait for the connection to take place. Default is 15 seconds.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Keyword	Description
Current Language	The SQL Server language name.
Database or Initial Catalog	The name of the database to connect to.
Encrypt	Indicates whether Secure Sockets Layer (SSL) encryption should be used for all data sent between the client and server. Needs a certificate installed on the server. Default is *false*.
Failover Partner	The name of the partner server to access in case of errors. Connection failover allows an application to connect to an alternate, or backup, database server if the primary database server is unavailable. *Not supported in ADO.NET 1.x*.
Integrated Security or Trusted_Connection	Indicates whether current Windows account credentials are used for authentication. When set to *false*, explicit user ID and password need to be provided. The special value *sspi* equals *true*. Default is *false*.
MultipleActiveResultSets	When *true*, an application can maintain multiple active result sets. Set to *true* by default, this feature requires SQL Server 2005. *Not supported in ADO.NET 1.x*.
Network Library or Net	Indicates the network library used to establish a connection to SQL Server. Default is *dbmssocn*, which is based on TCP/IP.
Packet Size	Bytes that indicate the size of the packet being exchanged. Default is 8192.
Password or pwd	Password for the account logging on.
Persist Security Info	Indicates whether the managed provider should include password information in the string returned as the connection string. Default is *false*.
Server or Data Source	Name or network address of the instance of SQL Server to connect to.
User ID or uid	User name for the account logging on.
Workstation ID	Name of the machine connecting to SQL Server.

The network dynamic-link library (DLL) specified by the *Network Library* keyword must be installed on the system to which you connect. If you use a local server, the default library is *dbmslpcn*, which uses shared memory. For a list of options, consult the MSDN documentation.

Any attempt to connect to an instance of SQL Server should not exceed a given time. The *Connection Timeout* keyword controls just this. Note that a connection timeout of 0 causes the connection attempt to wait indefinitely; it does not indicate no wait time.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

You normally shouldn't change the default packet size, which has been determined based on average operations and workload. However, if you're going to perform bulk operations in which large objects are involved, increasing the packet size can be of help because it decreases the number of reads and writes.

Some of the attributes you see listed in Table 7-8 are specific to ADO.NET 2.0 (and newer versions) and address features that have been introduced lately. They are asynchronous commands and multiple active result sets (MARS). MARS, in particular, removes a long-time constraint of the SQL Server programming model—that is, the constraint of having at most one pending request on a given session at a time. Before ADO.NET 2.0 and SQL Server 2005, several approaches have been tried to work around this limitation, the most common of which is using server-side cursors through ADO. We'll return to MARS later, in the section dedicated to SQL Server 2005.

Connection String Builders

How do you build the connection string to be used in an application? In many cases, you just consider it constant and read it out of a secured source. In other cases, though, you might need to construct it based on user input—for example, when retrieving user ID and password information from a dialog box. In ADO.NET 1.x, you can build the string only by blindly concatenating any name/value pairs. There are two major drawbacks with this technique. One is, the use of wrong keywords is caught only when the application goes under testing. More serious than the lack of a compile-time check, though, is that a blind pair concatenation leaves room for undesired data injections to attach users to a different database or to change in some way the final goal of the connection. Any measures to fend off injections and check the syntax should be manually coded, resulting in a specialized builder class—just like the brand new connection string builder classes you find in ADO.NET.

All default data providers support connection string builders in a guise that perfectly applies to the underlying provider. The following code snippet (and its result, shown in Figure 7-6) builds and displays a connection string for SQL Server:

```
SqlConnectionStringBuilder builder = new SqlConnectionStringBuilder();
builder.DataSource = serverName;
builder.UserID = userid;
builder.Password = pswd;
NewConnString.Text = builder.ConnectionString;
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

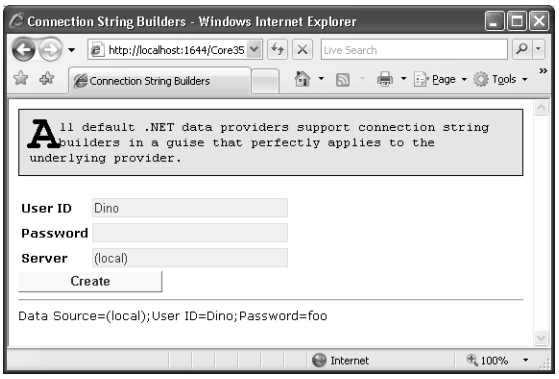

FIGURE 7-6 Building connection strings programmatically and securely.

By using connection string builders, you gain a lot in terms of security because you dramatically reduce injection. Imagine that a malicious user types in the password: *Foo;Trusted_Connection=true*. If you blindly concatenate strings, you might get the following:

```
Password=Foo;Trusted_Connection=true
```

Because the last pair wins, the connection will be opened based on the credentials of the logged-on user. If you use the builder class, you get the following appropriately quoted string:

```
Password="Foo;Trusted_Connection=true"
```

In addition, the builder class exposes all the supported 20+ keywords through easier-to-remember properties recognized by Microsoft IntelliSense.

Storing and Retrieving Connection Strings

Savvy developers avoid hard-coding connection strings in the compiled code. Configuration files (such as the *web.config* file) purposely support the <*appSettings*> named section, which is used to store custom data through name/value pairs. All these values populate the *AppSettings* collection and can be easily retrieved programmatically, as shown here:

```
string connString = ConfigurationSettings.AppSettings["NorthwindConn"];
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

This approach is far from perfect for two reasons. First, connection strings are not just data—they're a special kind of data not to be mixed up with general-purpose application settings. Second, connection strings are a critical parameter for the application and typically contain sensitive data. Therefore, at a minimum they need transparent encryption. Let's tackle storage first.

In the .NET Framework 2.0 and beyond, configuration files define a new section specifically designed to contain connection strings. The section is named *<connectionStrings>* and is laid out as follows:

```
<connectionStrings>
    <add name="NWind"
        connectionString="SERVER=...;DATABASE=...;UID=...;PWD=...;"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

You can manipulate the contents of the section by using *<add>*, *<remove>*, and *<clear>* nodes. You use an *<add>* node to add a new connection string to the current list, *<remove>* to remove a previously defined connection, and *<clear>* to reset all connections and create a new collection. By placing a *web.config* file in each of the application's directories, you can customize the collection of connection strings that are visible to the pages in the directory. Configuration files located in child directories can remove, clear, and extend the list of connection strings defined at the upper level. Note that each stored connection is identified with a name. This name references the actual connection parameters throughout the application. Connection names are also used within the configuration file to link a connection string to other sections, such as the *<providers>* section of *<membership>* and *<profile>* nodes.

All the connection strings defined in the *web.config* file are loaded into the new *ConfigurationManager.ConnectionStrings* collection. To physically open a connection based on a string stored in the *web.config* file, use following code:

```
string connStr;
connStr = ConfigurationManager.ConnectionStrings["NWind"].ConnectionString;
SqlConnection conn = new SqlConnection(connStr);
```

The full support from the configuration API opens up an interesting possibility for consuming connection strings—declarative binding. As we'll see in Chapter 9, ASP.NET 2.0 supports quite a few data source objects. A data source object is a server control that manages all aspects of data source interaction, including connection setup and command execution. You bind data source objects to data-bound controls and instruct the data source to retrieve data from a specific source. The great news is that you can now indicate the connection string declaratively, as follows:

```
<asp:SqlDataSource id="MySource" runat="server"
  ProviderName="System.Data.SqlClient"
  ConnectionString='<%#
    ConfigurationSettings.ConnectionStrings["NWind"].ConnectionString %>'
  SelectCommand="SELECT * FROM employees">
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

There's a lot more to be known about this feature, though, and we'll delve deeply into that later in the book. For now, it suffices to say that connection strings are much more than strings in the .NET Framework 2.0 and beyond.

Protecting Connection Strings

Starting with version 2.0, ASP.NET introduces a system for protecting sensitive data stored in the configuration system. It uses industry-standard XML encryption to encrypt specific sections of configuration files that might contain sensitive data. XML encryption (which you can learn more about at *http://www.w3.org/TR/xmlenc-core*) is a way to encrypt data and represent the result in XML. Prior to version 2.0, only a few specific ASP.NET sections that contain sensitive data support protection of this data using a machine-specific encryption in a registry key. This approach requires developers to come up with a utility to protect their own secrets—typically connection strings, credentials, and encryption keys.

In the newest versions of the .NET Framework, encryption of configuration sections is optional, and you can enable it for any configuration sections you want by referencing the name of the section in the *<protectedData>* section of the *web.config* file, as shown here:

```
<protectedData>
    <protectedDataSections>
        <add name="connectionStrings"
            provider="RSAProtectedConfigurationProvider" />
    </protectedDataSections>
</protectedData>
```

You can specify the type of encryption you want by selecting the appropriate provider from the list of available encryption providers. The .NET Framework comes with two predefined providers:

- **DPAPIProtectedConfigurationProvider** Uses the Windows Data Protection API (DPAPI) to encrypt and decrypt data
- **RSAProtectedConfigurationProvider** Default provider, uses the RSA encryption algorithm to encrypt and decrypt data

Being able to protect data stored in *web.config* is not a feature specific to connection strings. It applies, instead, to all sections, with very few exceptions. This said, let's see how to encrypt connection strings stored in the *web.config* file.

You can use the newest version of a popular system tool—*aspnet_regiis.exe*—or write your own tool by using the ASP.NET configuration API. If you use *aspnet_regiis*, examine the following code, which is a sample usage to encrypt connection strings for the Core35 application:

```
aspnet_regiis.exe -pe connectionStrings -app /core35
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Note that the section names are case-sensitive. Note also that connection strings are stored in a protected area that is completely transparent to applications, which continue working as before. If you open the *web.config* file after encryption, you see something like the following:

```
<configuration>
  <protectedData>
    <protectedDataSections>
      <add name="connectionStrings"
           provider="RSAProtectedConfigurationProvider" />
    </protectedDataSections>
  </protectedData>
  <connectionStrings>
    <EncryptedData …>
      ...
      <CipherData>
        <CipherValue>cQyofWFQ ... =</CipherValue>
      </CipherData>
    </EncryptedData>
  </connectionStrings>
</configuration>
```

To restore the *web.config* to its original clear state, you use the **–pd** switch in lieu of**–pe** in the aforementioned command line.

Caution Any page that uses protected sections works like a champ as long as you run it inside the local Web server embedded in Visual Studio. You might get an RSA provider configuration error if you access the same page from within a canonical (and much more realistic) IIS virtual folder. What's up with that?

The RSA-based provider—the default protection provider—needs a key container to work. A default key container is created upon installation and is named *NetFrameWorkConfigurationKey*. The *aspnet_regiis.exe* utility provides a lot of command-line switches for you to add, remove, and edit key containers. The essential point is that you have a key container created before you dump the RSA-protected configuration provider. The container must not only exist, but it also needs to be associated with the user account attempting to call it. The system account (running the local Web server) is listed with the container; the ASP.NET account on your Web server might not be. Assuming you run ASP.NET under the NETWORK SERVICE account (the default on Windows Server 2003 machines), you need the following code to add access to the container for the user:

```
aspnet_regiis.exe –pa "NetFrameworkConfigurationKey"
                   "NT AUTHORITY\NETWORK SERVICE"
```

It is important that you specify a complete account name, as in the preceding code. Note that granting access to the key container is necessary only if you use the RSA provider.

Both the RSA and DPAPI providers are great options for encrypting sensitive data. The DPAPI provider dramatically simplifies the process of key management—keys are generated based on machine credentials and can be accessed by all processes running on the machine. For the same reason, the DPAPI provider is not ideal to protect sections in a Web farm scenario where the same encrypted *web.config* file will be deployed to several servers. In this case,

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

either you manually encrypt all *web.config* files on each machine or you copy the same container key to all servers. To accomplish this, you create a key container for the application, export it to an XML file, and import it on each server that will need to decrypt the encrypted *web.config* file. To create a key container you do as follows:

```
aspnet_regiis.exe -pc YourContainerName -exp
```

Next, you export the key container to an XML file:

```
aspnet_regiis.exe -px YourContainerName YourXmlFile.xml
```

Next, you move the XML file to each server and import it as follows:

```
aspnet_regiis.exe -pi YourContainerName YourXmlFile.xml
```

As a final step, grant the ASP.NET account permission to access the container (again using *aspnet_regiis.exe* with the **–pa** option as just shown).

Note We won't cover the .NET Framework configuration API in this book. You can find deep coverage of the structure of configuration files and related APIs in my book, *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006).

Connection Pooling

Connection pooling is a fundamental aspect of high-performance, scalable applications. For local or intranet desktop applications that are not multithreaded, connection pooling is no big deal—you'll get nearly the same performance with and without pooling. Furthermore, using a nonpooled connection gives you more control over the lifetime. For multithreaded applications, the use of connection pooling is a necessity for performance reasons and to avoid nasty, hardware-dependent bugs. Finally, if ASP.NET applications are involved, every millisecond that the connection is idle steals valuable resources from other requests. Not only should you rely on connection pooling, but you should also open the connection as late as possible and close it as soon as you can.

Using connection pooling makes it far less expensive for the application to open and close the connection to the database, even if that is done frequently. (We'll cover more about this topic later.) All standard .NET data providers have pooling support turned on by default. The .NET data providers for SQL Server and Oracle manage connection pooling internally using ad hoc classes. For the OLE DB data provider, connection pooling is implemented through the OLE DB service infrastructure for session pooling. Connection-string arguments (for example, *OLE DB Service*) can be used to enable or disable various OLE DB services, including pooling. A similar situation occurs with ODBC, in which pooling is controlled by the ODBC driver manager.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Configuring Pooling

Some settings in the connection string directly affect the pooling mechanism. The parameters you can control to configure the SQL Server environment are listed in Table 7-9.

TABLE 7-9 SQL Server Connection Pooling Keywords

Keyword	Description
Connection Lifetime	Sets the maximum duration in seconds of the connection object in the pool. This keyword is checked only when the connection is returned to the pool. If the time the connection has been open is greater than the specified lifetime, the connection object is destroyed. (We'll cover more about this topic later.)
Connection Reset	Determines whether the database connection is reset when being drawn from the pool. Default is *true*.
Enlist	Indicates that the pooler automatically enlists the connection in the creation thread's current transaction context. Default is *true*.
Max Pool Size	Maximum number of connections allowed in the pool. Default is 100.
Min Pool Size	Minimum number of connections allowed in the pool. Default is 0.
Pooling	Indicates that the connection object is drawn from the appropriate pool or, if necessary, is created and added to the appropriate pool. Default is *true*.

With the exception of *Connection Reset*, all the keywords listed in Table 7-9 are acceptable to the Oracle managed provider too.

As far as SQL Server and Oracle providers are concerned, connection pooling is automatically enabled; to disable it, you need to set *Pooling* to *false* in the connection string. To control pooling for an ODBC data source, you use the ODBC Data Source Administrator in the Control Panel. The Connection Pooling tab allows you to specify connection pooling parameters for each ODBC driver installed. Note that any changes to a specific driver affect all applications that make use of it. The .NET data provider for OLE DB automatically pools connections using OLE DB session pooling. You can disable pooling by setting the *OLE DB Services* keyword to -4.

In ADO.NET 2.0 and newer versions, auto enlistment (the *Enlist* keyword) works in the connection strings of all standard data providers, including providers for OLE DB and ODBC. In ADO.NET 1.x, only managed providers for SQL Server and Oracle support auto-enlistment because they are made of native managed code instead of being wrappers around existing code. The new *EnlistTransaction* method on connection classes allows you to enlist a connection object programmatically, be it pooled or not.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Getting and Releasing Objects

Each connection pool is associated with a distinct connection string and the transaction context. When a new connection is opened, if the connection string does not exactly match an existing pool, a new pool is created. Once created, connection pools are not destroyed until the process ends. This behavior does not affect the system performance because maintenance of inactive or empty pools requires only minimal overhead.

When a pool is created, multiple connection objects are created and added so that the minimum size is reached. Next, connections are added to the pool on demand, up to the maximum pool size. Adding a brand new connection object to the pool is the really expensive operation here, as it requires a roundtrip to the database. Next, when a connection object is requested, it is drawn from the pool as long as a usable connection is available. A usable connection must currently be unused, have a matching or null transaction context, and have a valid link to the server. If no usable connection is available, the pooler attempts to create a new connection object. When the maximum pool size is reached, the request is queued and served as soon as an existing connection object is released to the pool.

Connections are released when you call methods such as *Close* or *Dispose*. Connections that are not explicitly closed might not be returned to the pool unless the maximum pool size has been reached and the connection is still valid.

A connection object is removed from the pool if the lifetime has expired (which will be explained further in a moment) or if a severe error has occurred. In these cases, the connection is marked as invalid. The pooler periodically scavenges the various pools and permanently removes invalid connection objects.

Important Connection pools in ADO.NET are created based on the connection string applying an exact match algorithm. In other words, to avoid the creation of an additional connection pool you must ensure that two connection strings carrying the same set of parameters are expressed by two byte-per-byte identical strings. A different order of keywords, or blanks interspersed in the text, are not ignored and end up creating additional pools and therefore additional overhead.

To make connection pooling work effectively, it is extremely important that connection objects are returned to the pool as soon as possible. It is even more important, though, that connections are returned. Note that a connection object that goes out of scope is not closed and, therefore, not immediately returned. For this reason, it is highly recommended that you work with connection objects according to the following pattern:

```
SqlConnection conn = new SqlConnection(connString);
try {
    conn.Open();
    // Do something here
}
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
catch {
    // Trap errors here
}
finally {
    conn.Close();
}
```

Alternately, you can resort to the C# *using* statement, as follows:

```
using (SqlConnection conn = new SqlConnection(connString))
{
    // Do something here
    // Trap errors here
}
```

The *using* statement is equivalent to the preceding *try/catch/finally* block in which *Close* or *Dispose* is invoked in the *finally* block. You can call either *Close* or *Dispose* or even both—they do the same thing. *Dispose* cleans the connection string information and then calls *Close*. In addition, note that calling each one multiple times doesn't result in runtime troubles, as closing or disposing of an already closed or disposed of connection is actually a no-operation.

Note Before the .NET Framework 2.0, there was no sort of *using* statement in Visual Basic .NET. Now, you can rely on a shortcut keyword for *try/catch/finally* blocks also in Visual Basic .NET. The keyword is *Using ... End Using*.

```
Using conn As New SqlConnection()
   ...
End Using
```

Detecting Connections Leaks

In ADO.NET 2.0 and newer, you can easily figure out whether you're leaking connections thanks to some new performance counters. In particular, you can monitor the *NumberOfReclaimedConnections* counter, and if you see it going up, you have the evidence that your application is making poor use of connection objects. A good symptom of connection leaking is when you get an invalid operation exception that claims the timeout period elapsed prior to obtaining a connection from the pool. You can make this exception disappear or, more exactly, become less frequent by tweaking some parameters in the connection string. Needless to say, this solution doesn't remove the leak; it simply changes runtime conditions to make it happen less frequently. Here's a quick list of things you should not do that relate to connection management:

- **Do not turn connection pooling off** With pooling disabled, a new connection object is created every time. No timeout can ever occur, but you lose a lot in performance and, more importantly, you are still leaking connections.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

- **Do not shrink the connection lifetime** Reducing the lifetime of the connection will force the pooler to renew connection objects more frequently. A short lifetime (a few seconds) will make the timeout exception extremely unlikely, but it adds significant overhead and doesn't solve the real problem. Let's say that it is only a little better than turning pooling off.
- **Do not increase the connection timeout** You tell the pooler to wait a longer time before throwing the timeout exception. Whatever value you set here, ASP.NET aborts the thread after 3 minutes. In general, this option worsens performance without alleviating the problem.
- **Do not increase the pool size** If you set the maximum pool size high enough (how high depends on the context), you stop getting timeout exceptions while keeping pooling enabled. The drawback is that you greatly reduce your application's scalability because you force your application to use a much larger number of connections than is actually needed.

To avoid leaking connections, you need to guarantee *only* that the connection is closed or disposed of when you're done, and preferably soon after you're done with it.

In the previous section, I emphasized the importance of writing code that guarantees the connection is always closed. However, there might be nasty cases in which your code places a call to *Close*, but it doesn't get called. Let's see why. Consider the following code:

```
SqlConnection conn = new SqlConnection(connString);
conn.Open();
SqlCommand cmd = new SqlCommand(cmdText, conn);
cmd.ExecuteNonQuery();
conn.Close();
```

What if the command throws an exception? The *Close* method is not called, and the connection is not returned to the pool. Wrapping the code in a *using* statement would do the trick because it ensures that *Dispose* is always invoked on the object being used. Here's the correct version of the code:

```
using (SqlConnection conn = new SqlConnection(connString))
{
    conn.Open();
    SqlCommand cmd = new SqlCommand(cmdText, conn);
    cmd.ExecuteNonQuery();
    conn.Close();  // Not called in case of exception
}  // Dispose always called
```

Wrapping your connection in such protected code sections is the only way to avoid connection leaking. (Note that *try/catch/finally* can also be used if you're interested in dealing with any exceptions yourself as close to the source of the problem, if that involves opening the connection or making the database query.)

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Managing Connection Lifetime

The *Connection Lifetime* keyword indicates in seconds the time a connection object is considered valid. When the time has elapsed, the connection object should be disposed of. But why on earth should you get rid of a perfectly good connection object? This keyword is useful only in a well-known situation, and it should never be used otherwise. Imagine you have a cluster of servers sharing the workload. At some point, you realize the load is too high and you turn on an additional server. With good reason, you expect the workload to be distributed among all servers. However, this might not happen—the newly added server is idle.

A plausible and common reason for this is that middle-tier components cache the connections and never open new ones. By disposing of working connections, you force the middle-tier applications to create new connections. Needless to say, new connections will be assigned to the least loaded server. In the end, you should set *Connection Lifetime* only if you're in a cluster scenario. Finally, note that in ADO.NET 2.0 the connection builder classes use a different (and more intuitive) name to address the keyword—*LoadBalanceTimeout*.

Note *LoadBalanceTimeout* is not a newly supported attribute for a connection string. If you use the *SqlConnectionStringBuilder* class to programmatically build the connection string, you'll find a *LoadBalanceTimeout* property to set the *Connection Lifetime* attribute.

Clearing the Connection Pool

Until ADO.NET 2.0, there was no way to programmatically clear the pool of open connections. Admittedly, this is not an operation you need to perform often, but it becomes essential in case the database server goes down for whatever reason. Consider the following scenario: your ASP.NET pages open and then successfully close some connections out of the same pool. Next, the server suddenly goes down and is restarted. As a result, all connection objects in the pool are now invalid because each of them holds a reference to a server connection that no longer exists. What happens when a new page request is issued?

The answer is that the pooler returns an apparently valid connection object to the page, and the page runs the command. Unfortunately, the connection object is not recognized by the database server, resulting in an exception. The connection object is removed from the pool and replaced. The exception will be raised for each command as long as there are connection objects in the pool. In summary, shutting down the server without shutting down the application brings the connection pool into an inconsistent, corrupted state.

This situation is common for applications that deal with server reboots, like a failover cluster. Only one solution is possible—flushing the connection pool. It is not as easy to implement as it might seem at first, though. An easier workaround is catching the exception and changing the connection string slightly to force the use of a new connection pool.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

ADO.NET is smart enough to recognize when an exception means that the pool is corrupted. When an exception is thrown during the execution of a command, ADO.NET realizes whether the exception means that the pool is corrupted. In this case, it walks down the pool and marks each connection as obsolete. When does an exception indicate pool corruption? It has to be a fatal exception raised from the network layer on a previously opened connection. All other exceptions are ignored and bubble up as usual.

Two new static methods—*ClearPool* and *ClearAllPools* defined for both *SqlConnection* and *OracleConnection*—can be used to programmatically clear the pool, if you know that the server has been stopped and restarted. These methods are used internally by ADO.NET to clear the pool as described earlier.

Executing Commands

Once you have a physical channel set up between your client and the database, you can start preparing and executing commands. The ADO.NET object model provides two types of command objects—the traditional one-off command and the data adapter. The one-off command executes a SQL command or a stored procedure and returns a sort of cursor. Using that, you then scroll through the rows and read data. While the cursor is in use, the connection is busy and open. The data adapter, on the other hand, is a more powerful object that internally uses a command and a cursor. It retrieves and loads the data into a data container class—*DataSet* or *DataTable*. The client application can then process the data while disconnected from the source.

We'll cover container classes and data adapters in the next chapter. Let's focus on one-off commands, paying particular attention to SQL Server commands.

The *SqlCommand* Class

The *SqlCommand* class represents a SQL Server statement or stored procedure. It is a cloneable and sealed class that implements the *IDbCommand* interface. In ADO.NET, it derives from *DbCommand* which, in turn, implements the interface. A command executes in the context of a connection and, optionally, a transaction. This situation is reflected by the constructors available in the *SqlCommand* class:

```
public SqlCommand();
public SqlCommand(string);
public SqlCommand(string, SqlConnection);
public SqlCommand(string, SqlConnection, SqlTransaction);
```

The string argument denotes the text of the command to execute (and it can be a stored procedure name), whereas the *SqlConnection* parameter is the connection object to use. Finally, if specified, the *SqlTransaction* parameter represents the transactional context

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

in which the command has to run. ADO.NET command objects never implicitly open a connection. The connection must be explicitly assigned to the command by the programmer and opened and closed with direct operations. The same holds true for the transaction.

Properties of the *SqlCommand* Class

Table 7-10 shows the attributes that make up a command in the.NET data provider for SQL Server.

TABLE 7-10 Properties of the *SqlCommand* Class

Property	*IDbCommand* Interface Property?	Description
CommandText	Yes	Gets or sets the statement or the stored procedure name to execute.
CommandTimeout	Yes	Gets or sets the seconds to wait while trying to execute the command. The default is 30.
CommandType	Yes	Gets or sets how the *CommandText* property is to be interpreted. Set to *Text* by default, which means the *CommandText* property contains the text of the command.
Connection	Yes	Gets or sets the connection object used by the command. It is null by default.
Notification	No	Gets or sets the *SqlNotificationRequest* object bound to the command. *This property requires SQL Server 2005.*
NotificationAutoEnlist	No	Indicates whether the command will automatically enlist the SQL Server 2005 notification service. *This property requires SQL Server 2005.*
Parameters	Yes	Gets the collection of parameters associated with the command.
Transaction	Yes	Gets or sets the transaction within which the command executes. The transaction must be connected to the same connection as the command.
UpdatedRowSource	Yes	Gets or sets how query command results are applied to the row being updated. The value of this property is used only when the command runs within the *Update* method of the data adapter. Acceptable values are in the *UpdateRowSource* enumeration.

Commands can be associated with parameters, and each parameter is rendered using a provider-specific object. For the SQL Server managed provider, the parameter class is

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

SqlParameter. The command type determines the role of the *CommandText* property. The possible values for *CommandType* are as follows:

- **Text** The default setting, which indicates the property contains Transact-SQL text to execute directly.
- **StoredProcedure** Indicates that the content of the property is intended to be the name of a stored procedure contained in the current database.
- **TableDirect** Indicates the property contains a comma-separated list containing the names of the tables to access. All rows and columns of the tables will be returned. It is supported only by the data provider for OLE DB.

To execute a stored procedure, you need the following:

```
using (SqlConnection conn = new SqlConnection(ConnString))
{
    SqlCommand cmd = new SqlCommand(sprocName, conn);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Connection.Open();
    cmd.ExecuteNonQuery();
}
```

In ADO.NET 2.0, commands have two main new features—asynchronous executors and support for notification services. We'll cover both later.

Methods of the *SqlCommand* Class

Table 7-11 details the methods available for the *CommandText* class.

TABLE 7-11 Methods of the *CommandText* Class

Property	*IDbCommand* Interface Method?	Description
BeginExecuteNonQuery	No	Executes a nonquery command in a nonblocking manner. *Not supported in ADO.NET 1.x.*
BeginExecuteReader	No	Executes a query command in a nonblocking manner. *Not supported in ADO.NET 1.x.*
BeginExecuteXmlReader	No	Executes an XML query command in a nonblocking manner. *Not supported in ADO.NET 1.x.*
Cancel	Yes	Attempts to cancel the execution of the command. No exception is generated if the attempt fails.
CreateParameter	Yes	Creates a new instance of a *SqlParameter* object.
EndExecuteNonQuery	No	Completes a nonquery command executed asynchronously. *Not supported in ADO.NET 1.x.*

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Property	*IDbCommand* Interface Method?	Description
EndExecuteReader	No	Completes a query command executed asynchronously. *Not supported in ADO.NET 1.x.*
EndExecuteXmlReader	No	Completes an XML query command executed asynchronously. *Not supported in ADO.NET 1.x.*
ExecuteNonQuery	Yes	Executes a nonquery command, and returns the number of rows affected.
ExecuteReader	Yes	Executes a query, and returns a forward-only, read-only cursor—the data reader—to the data.
ExecuteScalar	Yes	Executes a query, and returns the value in the 0,0 position (first column of first row) in the result set. Extra data is ignored.
ExecuteXmlReader	No	Executes a query that returns XML data and builds an *XmlReader* object.
Prepare	Yes	Creates a prepared version of the command in an instance of SQL Server.
ResetCommandTimeout	No	Resets the command timeout to the default.

Parameterized commands define their own arguments using instances of the *SqlParameter* class. Parameters have a name, value, type, direction, and size. In some cases, parameters can also be associated with a source column. A parameter is associated with a command by using the *Parameters* collection:

```
SqlParameter parm = new SqlParameter();
parm.ParameterName = "@employeeid";
parm.DbType = DbType.Int32;
parm.Direction = ParameterDirection.Input;
cmd.Parameters.Add(parm);
```

The following SQL statement uses a parameter:

```
SELECT * FROM employees WHERE employeeid=@employeeid
```

The .NET data provider for SQL Server identifies parameters by name, using the @ symbol to prefix them. In this way, the order in which parameters are associated with the command is not critical.

Note Named parameters are supported by the managed provider for Oracle but not by the providers for OLE DB and ODBC data sources. The OLE DB and ODBC data sources use positional parameters identified with the question mark (?) placeholder. The order of parameters for these data sources is important.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Ways to Execute

As Table 7-11 shows, a *SqlCommand* object can be executed either synchronously or asynchronously. Let's focus on synchronous execution, which is supported on all .NET platforms. Execution can happen in four different ways: *ExecuteNonQuery*, *ExecuteReader*, *ExecuteScalar*, and *ExecuteXmlReader*. The various executors work in much the same way, but they differ in the return values. Typically, you use the *ExecuteNonQuery* method to perform update operations such as those associated with UPDATE, INSERT, and DELETE statements. In these cases, the return value is the number of rows affected by the command. For other types of statements, such as SET or CREATE, the return value is -1.

The *ExecuteReader* method is expected to work with query commands, and it returns a data reader object—an instance of the *SqlDataReader* class. The data reader is a sort of read-only, forward-only cursor that client code scrolls and reads from. If you execute an update statement through *ExecuteReader*, the command is successfully executed but no affected rows are returned. We'll return to data readers in a moment.

The *ExecuteScalar* method helps considerably when you have to retrieve a single value. It works great with SELECT COUNT statements or for commands that retrieve aggregate values. If you call the method on a regular query statement, only the value in the first column of the first row is read and all the rest is discarded. Using *ExecuteScalar* results in more compact code than you'd get by executing the command and manually retrieving the value in the top-left corner of the rowset.

These three executor methods are common to all command objects. The *SqlCommand* class also features the *ExecuteXmlReader* method. It executes a command that returns XML data and builds an XML reader so that the client application can easily navigate through the XML tree. The *ExecuteXmlReader* method is ideal to use with query commands that end with the FOR XML clause or with commands that query for text fields filled with XML data. Note that while the *XmlReader* object is in use, the underlying connection is busy.

ADO.NET Data Readers

The data reader class is specific to a DBMS and works like a firehose-style cursor. It allows you to scroll through and read one or more result sets generated by a command. The data reader operates in a connected way and moves in a forward-only direction. A data reader is instantiated during the execution of the *ExecuteReader* method. The results are stored in a buffer located on the client and are made available to the reader.

By using the data reader object, you access data one record at a time as soon as it becomes available. An approach based on the data reader is effective both in terms of system overhead and performance. Only one record is cached at any time, and there's no wait time to have the entire result set loaded in memory.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Table 7-12 shows the properties of the *SqlDataReader* class—that is, the data reader class for SQL Server.

TABLE 7-12 Properties of the *SqlDataReader* Class

Property	Description
Depth	Indicates the depth of nesting for the current row. For the *SqlDataReader* class, it always returns 0.
FieldCount	Gets the number of columns in the current row.
HasRows	Gets a value that indicates whether the data reader contains one or more rows. *Not supported in ADO.NET 1.0.*
IsClosed	Gets a value that indicates whether the data reader is closed.
Item	Indexer property, gets the value of a column in the original format.
RecordsAffected	Gets the number of rows modified by the execution of a batch command.

The *Depth* property is meant to indicate the level of nesting for the current row. The depth of the outermost table is always 0; the depth of inner tables grow by one. Most data readers, including the *SqlDataReader* and *OracleDataReader* classes, do not support multiple levels of nesting so that the *Depth* property always returns 0.

The *RecordsAffected* property is not set until all rows are read and the data reader is closed. The default value of *RecordsAffected* is -1. Note that *IsClosed* and *RecordsAffected* are the only properties you can invoke on a closed data reader.

Table 7-13 lists the methods of the SQL Server data reader class.

TABLE 7-13 Methods of the *SqlDataReader* Class

Methods	Description
Close	Closes the reader object. Note that closing the reader does not automatically close the underlying connection.
GetBoolean	Gets the value of the specified column as a Boolean.
GetByte	Gets the value of the specified column as a byte.
GetBytes	Reads a stream of bytes from the specified column into a buffer. You can specify an offset both for reading and writing.
GetChar	Gets the value of the specified column as a single character.
GetChars	Reads a stream of characters from the specified column into a buffer. You can specify an offset both for reading and writing.
GetDataTypeName	Gets the name of the back-end data type in the specified column.
GetDateTime	Gets the value of the specified column as a *DateTime* object.
GetDecimal	Gets the value of the specified column as a decimal.
GetDouble	Gets the value of the specified column as a double-precision floating-point number.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Methods	Description
GetFieldType	Gets the *Type* object for the data in the specified column.
GetFloat	Gets the value of the specified column as a single-precision floating-point number.
GetGuid	Gets the value of the specified column as a globally unique identifier (GUID).
GetInt16	Gets the value of the specified column as a 16-bit integer.
GetInt32	Gets the value of the specified column as a 32-bit integer.
GetInt64	Gets the value of the specified column as a 64-bit integer.
GetName	Gets the name of the specified column.
GetOrdinal	Given the name of the column, returns its ordinal number.
GetSchemaTable	Returns a *DataTable* object that describes the metadata for the columns managed by the reader.
GetString	Gets the value of the specified column as a string.
GetValue	Gets the value of the specified column in its original format.
GetValues	Copies the values of all columns in the supplied array of objects.
IsDbNull	Indicates whether the column contains null values. The type for a null column is *System.DBNull*.
NextResult	Moves the data reader pointer to the beginning of the next result set, if any.
Read	Moves the data reader pointer to the next record, if any.

The SQL Server data reader also features a variety of other DBMS-specific *get* methods. They include methods such as *GetSqlDouble*, *GetSqlMoney*, *GetSqlDecimal*, and so on. The difference between the *GetXXX* and *GetSqlXXX* methods is in the return type. With the *GetXXX* methods, a base .NET Framework type is returned; with the *GetSqlXXX* methods, a .NET Framework wrapper for a SQL Server type is returned—such as *SqlDouble*, *SqlMoney*, or *SqlDecimal*. The SQL Server types belong to the *SqlDbType* enumeration.

All the *GetXXX* methods that return a value from a column identify the column through a 0-based index. Note that the methods don't even attempt a conversion; they simply return data as is and just make a cast to the specified type. If the actual value and the type are not compatible, an exception is thrown.

Note The *GetBytes* method is useful to read large fields one step at a time. However, the method can also be used to obtain the length in bytes of the data in the column. To get this information, pass a buffer that is a null reference and the return value of the method will contain the length.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Reading Data with the Data Reader

The key thing to remember when using a data reader is that you're working while connected. The data reader represents the fastest way to read data out of a source, but you should read your data as soon as possible and release the connection. One row is available at a time, and you must move through the result set by using the *Read* method. The following code snippet illustrates the typical loop you implement to read all the records of a query:

```
using (SqlConnection conn = new SqlConnection(connString))
{
    string cmdText = "SELECT * FROM customers";
    SqlCommand cmd = new SqlCommand(cmdText, conn);
    cmd.Connection.Open();
    SqlDataReader reader = cmd.ExecuteReader();
    while (reader.Read())
        CustomerList.Items.Add(reader["companyname"].ToString());
    reader.Close();
}
```

You have no need to explicitly move the pointer ahead and no need to check for the end of the file. The *Read* method returns *false* if there are no more records to read. A data reader is great if you need to consume data by processing the records in some way. If you need to cache values for later use, the data reader is not appropriate. You need a container object in this case, as we'll see in Chapter 8.

Note Although accessing row fields by name is easy to read and understand, it is not the fastest approach. Internally, in fact, the data reader needs to resolve the name to a 0-based index. If you provide the index directly, you get slightly faster code.

```
const int Customers_CustomerID = 0;
...
Response.Write(reader[Customers_CustomerID].ToString());
```

The preceding code shows that using constants turns out to be a good compromise between speed and readability.

Command Behaviors

When calling the *ExecuteReader* method on a command object—on any command object regardless of the underlying DBMS—you can require a particular working mode known as a command behavior. *ExecuteReader* has a second overload that takes an argument of type *CommandBehavior*:

```
cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

CommandBehavior is an enumeration. Its values are listed in Table 7-14.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

TABLE 7-14 Command Behaviors for the Data Reader

Behavior	Description
CloseConnection	Automatically closes the connection when the data reader is closed.
Default	No special behavior is required. Setting this option is functionally equivalent to calling *ExecuteReader* without parameters.
KeyInfo	The query returns only column metadata and primary key information. The query is executed without any locking on the selected rows.
SchemaOnly	The query returns only column metadata and does not put any lock on the database rows.
SequentialAccess	Enables the reader to load data as a sequential stream. This behavior works in conjunction with methods such as *GetBytes* and *GetChars*, which can be used to read bytes or characters having a limited buffer size for the data being returned.
SingleResult	The query is expected to return only the first result set.
SingleRow	The query is expected to return a single row.

The sequential access mode applies to all columns in the returned result set. This means you can access columns only in the order in which they appear in the result set. For example, you cannot read column #2 before column #1. More exactly, if you read or move past a given location, you can no longer read or move back. Combined with the *GetBytes* method, sequential access can be helpful in cases in which you must read binary large objects (BLOB) with a limited buffer.

Note You can specify *SingleRow* also when executing queries that are expected to return multiple result sets. In this case, all the generated result sets are correctly returned, but each result set has a single row. *SingleRow* and *SingleResult* serve the purpose of letting the underlying provider machinery know about the expected results so that some internal optimization can optionally be made.

Closing the Reader

The data reader is not a publicly creatable object. It does have a constructor, but not one that is callable from within user applications. The data reader constructor is marked as internal and can be invoked only from classes defined in the same assembly—*System.Data*. The data reader is implicitly instantiated when the *ExecuteReader* method is called. Opening and closing the reader are operations distinct from instantiation and must be explicitly invoked by the application. The *Read* method advances the internal pointer to the next readable record in the current result set. The *Read* method returns a Boolean value indicating whether or not more records can be read. While records are being read, the connection is busy and no operation, other than closing, can be performed on the connection object.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The data reader and the connection are distinct objects and should be managed and closed independently. Both objects provide a *Close* method that should be called twice—once on the data reader (first) and once on the connection. When the *CloseConnection* behavior is required, closing the data reader also closes the underlying connection. In addition, the data reader's *Close* method fills in the values for any command output parameters and sets the *RecordsAffected* property.

Tip Because of the extra work *Close* always performs on a data reader class, closing a reader with success can sometimes be expensive, especially in cases of long-running and complicated queries. In situations in which you need to squeeze out every bit of performance, and where the return values and number of records affected are not significant, you can invoke the *Cancel* method of the associated *SqlCommand* object instead of closing the reader. *Cancel* aborts the operation and closes the reader faster. Aside from this, you're still responsible for properly closing the underlying connection.

Accessing Multiple Result Sets

Depending on the syntax of the query, multiple result sets can be returned. By default, the data reader is positioned on the first of them. You use the *Read* method to scroll through the various records in the current result set. When the last record is found, the *Read* method returns *false* and does not advance further. To move to the next result set, you should use the *NextResult* method. The method returns *false* if there are no more result sets to read. The following code shows how to access all records in all returned result sets:

```
using (SqlConnection conn = new SqlConnection(connString))
{
    string cmdText = Query.Text;
    SqlCommand cmd = new SqlCommand(cmdText, conn);
    cmd.Connection.Open();
    SqlDataReader reader = cmd.ExecuteReader();

    do {
        // Move through the first result set
        while (reader.Read())
            sb.AppendFormat("{0}, {1}<br/>", reader[0], reader[1]);

        // Separate result sets
        sb.Append("<hr />");
    } while (reader.NextResult());

    reader.Close();
}

// Display results in the page
Results.Text = sb.ToString();
```

Figure 7-7 shows the output generated by the sample page based on this code.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

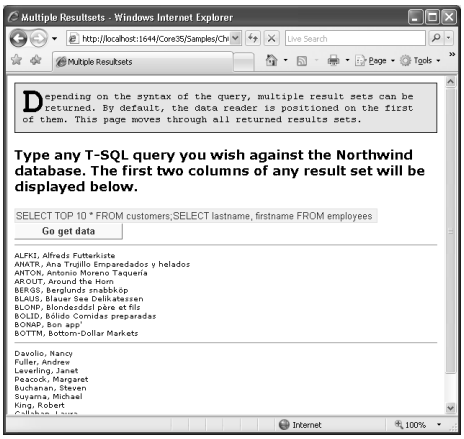

FIGURE 7-7 Processing multiple result sets.

Note The .NET Framework version 1.1 extends the programming interface of data readers by adding the *HasRows* method, which returns a Boolean value indicating whether or not there are more rows to read. However, the method does not tell anything about the number of rows available. Similarly, there is no method or trick to know in advance how many result sets have been returned.

Asynchronous Commands

A database operation is normally a synchronous operation—the caller regains control of the application only after the interaction with the database is completed. This approach can lead to performance and scalability issues in lengthy operations—a common scenario when you interact with a DBMS. The .NET Framework 1.x supports asynchronous operations, but the model is implemented around user-level code. In other words, you can implement your own procedures asynchronously and connect to databases and run commands as part of the code, but connection management and command execution remain atomic operations that execute synchronously.

The .NET data provider for SQL Server in ADO.NET 2.0 (and beyond) provides true asynchronous support for executing commands. This offers a performance advantage because you can perform other actions until the command completes. However, this is not the only benefit. The support for asynchronous operations is built into the *SqlCommand* class and is limited to executing nonquery commands and getting a reader or an XML reader. You can use three

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

different approaches to build commands that work asynchronously. They are nonblocking, polling, and callback.

Setting Up Asynchronous Commands

To enable asynchronous commands, you must set the new *Async* attribute to *true* in the connection string. You'll receive an exception if any of the asynchronous methods are called over a connection that doesn't have asynchronous capabilities explicitly turned on. Enabling asynchronous commands does have a cost in terms of overall performance; for this reason, you're better off using the *Async* keyword only with connection objects that execute asynchronous operations only.

If you need both synchronous and asynchronous commands, employ different connections wherever possible. Note, though, that you can still call synchronous methods over connections enabled to support asynchronous operations. However, you'll only end up using more resources than needed and experience a performance degradation.

Note Asynchronous commands are not implemented by creating a new thread and blocking execution on it. Among other things, ADO.NET is not thread-safe and blocking threads would be a serious performance hit. When asynchronous commands are enabled, ADO.NET opens the TCP socket to the database in overlapped mode and binds it to an I/O completion port. In light of this, synchronous operations execute as the emulation of asynchronous operations, and this explains why they're more expensive than asynchronous-enabled connections.

Nonblocking Commands

Nonblocking commands are the simplest case of asynchronous commands. The code starts the operation and continues executing other unrelated methods; then it comes back to get the results. Whatever the model of choice happens to be, the first step of an asynchronous command is calling one of the *BeginExecuteXXX* methods. For example, if you want to execute a reading command, you call *BeginExecuteReader*:

```
// Start a non-blocking execution
IAsyncResult iar = cmd.BeginExecuteReader();

// Do something else meanwhile
...

// Block the execution until done
SqlDataReader reader = cmd.EndExecuteReader(iar);

// Process data here ...
ProcessData(reader);
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The *BeginExecuteReader* function returns an *IAsyncResult* object you will use later to complete the call. Note that *EndExecuteReader* is called to finish the operation and will block execution until the ongoing command terminates. The *EndExecuteReader* function will automatically sync up the command with the rest of the application, blocking the code whenever the results of the command are not ready.

As an alternative to the aforementioned approach, the client code might want to check the status of a running asynchronous operation and poll for completion. The following code illustrates the polling option with a query statement:

```
// Executes a query statement
IAsyncResult iar = cmd.BeginExecuteReader();
do {
    // Do something here
} while (!iar.IsCompleted);

// Sync up
SqlDataReader reader = cmd.EndExecuteReader(iar);
ProcessData(reader);
```

It is important to note that if *iar.IsCompleted* returns *true*, the *EndExecuteReader* method will not block the application.

The third option for nonblocking commands has the client code start the database operation and continue without waiting. Later on, when the operation is done, it receives a call. In this case, you pass a delegate to a *BeginExecuteXXX* method and any information that constitutes the state of the particular call. The state is any information you want to pass to the callback function. In this case, you pass the command object:

```
// Begin executing the command
IAsyncResult ar = cmd.BeginExecuteReader(
    new AsyncCallback(ProcessData), cmd);
```

After initiating the asynchronous operation, you can forget about it and do any other work. The specified callback function is invoked at the end of the operation. The callback must have the following layout:

```
public void ProcessData(IAsyncResult ar)
{
    // Retrieve the context of the call
    SqlCommand cmd = (SqlCommand) iar.AsyncState;

    // Complete the async operation
    SqlDataReader reader = cmd.EndExecuteReader(iar);
    ...
}
```

The context of the call you specified as the second argument to *BeingExecuteReader* is packed in the *AsyncState* property of the *IAsyncResult* object.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Note The callback will be called in a thread-pool thread, which is likely to be different from the thread that initiated the operation. Proper thread synchronization might be needed, depending on the application. This also poses a problem with the user interface of applications, especially Windows Forms applications. Ensuring that the UI is refreshed in the right thread is up to you. Windows Forms controls and forms provide mechanisms for deciding if the correct thread is currently executing and for accessing the correct thread if it isn't. You should consult the MSDN documentation or a good Windows Forms programming book for more information regarding multithreaded Windows Forms programming. Note that if you fail to use the threading model correctly, your application will almost certainly lock up and quite possibly even crash.

Executing Parallel Commands in an ASP.NET Page

Having asynchronous commands available is not necessarily a good reason for using them without due forethought. Let's examine a couple of scenarios where asynchronous commands are useful for building better Web pages. The first scenario we'll consider is the execution of multiple SQL statements in parallel, either against the same or different database servers.

Imagine your page displays information about a particular customer—both personal and accounting data. The former block of data comes from the client's database; the latter is excerpted from the accounting database. You can fire both queries at the same time and have them execute in parallel on distinct machines—thus benefitting from true parallelism. Here's an example:

```
protected void QueryButton_Click(object sender, EventArgs e)
{
  string custID = CustomerList.SelectedValue;

  using (SqlConnection conn1 = new SqlConnection(ConnString1))
  using (SqlConnection conn2 = new SqlConnection(ConnString2))
  {
    // Fire the first command: get customer info
    SqlCommand cmd1 = new SqlCommand(CustomerInfoCmd, conn1);
    cmd1.Parameters.Add("@customerid", SqlDbType.Char, 5).Value = custID;
    conn1.Open();
    IAsyncResult arCustomerInfo = cmd1.BeginExecuteReader();

    // Fire the second command: get order info
    SqlCommand cmd2 = new SqlCommand(CustomerOrderHistory, conn2);
    cmd2.CommandType = CommandType.StoredProcedure;
    cmd2.Parameters.Add("@customerid", SqlDbType.Char, 5).Value = custID;
    conn2.Open();
    IAsyncResult arOrdersInfo = cmd2.BeginExecuteReader();

    // Prepare wait objects to sync up
    WaitHandle[] handles = new WaitHandle[2];
    handles[0] = arCustomerInfo.AsyncWaitHandle;
    handles[1] = arOrdersInfo.AsyncWaitHandle;
    SqlDataReader reader;
```


Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    // Wait for all commands to terminate (no longer than 5 secs)
    for (int i=0; i<2; i++)
    {
      StringBuilder builder = new StringBuilder();
      int index = WaitHandle.WaitAny(handles, 5000, false);
          if (index == WaitHandle.WaitTimeout)
                throw new Exception("Timeout expired");

      if (index == 0) {    // Customer info
        reader = cmd1.EndExecuteReader(arCustomerInfo);
        if (!reader.Read())
          continue;

        builder.AppendFormat("{0}<br>", reader["companyname"]);
        builder.AppendFormat("{0}<br>", reader["address"]);
        builder.AppendFormat("{0}<br>", reader["country"]);
        Info.Text = builder.ToString();
        reader.Close();
      }
      if (index == 1) {    // Orders info
        reader = cmd2.EndExecuteReader(arOrdersInfo);
        gridOrders.DataSource = reader;
        gridOrders.DataBind();
        reader.Close();
      }
    }
  }
}
```

The page fires the two commands and then sits waiting for the first command to terminate. The *AsyncWaitHandle* object of each *IAsyncResult* is stored in an array and passed to the *WaitAny* method of the *WaitHandle* class. *WaitAny* signals out when any of the commands terminates, but the surrounding *for* statement reiterates the wait until all pending commands terminate. You could have more easily opted for the *WaitAll* method. In this case, though, you can process results as they become available. This fact ensures a performance gain, especially for long-running stored procedures.

Note You can implement the same behavior in ADO.NET 1.x without asynchronous commands by simply assigning each command to a different thread—either a user-defined one or one from the thread pool. In this case, though, each command would have blocked a thread. Blocking threads is fine for client-side applications, but it might compromise scalability in server-side applications such as ASP.NET applications.

Nonblocking Data-Driven ASP.NET Pages

Imagine a data-driven ASP.NET page that employs long-running, synchronous commands. The more the page is requested, the more likely it is that a large share of system threads are blocked while waiting for the database to return results. The paradoxical effect of this is that

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

the Web server is virtually idle (with almost no CPU and network usage) but can't accept new requests because it has very few threads available.

To address this problem, since version 1.0 ASP.NET supports asynchronous HTTP handlers—that is, a special breed of page classes that implement the *IHttpAsyncHandler* interface instead of *IHttpHandler*. Asynchronous HTTP handlers take care of a request and produce a response in an asynchronous manner. In the .NET Framework 2.0, asynchronous handlers can combine with asynchronous commands to boost data-driven pages.

The *IHttpAsyncHandler* interface has *BeginProcessRequest* and *EndProcessRequest* methods. In the former method, you connect to the database and kick off the query. *BeginProcessRequest* receives a callback function directly from ASP.NET; the same callback is used to detect the completion of the asynchronous command.

When *BeginProcessRequest* returns, the page gives the control back to ASP.NET as if it was served. ASP.NET is now free to reuse the thread to process another request while the database server proceeds. When the query is complete, the signaling mechanism ends up invoking the *EndProcessRequest* method, although not necessarily on the same thread as the rest of the page, so to speak. The *EndProcessRequest* method is where you simply collect the data and render the page out.

I cover asynchronous handlers in my other ASP.NET book, *Programming Microsoft ASP.NET 2.0 Applications: Advanced Topics* (Microsoft Press, 2006).

Note A fair number of methods work synchronously even in the context of asynchronous commands. The list includes *BeginXXX* methods and most methods of the data reader class, such as *GetXXX* methods, *Read*, *Close*, and *Dispose*.

Working with Transactions

In ADO.NET, you can choose between two types of transactions: local and distributed. A local transaction involves a single resource—typically, the database you're connected to. You begin the transaction, attach one or more commands to its context, and decide whether the whole operation was successful or whether it failed. The transaction is then committed or rolled back accordingly. This approach is functionally similar to simply running a SQL stored procedure that groups a few commands under the same transaction. Using ADO.NET code makes it more flexible but doesn't change the final effect.

A distributed transaction spans multiple heterogeneous resources and ensures that if the entire transaction is committed or rolled back, all modifications made at the various steps are committed or rolled back as well. A distributed transaction requires a Transaction Processing (TP) monitor. The Distributed Transaction Coordinator (DTC) is the TP monitor for Microsoft Windows 2000 and later.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In ADO.NET 1.x, you manage a local transaction through a bunch of database-specific transaction objects—for example, *SqlTransaction* for SQL Server transactions. You begin the transaction, associate commands to it, and decide about the outcome. For distributed transactions, you need Enterprise Services and serviced components. You can enlist database connections to Enterprise Services DTC managed transactions by using the aforementioned *EnlistDistributedTransaction* method on the connection class.

In ADO.NET 2.0 and beyond, local and distributed transactions can also be managed (more easily, actually) through the new classes defined in the *System.Transactions* namespace—specifically, with the *TransactionScope* class.

Managing Local Transactions as in ADO.NET 1.x

You start a new local transaction through the *BeginTransaction* method of the connection class. You can give the transaction a name and an isolation level. The method maps to the SQL Server implementation of *BEGIN TRANSACTION*. The following code snippet shows the typical flow of a transactional piece of code:

```
SqlTransaction tran;
tran = conn.BeginTransaction();
SqlCommand cmd1 = new SqlCommand(cmdText1);
cmd1.Connection = conn;
cmd1.Transaction = tran;
...
SqlCommand cmd2 = new SqlCommand(cmdText2);
cmd2.Connection = conn;
cmd2.Transaction = tran;
...
try {
  cmd1.ExecuteNonQuery();
  cmd2.ExecuteNonQuery();
  tran.Commit();
}
catch {
  tran.Rollback();
}
finally {
  conn.Close();
}
```

The newly created transaction object operates on the same connection represented by the connection object you used to create it. To add commands to the transaction, you set the *Transaction* property of command objects. Note that if you set the *Transaction* property of a command to a transaction object that is not connected to the same connection, an exception will be thrown as you attempt to execute a statement. Once all the commands have terminated, you call the *Commit* method of the transaction object to complete the transaction, or you call the *Rollback* method to cancel the transaction and undo all changes.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The isolation level of a transaction indicates the locking behavior for the connection. Common values are as follows: *ReadCommitted* (default), *ReadUncommitted*, *RepeatableRead*, and *Serializable*. Imagine a situation in which one transaction changes a value that a second transaction might need to read. *ReadCommitted* locks the row and prevents the second transaction from reading until the change is committed. *ReadUncommitted* doesn't hold locks, thus improving the overall performance. In doing so, though, it allows the second transaction to read a modified row before the original change is committed or rolled back. This is a "dirty read" because if the first transaction rolls the change back, the read value is invalid and there's nothing you can do about it. (Of course, you set *ReadUncommitted* only if dirty reads are not a problem in your scenario.) Note also that disallowing dirty reads also decreases overall system concurrency.

Imagine one transaction reads a committed row; next, another transaction modifies or deletes the row and commits the change. At this point, if the first transaction attempts to read the row again, it will obtain different results. To prevent this, you set the isolation level to *RepeatableRead*. *RepeatableRead* prevents further updates and dirty reads but not other operations that can generate phantom rows. Imagine that a transaction runs a query; next, another transaction does something that modifies the results of the previous query. When the first transaction ends, it returns an inconsistent result to the client. The *Serializable* level prevents concurrent transactions from updating or inserting rows until a given transaction is complete. Table 7-15 summarizes the isolation levels.

TABLE 7-15 Isolation Levels

Level	Dirty Reads	Nonrepeatable	Phantom Rows
ReadUncommitted	Yes	Yes	Yes
ReadCommitted	No	Yes	Yes
RepeatableRead	No	No	Yes
Serializable	No	No	No

The highest isolation level, *Serializable*, provides a high degree of protection against concurrent transactions, but it requires that each transaction complete before any other transaction is allowed to work on the database.

The isolation level can be changed at any time and remains in effect until explicitly changed. If the level is changed during a transaction, the server is expected to apply the new locking level to all statements remaining.

You terminate a transaction explicitly by using the *Commit* or *Rollback* method. The *SqlTransaction* class supports named savepoints in the transaction that can be used to roll back a portion of the transaction. Named savepoints exploit a specific SQL Server feature—the *SAVE TRANSACTION* statement.

This approach to local transactions is possible only in ADO.NET 1.x and is, of course, fully supported in ADO.NET 2.0 and later versions. Let's explore alternative approaches.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Introducing the *TransactionScope* Object

The preceding code based on *BeginTransaction* ties you to a specific database and requires you to start a new transaction to wrap a few database commands. What if you need to work with distinct databases and then, say, send a message to a message queue? In ADO.NET 1.x, you typically create a distributed transaction in Enterprise Services. In ADO.NET 2.0 and beyond, you can perform both local and distributed transactions through a new object—*TransactionScope*. Here's the code:

```
using (TransactionScope ts = new TransactionScope())
{
  using (SqlConnection conn = new SqlConnection(ConnString))
  {
    SqlCommand cmd = new SqlCommand(cmdText, conn);
    cmd.Connection.Open();
    try {
      cmd.ExecuteNonQuery();
    }
    catch (SqlException ex) {
      // Error handling code goes here
      lblMessage.Text = ex.Message;
    }
  }

  // Must call to complete; otherwise abort
  ts.Complete();
}
```

The connection object is defined within the scope of the transaction, so it automatically participates in the transaction. The only thing left to do is commit the transaction, which you do by placing a call to the method *Complete*. If you omit that call, the transaction fails and rolls back no matter what really happened with the command or commands. Needless to say, any exceptions will abort the transaction.

Important You must guarantee that the *TransactionScope* object will be disposed of. By design, the transaction scope commits or rolls back on disposal. Waiting for the garbage collector to kick in and dispose of the transaction scope can be expensive because distributed transactions have a one-minute timeout by default. Keeping multiple databases locked for up to a minute is an excellent scalability killer. Calling *TransactionScope.Dispose* manually in the code might not be enough, as it won't be called in case of exceptions. You should either opt for a *using* statement or a *try/catch/finally* block.

Distributed Transactions with *TransactionScope*

Let's consider a transaction that includes operations on different databases—the Northwind database of SQL Server 2000 and a custom *MyData.mdf* file managed through SQL Server 2005 Express. The file is available in the *app_Data* directory of the sample project in the

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

book's sample code. The sample table we're interested in here can be created with the following command:

```
CREATE TABLE Numbers (ID int, Text varchar(50))
```

You create a unique and all-encompassing *TransactionScope* instance and run the various commands even on different connections. You track the outcome of the various operations and call *Complete* if all went fine. Here's an example:

```
using (TransactionScope ts = new TransactionScope())
{
    // ******************************************************
    // Update Northwind on SQL Server 2000
    using (SqlConnection conn = new SqlConnection(ConnString))
    {
       SqlCommand cmd = new SqlCommand(UpdateCmd, conn);
       cmd.Connection.Open();
       cmd.ExecuteNonQuery(); // note exception aborts transaction
    }

    // ******************************************************
    // Update Numbers on SQL Server 2005
    using (SqlConnection conn = new SqlConnection(ConnString05))
    {
        SqlCommand cmd = new SqlCommand(InsertCmd, conn);
        cmd.Connection.Open();
        cmd.ExecuteNonQuery(); // note exception aborts transaction
    }

    // Must call to complete; otherwise abort
    ts.Complete();
}
```

If an error occurs, say, on the SQL Server 2005 database, any changes successfully entered on the SQL Server 2000 database are automatically rolled back.

TransactionScope is a convenience class that supports the dispose pattern, and internally it simply sets the current transaction, plus it has some state to track scoping. By wrapping everything in a *TransactionScope* object, you're pretty much done, as the object takes care of everything else for you. For example, it determines whether you need a local or distributed transaction, enlists any necessary distributed resources, and proceeds with local processing otherwise. As the code reaches a point where it won't be running locally, *TransactionScope* escalates your transaction to the DTC as appropriate.

Which objects can be enlisted with a transaction? Anything that implements the required interface—*ITransaction*—can be enlisted. ADO.NET 2.0 and later versions ship all standard data providers with support for *System.Transactions*. MSMQ works in compatibility mode.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When some code invokes the *Complete* method, it indicates that all operations in the scope are completed successfully. Note that the method does not physically terminate the distributed transaction, as the commit operation will still happen on *TransactionScope* disposal. However, after calling the method, you can no longer use the distributed transaction.

Note There are a number of differences between *System.Transactions* and Enterprise Services as far as distributed transactions are concerned. First, *System.Transactions* is a transaction framework designed specifically for the managed environment, so it fits more naturally into .NET applications. Of course, internally the classes of the *System.Transactions* namespace might end up delegating some work to DTC and COM+, but that is nothing more than an implementation detail. Another important difference between the two is the existence of a lightweight transaction manager implemented on the managed side that allows for a number of optimizations, including presenting several enlistments as only one for DTC and support for promotable transactions.

Enlisting in a Distributed Transaction in ADO.NET 1.x

If your code uses *TransactionScope*, there's no need for a connection object to explicitly enlist in a transaction. However, if needed, the *EnlistTransaction* method provides you with exactly that capability.

Manually enlisting connections into distributed transactions is a feature already available in ADO.NET 1.1 through the *EnlistDistributedTransaction* method of the connection class. The method manually enlists the connection into a transaction being managed by the Enterprise Services DTC. In this case, you work with a distributed transaction that is defined elsewhere and takes direct advantage of the DTC.

Note *EnlistDistributedTransaction* is useful when you have pooled business objects with an open connection. In this case, enlistment occurs only when the connection is opened. If the object participates in multiple transactions, the connection for that object is not reopened and therefore has no way to automatically enlist in new transactions. In this case, you can disable automatic transaction enlistment and enlist the connection explicitly by using *EnlistDistributedTransaction*.

SQL Server 2005–Specific Enhancements

The .NET data provider for SQL Server also has new features that are tied to the enhancements in SQL Server 2005. SQL Server 2005 introduces significant enhancements in various areas, including data type support, query dependency and notification, and multiple active result sets (MARS).

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Support for CLR Types

SQL Server 2005 supports any CLR types. In addition to default types, you can store into and retrieve from SQL Server tables any object that is a valid .NET type. This includes both system types—such as a *Point*—and user-defined classes. This extended set of capabilities is reflected in the ADO.NET provider for SQL Server.

CLR types appear as objects to the data reader, and parameters to commands can be instances of CLR types. The following code snippet demonstrates how to retrieve a value from the MyCustomers table that corresponds to an instance of the user-defined *Customer* class:

```
string cmdText = "SELECT CustomerData FROM MyCustomers";
SqlConnection conn = new SqlConnection(connStr);
using (conn)
{
  SqlCommand cmd = new SqlCommand(cmdText, conn);
  cmd.Connection.Open();
  SqlDataReader reader = cmd.ExecuteReader();
  while(reader.Read())
  {
    Customer cust = (Customer) reader[0];
    // Do some work
  }
  cmd.Connection.Close();
}
```

A SQL Server 2005 user-defined type is stored as a binary stream of bytes. The *get* accessor of the data reader gets the bytes and deserializes them to a valid instance of the original class. The reverse process (serialization) takes place when a user-defined object is placed in a SQL Server column.

Support for XML as a Native Type

SQL Server 2005 natively supports the XML data type, which means you can store XML data in columns. At first glance, this feature seems to be nothing new because XML data is plain text and to store XML data in a column you only need the column to accept text. Native XML support in SQL Server 2005, however, means something different—you can declare the type of a given column as native XML, not plain text adapted to indicate markup text.

In ADO.NET 1.x, the *ExecuteXmlReader* method allows you to process the results of a query as an XML stream. The method builds an *XmlTextReader* object on top of the data coming from SQL Server. Therefore, for the method to work, the entire result set must be XML. Scenarios in which this method is useful include when the *FOR XML* clause is appended or when you query for a scalar value that happens to be XML text.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In ADO.NET 2.0 and beyond, when SQL Server 2005 is up and running, you can obtain an *XmlTextReader* object for each table cell (row, column) whose type is XML. You obtain a *SqlDataReader* object and have it return XML to you using the new *GetSqlXml* method. The following code snippet provides a useful example:

```
string cmdText = " SELECT * FROM MyCustomers";
SqlCommand cmd = new SqlCommand(cmdText, conn);
SqlDataReader reader = cmd.ExecuteReader();
while(reader.Read())
{
  // Assume that field #3 contains XML data

  // Get data and do some work
  SqlXml xml = reader.GetSqlXml(3);
  ProcessData(xml.Value);
}
```

The *SqlXml* class represents the XML data type. The *Value* property of the class returns the XML text as a string.

SQL Notifications and Dependencies

Applications that display volatile data or maintain a cache would benefit from friendly server notification whenever their data changes. SQL Server 2005 offers this feature—it notifies client applications about dynamic changes in the result set generated by a given query. Suppose your application manages the results of a query. If you register for a notification, your application is informed if something happens at the SQL Server level that modifies the result set generated by that query. This means that if a record originally selected by your query is updated or deleted, or if a new record is added that meets the criteria of the query, you're notified. Note, though, the notification reaches your application only if it is still up and running—which poses a clear issue with ASP.NET pages. But let's move forward one step at a time.

The SQL Server provider in ADO.NET provides two ways to use this notification feature and two related classes—*SqlDependency* and *SqlNotificationRequest*. *SqlNotificationRequest* is a lower-level class that exposes server-side functionality, allowing you to execute a command with a notification request. When a T-SQL statement is executed in SQL Server 2005, the notification mechanism keeps track of the query, and if it detects a change that might cause the result set to change, it sends a message to a queue. A queue is a new SQL Server 2005 database object that you create and manage with a new set of T-SQL statements. How the queue is polled and how the message is interpreted is strictly application-specific.

The *SqlDependency* class provides a high-level abstraction of the notification mechanism, and it allows you to set an application-level dependency on the query so that changes in the

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

server can be immediately communicated to the client application through an event. The following code binds a command to a SQL dependency:

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Employees", conn);
SqlDependency dep = new SqlDependency(cmd);
dep.OnChange += new OnChangeEventHandler(OnDependencyChanged);
SqlDataReader reader = cmd.ExecuteReader();
```

The *OnChange* event on the *SqlDependency* class fires whenever the class detects a change that affects the result set of the command. Here's a typical handler:

```
void OnDependencyChanged(object sender, SqlNotificationsEventArgs e)
{
    ...
}
```

When the underlying machinery detects a change, it fires the event to the application.

As mentioned, using notifications in this way is not particularly interesting from an ASP.NET perspective because the page returns immediately after running the query. However, the caching API of ASP.NET 2.0 (and newer versions) provides a similar feature that automatically tracks the results of a query via the ASP.NET cache. What you have in ASP.NET is a custom type of cache dependency that monitors the results of a query for both SQL Server 2000 and SQL Server 2005, although in radically different ways. You create a dependency on a command or a table and place it in the ASP.NET *Cache* object. The cache item will be invalidated as soon as a change in the monitored command or table is detected. If a SQL Server 2000 instance is involved, you can detect changes to only one of the tables touched by the query; if SQL Server 2005 is involved, you get finer control and can track changes to the result set of the query. We'll cover ASP.NET caching in great detail in Chapter 16.

Multiple Active Result Sets

Version 1.x of the SQL Server managed provider, along with the SQL Server ODBC driver, supports only one active result set per connection. The (unmanaged) OLE DB provider and the outermost ADO library appear to support multiple active result sets, but this is an illusion. In OLE DB, the effect is obtained by opening additional and nonpooled connections.

In SQL Server 2005, the MARS feature is natively implemented and allows an application to have more than one *SqlDataReader* open on a connection, each started from a separate command. Having more than one data reader open on a single connection offers a potential performance improvement because multiple readers are much less expensive than multiple connections. At the same time, MARS adds some hidden per-operation costs that are a result of its implementation. Considering the trade-offs and making a thoughtful decision is up to you.

The canonical use of MARS is when you get a data reader to walk through a result set while using another command on the same connection to issue update statements to the

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

database. The following code demonstrates a sample page that walks through a data reader and updates the current record using a second command. If you try this approach in ADO.NET 1.x, or in ADO.NET 2.0 and later with MARS disabled, you get an exception complaining that the data reader associated with this connection is open and should be closed first.

```
using (SqlConnection conn = new SqlConnection(connString))
{
  SqlCommand cmd1 = new SqlCommand("SELECT * FROM employees", conn);
  cmd1.Connection.Open();
  SqlDataReader reader = cmd1.ExecuteReader();

  // Walks the data reader
  while (reader.Read())
  {
    // Reverses the first name
    string firstNameReversed = reader["firstname"].ToString();
    char[] buf = firstNameReversed.ToCharArray();
    Array.Reverse(buf);
    firstNameReversed = new string(buf);

    // Set the new first name on the same connection
    int id = (int)reader["employeeid"];
    SqlCommand cmd2 = new SqlCommand(
           "UPDATE employees SET firstname=@newFirstName WHERE
            employeeid=@empID", conn);
    cmd2.Parameters.AddWithValue("@newFirstName", firstNameReversed);
    cmd2.Parameters.AddWithValue("empID", id);
    cmd2.ExecuteNonQuery();
  }
  reader.Close();

  // Get a new reader to refresh the UI
  grid.DataSource = cmd1.ExecuteReader();
  grid.DataBind();
  cmd1.Connection.Close();
}
```

Note that for MARS to work, you must use a distinct *SqlCommand* object, as shown in the preceeding code. If you use a third command object to re-execute the query to get up-to-date records, there's no need to close the reader explicitly.

Another big benefit of MARS is that, if you're engaged in a transaction, it lets you execute code in the same isolation-level scope of the original connection. You won't get this benefit if you open a second connection under the covers.

The MARS feature is enabled by default when SQL Server 2005 is the database server. To disable MARS, you set the *MultipleActiveResultSets* attribute to *false* in the connection string. There are some hidden costs associated with MARS. First, MARS requires the continuous creation of *SqlCommand* objects. To deal with this issue, a pool of command objects is constituted and maintained. Second, there is a cost in the network layer due to multiplexing the I/O stream of data. Most of these costs are structural, and you should not expect a great

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

performance improvement by disabling the MARS feature. So what's the purpose of the *MultipleActiveResultSets* attribute? The attribute appears mostly for backward compatibility. In this way, applications that expect an exception when more than one result set is used can continue working.

Note MARS-like behavior is available in the .NET Framework 2.0 (and newer) versions of the OLE DB and Oracle managed providers. The Oracle provider doesn't support the MARS attribute on the connection string, but it enables the feature automatically. The OLE DB provider doesn't support the connection string attribute either—it simulates multiple result sets when you connect to earlier versions of SQL Server or when the Microsoft Data Access Components (MDAC 9.0) library is not available. When you operate through OLE DB on a version of SQL Server 2005 equipped with MDAC 9.0, multiple result sets are active and natively implemented.

Conclusion

The .NET data access subsystem is made of two main subtrees—the managed providers and the database-agnostic container classes. ADO.NET managed providers are a new type of data source connectors and replace the COM-based OLE DB providers of ADO and ASP. As of this writing, the .NET Framework includes two native providers—one for SQL Server and one for Oracle—and support for all OLE DB providers and ODBC drivers. Third-party vendors also support MySQL, DB2, and Sybase, and they have alternate providers for Oracle.

A managed provider is faster and more appropriate than any other database technology for data-access tasks in .NET. Especially effective with SQL Server, a managed provider hooks up at the wire level and removes any sort of abstraction layer. In this way, a managed provider makes it possible for ADO.NET to return to callers the same data types they would use to refresh the user interface. A managed provider supplies objects to connect to a data source, execute a command, start a transaction, and then grab or set some data.

In this chapter, we focused on establishing a connection to the data source and setting up commands and transactions. In the next chapter, we'll complete our look at ADO.NET by exploring data container classes such as *DataSet* and *DataTable*.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Just The Facts

- ADO.NET is a data-access subsystem in the Microsoft .NET Framework and is made of two distinct, but closely related, sets of classes—data providers and data containers.
- The functionalities supplied by a .NET data provider fall into a couple of categories: capability of populating container classes, and capability of setting up a connection and executing commands.
- The .NET Framework comes with data providers for SQL Server, Oracle, and all OLE DB and ODBC data sources.
- A data provider is faster and more appropriate than any other database technology for data-access tasks in .NET. Especially effective with SQL Server, a managed data provider hooks up at the wire level and removes any sort of abstraction layer.
- The data provider supplies an object to establish and manage the connection to a data source. This object implements connection pooling.
- In .NET applications, connection strings can be stored in a special section of the configuration file and encrypted if required.
- The data provider supplies an object to execute commands on an open connection and optionally within a transaction. The command object lists various execute methods to account for query and nonquery commands. Commands can execute either synchronously or asynchronously.
- Data returned by a query command are cached in a data reader object, which is a kind of optimized read-only, forward-only cursor.
- Local and distributed transactions can be managed through the TransactionScope class introduced with ADO.NET 2.0 and also supported by newer versions.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Programming Microsoft® ASP.NET 3.5 By Dino Esposito ISBN: 9780735625273
Publisher: Microsoft Press
Print Publication Date: 2008/02/23
Prepared for Ashwin Srinivas, Safari ID: avsrinivas@gmail.com
User number: 300640
© 2009 Safari Books Online, LLC. This PDF is made available for personal use only during the relevant subscription term, subject to the Safari Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.