Microsoft

.NET Data Access Architecture Guide

patterns & practices

Version 1.0

Contents

.NET Data Access Architecture Guide

Introduction

If you are designing a data access layer for a .NET-based application, you should use Microsoft® ADO.NET as the data access model. ADO.NET is feature rich and supports the data access requirements of loosely coupled, multitier Web applications and Web services. As with other feature-rich object models, ADO.NET offers a number of ways to solve a particular problem.

The .NET Data Access Architecture Guide provides information to help you choose the most appropriate data access approach. It does this by describing a wide range of common data access scenarios, providing performance tips, and prescribing best practices. This guide also provides answers to frequently asked questions, such as: Where is the best place to store database connection strings? How should I implement connection pooling? How should I work with transactions? How should I implement paging to allow users to scroll through large numbers of records?

This guide focuses on the use of ADO.NET to access Microsoft SQL Server™ 2000 by using the SQL Server .NET data provider, one of the two providers shipped with ADO.NET. Where appropriate, this guide highlights any differences that you need to be aware of when you use the OLE DB .NET data provider to access other OLE DB–aware data sources.

For a concrete implementation of a data access component developed using the guidelines and best practices discussed in this document, see the Data Access Application Block. The Data Access Application Block includes the source code for the implementation, and you can use that code directly in your .NET-based applications.

The .NET Data Access Architecture Guide includes the following sections:

- Introducing ADO.NET
- Managing Database Connections
- Error Handling
- Performance
- Connecting Through Firewalls
- Handling BLOBs

- Performing Database Updates with DataSets
- Using Strongly Typed DataSet Objects
- Working with Null Data Fields
- Transactions
- Data Paging

Who Should Read This Document

This document provides guidelines for application architects and enterprise developers who want to build .NET-based applications. Read this document if you are responsible for designing and developing the data tier of a multitier .NET-based application.

What You Must Know

To use this guide to build .NET-based applications, you must have experience developing data access code using ActiveX® Data Objects (ADO) and/or OLE DB, as well as SQL Server experience. You must understand how to develop managed code for the .NET platform, and you must be aware of the fundamental changes that the ADO.NET data access model introduces. For more information about .NET development, see *http://msdn.microsoft.com/net.*

What's New

This document has been updated to include sections on performing database updates, using typed **DataSets**, and using null data fields.

As indicated in the text, some of the content in this guide applies specifically to the Microsoft Visual Studio® 2003 development system and the .NET Framework SDK version 1.1.

Introducing ADO.NET

ADO.NET is the data access model for .NET-based applications. It can be used to access relational database systems such as SQL Server 2000, Oracle, and many other data sources for which there is an OLE DB or ODBC provider. To a certain extent, ADO.NET represents the latest evolution of ADO technology. However, ADO.NET introduces some major changes and innovations that are aimed at the loosely coupled—and inherently disconnected—nature of Web applications. For a comparison of ADO and ADO.NET, see the MSDN article "ADO.NET for the ADO Programmer," at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us /dndotnet/html/adonetprogmsdn.asp.*

One of the key changes that ADO.NET introduces is the replacement of the ADO **Recordset** object with a combination of the **DataTable**, **DataSet**, **DataAdapter**, and **DataReader** objects. A **DataTable** represents a collection of rows from a single table, and in this respect is similar to the **Recordset**. A **DataSet** represents a collection of **DataTable** objects, together with the relationships and constraints that bind the various tables together. In effect, the **DataSet** is an in-memory relational structure with built-in XML support.

One of the key characteristics of the **DataSet** is that it has no knowledge of the underlying data source that might have been used to populate it. It is a disconnected, stand-alone entity used to represent a collection of data, and it can be passed from component to component through the various layers of a multitier application. It can also be serialized as an XML data stream, which makes it ideally suited for data transfer between heterogeneous platforms. ADO.NET uses the **DataAdapter** object to channel data to and from the **DataSet** and the underlying data source. The **DataAdapter** object also provides enhanced batch update features previously associated with the **Recordset**.

Figure 1 on the next page shows the full **DataSet** object model.

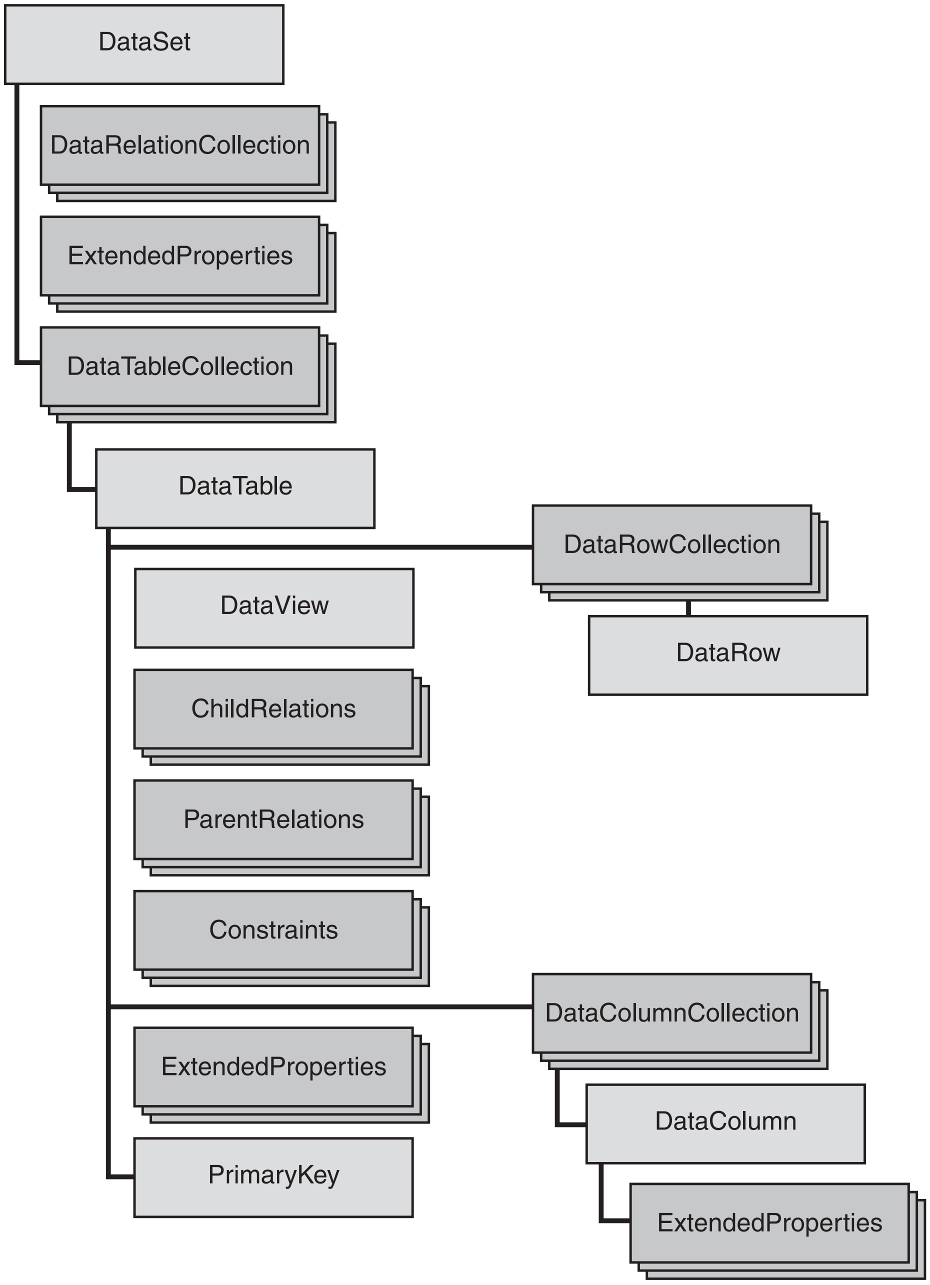

Figure 1.1
DataSet object model

.NET Data Providers

ADO.NET relies on the services of .NET data providers. These provide access to the underlying data source, and they comprise four key objects (**Connection**, **Command**, **DataReader**, and **DataAdapter**).

Currently, ADO.NET ships with two categories of providers: bridge providers and native providers. Bridge providers, such as those supplied for OLE DB and ODBC, allow you to use data libraries designed for earlier data access technologies. Native providers, such as the SQL Server and Oracle providers, typically offer performance improvements due, in part, to the fact that there is one less layer of abstraction.

- **The SQL Server .NET Data Provider**. This is a provider for Microsoft SQL Server 7.0 and later databases. It is optimized for accessing SQL Server, and it communicates directly with SQL Server by using the native data transfer protocol of SQL Server.

  Always use this provider when you connect to SQL Server 7.0 or SQL Server 2000.
- **The Oracle .NET Data Provider**. The .NET Framework Data Provider for Oracle enables data access to Oracle data sources through Oracle client connectivity software. The data provider supports Oracle client software version 8.1.7 and later.
- **The OLE DB .NET Data Provider**. This is a managed provider for OLE DB data sources. It is slightly less efficient than the SQL Server .NET Data Provider, because it calls through the OLE DB layer when communicating with the database. Note that this provider does not support the OLE DB provider for Open Database Connectivity (ODBC), MSDASQL. For ODBC data sources, use the ODBC .NET Data Provider (described later) instead. For a list of OLE DB providers that are compatible with ADO.NET, see *http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconadonetproviders.asp.*

Other .NET data providers currently in beta testing include:

- **The ODBC .NET Data Provider**. The .NET Framework Data Provider for ODBC uses native ODBC Driver Manager (DM) to enable data access by means of COM interoperability.
- **A managed provider for retrieving XML from SQL Server 2000**. The XML for SQL Server Web update 2 (currently in beta) includes a managed provider specifically for retrieving XML from SQL Server 2000. For more information about this update, see *http://msdn.microsoft.com/library/default.asp?url=/nhp/default.asp?contentid=28001300.*

For a more detailed overview of the different data providers, see ".NET Framework Data Providers" in the *.NET Framework Developer's Guide*, at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconadonetproviders.asp.*

Namespace Organization

The types (classes, structs, enums, and so on) associated with each .NET data provider are located in their own namespaces:

- **System.Data.SqlClient**. Contains the SQL Server .NET Data Provider types.
- **System.Data.OracleClient**. Contains the Oracle .NET Data Provider
- **System.Data.OleDb**. Contains the OLE DB .NET Data Provider types.
- **System.Data.Odbc**. Contains the ODBC .NET Data Provider types.
- **System.Data**. Contains provider-independent types such as the **DataSet** and **DataTable**.

Within its associated namespace, each provider provides an implementation of the **Connection**, **Command**, **DataReader**, and **DataAdapter** objects. The **SqlClient** implementations are prefixed with "Sql" and the **OleDb** implementations are prefixed with "OleDb." For example, the **SqlClient** implementation of the **Connection** object is **SqlConnection**, and the **OleDb** equivalent is **OleDbConnection**. Similarly, the two incarnations of the **DataAdapter** object are **SqlDataAdapter** and **OleDbDataAdapter**, respectively.

In this guide, the examples are drawn from the SQL Server object model. Although not illustrated here, similar features are available in Oracle/OLEDB and ODBC.

Generic Programming

If you are likely to target different data sources and want to move your code from one to the other, consider programming to the **IDbConnection**, **IDbCommand**, **IDataReader**, and **IDbDataAdapter** interfaces located within the **System.Data** namespace. All implementations of the **Connection**, **Command**, **DataReader**, and **DataAdapter** objects must support these interfaces.

For more information about implementing .NET data providers, see *http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconimplementingnetdataprovider.asp.*

It should also be noted that both the OLE DB and ODBC bridging providers are alternatives if an application uses a single object model to access multiple databases. In this situation, it is important to consider the application's need for flexibility, and the extent to which database-specific functionality is required, in comparison with the application's need for performance.

Figure 2 illustrates the data access stack and how ADO.NET relates to other data access technologies, including ADO and OLE DB. It also shows the two managed providers and the principal objects within the ADO.NET model.

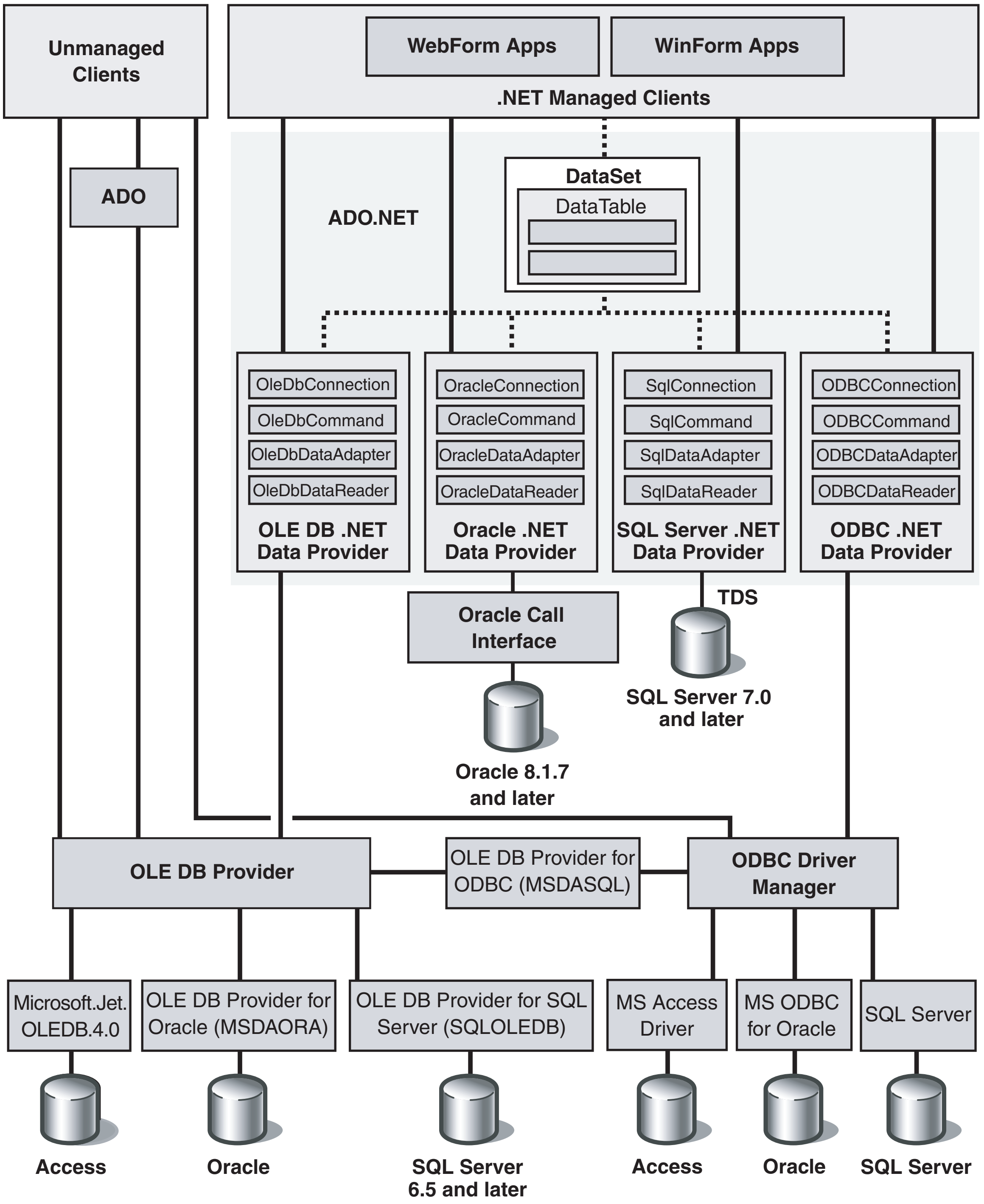

Figure 1.2
Data access stack

For more information about the evolution of ADO to ADO.NET, see the article "Introducing ADO+: Data Access Services for the Microsoft .NET Framework" in the November 2000 issue of MSDN Magazine, at *http://msdn.microsoft.com/msdnmag/issues/1100/adoplus/default.aspx.*

Stored Procedures vs. Direct SQL

Most code fragments shown in this document use **SqlCommand** objects to call stored procedures to perform database manipulation. In some cases, you will not see the **SqlCommand** object because the stored procedure name is passed directly to a **SqlDataAdapter** object. Internally, this still results in the creation of a **SqlCommand** object.

You should use stored procedures instead of embedded SQL statements for a number of reasons:

- Stored procedures generally result in improved performance because the database can optimize the data access plan used by the procedure and cache it for subsequent reuse.
- Stored procedures can be individually secured within the database. A client can be granted permissions to execute a stored procedure without having any permissions on the underlying tables.
- Stored procedures result in easier maintenance because it is generally easier to modify a stored procedure than it is to change a hard-coded SQL statement within a deployed component.
- Stored procedures add an extra level of abstraction from the underlying database schema. The client of the stored procedure is isolated from the implementation details of the stored procedure and from the underlying schema.
- Stored procedures can reduce network traffic, because SQL statements can be executed in batches rather than sending multiple requests from the client.

The SQL Server online documentation strongly recommends that you do not create any stored procedures using "sp_" as a name prefix because such names have been designated for system stored procedures. SQL Server always looks for stored procedures beginning with sp_ in this order:

1. Look for the stored procedure in the master database.
2. Look for the stored procedure based on any qualifiers provided (database name or owner).
3. Look for the stored procedure, using **dbo** as the owner if an owner is not specified.

Properties vs. Constructor Arguments

You can set specific property values of ADO.NET objects either through constructor arguments or by directly setting the properties. For example, the following code fragments are functionally equivalent.

```
// Use constructor arguments to configure command object
SqlCommand cmd = new SqlCommand( "SELECT * FROM PRODUCTS", conn );

// The above line is functionally equivalent to the following
// three lines which set properties explicitly
sqlCommand cmd = new SqlCommand();
cmd.Connection = conn;
cmd.CommandText = "SELECT * FROM PRODUCTS";
```

From a performance perspective, there is negligible difference between the two approaches because setting and getting properties against .NET objects is more efficient than performing similar operations against COM objects.

The choice is one of personal preference and coding style. The explicit setting of properties does, however, make the code easier to comprehend (particularly if you are not familiar with the ADO.NET object model) and easier to debug.

Note: In the past, developers of the Microsoft Visual Basic® development system were advised to avoid creating objects with the "Dim x As New…" construct. In the world of COM, this code could result in the short circuit of the COM object creation process, leading to some subtle and some not-so-subtle bugs. In the .NET world, however, this is no longer an issue.

Managing Database Connections

Database connections represent a critical, expensive, and limited resource, particularly in a multitier Web application. It is imperative that you manage your connections correctly because your approach can significantly affect the overall scalability of your application. Also, think carefully about where to store connection strings. You need a configurable and secure location.

When managing database connections and connection strings, you should strive to:

- Help realize application scalability by multiplexing a pool of database connections across multiple clients.
- Adopt a configurable and high performance connection pooling strategy.
- Use Windows authentication when accessing SQL Server.
- Avoid impersonation in the middle tier.
- Store connection strings securely.
- Open database connections late and close them early.

This section discusses connection pooling and will help you choose an appropriate connection pooling strategy. This section also considers how you should manage, store, and administer your database connection strings. Finally, this section presents two coding patterns that you can use to help ensure that connections are closed reliably and returned to the connection pool.

Using Connection Pooling

Database connection pooling allows an application to reuse an existing connection from a pool instead of repeatedly establishing a new connection with the database. This technique can significantly increase the scalability of an application, because a limited number of database connections can serve a much larger number of clients. This technique also improves performance, because the significant time required to establish a new connection can be avoided.

Data access technologies such as ODBC and OLE DB provide forms of connection pooling, which are configurable to varying degrees. Both approaches are largely transparent to the database client application. OLE DB connection pooling is often referred to as session or resource pooling.

For a general discussion of pooling within Microsoft Data Access Components (MDAC), see “Pooling in the Microsoft Data Access Components,” at *http://msdn.microsoft.com/library/en-us/dnmdac/html/pooling2.asp.*

ADO.NET data providers provide transparent connection pooling, the exact mechanics of which vary for each provider. This section discusses connection pooling in relation to:

- The SQL Server .NET Data Provider
- The Oracle .NET Data Provider
- The OLE DB .NET Data Provider
- The ODBC .NET Data Provider

Pooling with the SQL Server .NET Data Provider

If you are using the SQL Server .NET Data Provider, use the connection pooling support offered by the provider. It is a transaction-aware and efficient mechanism implemented internally by the provider, within managed code. Pools are created on a per application domain basis, and pools are not destroyed until the application domain is unloaded.

You can use this form of connection pooling transparently, but you should be aware of how pools are managed and of the various configuration options that you can use to fine-tune connection pooling.

In many cases, the default connection pooling settings for the SQL Server .NET data provider may be sufficient for your application. During the development and testing of your .NET-based application, it is recommended that you simulate projected traffic patterns to determine if modifications to the connection pool size are required.

Developers building scalable, high performance applications should minimize the amount of time a connection is used, keeping it open for only as long as it takes to retrieve or update data. When a connection is closed, it is returned to the connection pool and made available for reuse. In this case, the actual connection to the database is not severed; however, if connection pooling is disabled, the actual connection to the database will be closed.

Developers should be careful not to rely on the garbage collector to free connections because a connection is not necessarily closed when the reference goes out of scope. This a common source of connection leaks, resulting in connection exceptions when new connections are requested.

Configuring SQL Server .NET Data Provider Connection Pooling

You can configure connection pooling by using a set of name-value pairs, supplied by means of the connection string. For example, you can configure whether or not pooling is enabled (it is enabled by default), the maximum and minimum pool sizes, and the amount of time that a queued request to open a connection can block. The following is an example connection string that configures the maximum and minimum pool sizes.

```
"Server=(local); Integrated Security=SSPI; Database=Northwind;
Max Pool Size=75; Min Pool Size=5"
```

When a connection is opened and a pool is created, multiple connections are added to the pool to bring the connection count to the configured minimum level. Connections can be subsequently added to the pool up to the configured maximum pool count. When the maximum count is reached, new requests to open a connection are queued for a configurable duration.

Choosing Pool Sizes

Being able to establish a maximum threshold is very important for large-scale systems that manage the concurrent requests of many thousands of clients. You need to monitor connection pooling and the performance of your application to determine the optimum pool sizes for your system. The optimum size also depends on the hardware on which you are running SQL Server.

During development, you might want to reduce the default maximum pool size (currently 100) to help find connection leaks.

If you establish a minimum pool size, you will incur a small performance overhead when the pool is initially populated to bring it to that level, although the first few clients that connect will benefit. Note that the process of creating new connections is serialized, which means that your server will not be flooded with simultaneous requests when a pool is being initially populated.

For more details about monitoring connection pooling, see the Monitoring Connection Pooling section in this document. For a complete list of connection pooling connection string keywords, see "Connection Pooling for the .NET Framework Data Provider for SQL Server" in the *.NET Framework Developer's Guide*, at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html /cpconconnectionpoolingforsqlservernetdataprovider.asp.*

More Information

When using SQL Server .NET Data Provider connection pooling, be aware of the following:

- Connections are pooled through an exact match algorithm on the connection string. The pooling mechanism is even sensitive to spaces between name-value pairs. For example, the following two connection strings will result in two separate pools because the second contains an extra space character.

```
SqlConnection conn = new SqlConnection(
         "Integrated Security=SSPI;Database=Northwind");
conn.Open(); // Pool A is created

SqlConmection conn = new SqlConnection(
         "Integrated Security=SSPI ; Database=Northwind");
conn.Open(); // Pool B is created (extra spaces in string)
```

- The connection pool is divided into multiple transaction-specific pools and one pool for connections not currently enlisted in a transaction. For threads associated with a particular transaction context, a connection from the appropriate pool (containing connections enlisted with that transaction) is returned. This makes working with enlisted connections a transparent process.

Pooling with the OLE DB .NET Data Provider

The OLE DB .NET Data Provider pools connections by using the underlying services of OLE DB resource pooling. You have a number of options for configuring resource pooling:

- You can use the connection string to configure, enable, or disable resource pooling.
- You can use the registry.
- You can programmatically configure resource pooling.

To circumvent registry-related deployment issues, avoid using the registry to configure OLE DB resource pooling.

For more details about OLE DB resource pooling, see "Resource Pooling" in Chapter 19, "OLE DB Services" of the *OLE DB Programmer's Reference*, at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/oledb/htm/olprcore_chapter19.asp.*

Managing Connection Pooling with Pooled Objects

As Windows DNA developers, you were encouraged to disable OLE DB resource pooling and/or ODBC connection pooling and use COM+ object pooling as a technique to pool database connections. There are two primary reasons for this:

- Pool sizes and thresholds can be explicitly configured (in the COM+ catalog).
- Performance is improved. The pooled object approach can outperform native pooling by a factor of two.

However, because the SQL Server .NET Data Provider uses pooling internally, you no longer need to develop your own object pooling mechanism (when using this provider). You can thus avoid the complexities associated with manual transaction enlistment.

You might want to consider COM+ object pooling if you are using the OLE DB .NET Data Provider to benefit from superior configuration and improved performance. If you develop a pooled object for this purpose, you must disable OLE DB resource pooling and automatic transaction enlistment (for example, by including "OLE DB Services=-4" in the connection string). You must handle transaction enlistment within your pooled object implementation.

Monitoring Connection Pooling

To monitor your application's use of connection pooling, you can use the Profiler tool that ships with SQL Server, or the Performance Monitor tool that ships with the Microsoft Windows® 2000 operating system.

► **To monitor connection pooling with SQL Server Profiler**

1. Click **Start**, point to **Programs**, point to **Microsoft SQL Server**, and then click **Profiler** to start Profiler.
2. On the **File** menu, point to **New**, and then click **Trace**.
3. Supply connection details, and then click **OK**.
4. In the **Trace Properties** dialog box, click the **Events** tab.
5. In the **Selected event classes** list, ensure that the **Audit Login** and **Audit Logout** events are shown beneath **Security Audit**. To make the trace clearer, remove all other events from the list.
6. Click **Run** to start the trace. You will see **Audit Login** events when connections are established and **Audit Logout** events when connections are closed.

► **To monitor connection pooling with Performance Monitor**

1. Click **Start**, point to **Programs**, point to **Administrative Tools**, and then click **Performance** to start Performance Monitor.
2. Right-click the graph background, and then click **Add Counters**.
3. In the **Performance object** drop-down list, click **SQL Server: General Statistics**.
4. In the list that appears, click **User Connections**.
5. Click **Add**, and then click **Close**.

Managing Security

Although database connection pooling improves the overall scalability of your application, it means you can no longer manage security at the database. This is because to support connection pooling, the connection strings must be identical. If you need to track database operations on a per user basis, consider adding a parameter through which you can pass the user identity and manually log user actions in the database. You need to add this parameter to each operation.

Using Windows Authentication

You should use Windows authentication when connecting to SQL Server because it provides a number of benefits:

- Security is easier to manage because you work with a single (Windows) security model rather than the separate SQL Server security model.
- You avoid embedding user names and passwords in connection strings.
- User names and passwords are not passed over the network in clear text.
- Logon security improves through password expiration periods, minimum lengths, and account lockout after multiple invalid logon requests.

More Information

When you use Windows authentication to access SQL Server, use the following guidelines:

- **Consider performance tradeoffs**. Performance tests have shown that it takes longer to open a pooled database connection when using Windows authentication as compared to using SQL Server authentication. The .NET runtime version 1.1 has reduced the margin by which SQL Server security outperforms Windows authentication, but SQL Server authentication is still faster.

  However, although Windows authentication is still more expensive, the performance reduction is relatively insignificant in comparison to the time it takes to execute a command or stored procedure. As a result, in most cases the security benefits of using Windows authentication outweigh this slight performance degradation. Before making a decision, assess the performance requirements of your application.

- **Avoid impersonation in the middle tier**. Windows authentication requires a Windows account for database access. Although it might seem logical to use impersonation in the middle tier, avoid doing so because it defeats connection pooling and has a severe impact on application scalability.

  To address this problem, consider impersonating a limited number of Windows accounts (rather than the authenticated principal) with each account representing a particular role.

  For example, you can use this approach:

  1. Create two Windows accounts, one for read operations and one for write operations. (Or, you might want separate accounts to mirror application-specific roles. For example, you might want to use one account for Internet users and another for internal operators and/or administrators.)
  2. Map each account to a SQL Server database role, and establish the necessary database permissions for each role.
  3. Use application logic in your data access layer to determine which Windows account to impersonate before you perform a database operation.

  Note: Each account must be a domain account with Internet Information Services (IIS) and SQL Server in the same domain or in trusted domains. Or, you can create matching accounts (with the same name and password) on each computer.

- **Use TCP/IP for your network library**. SQL Server 7.0 and later support Windows authentication for all network libraries. Use TCP/IP to gain configuration, performance, and scalability benefits. For more information about using TCP/IP, see the Connecting Through Firewalls section in this document.

For general guidance on developing secure ASP.NET and Web applications, refer to the following Microsoft *patterns & practices* guides:

- Volume I, *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication,* available at *http://www.microsoft.com/practices*
- Volume II, *Improving Web Application Security: Threats and Countermeasures*, which will be available at *http://www.microsoft.com/practices*

Storing Connection Strings

To store database connection strings, you have a variety of options with different degrees of flexibility and security. Although hard coding a connection string within source code offers the best performance, file system caching ensures that the performance degradation associated with storing the string externally in the file system is negligible. The extra flexibility provided by an external connection string, which supports administrator configuration, is preferred in virtually all cases.

When you are choosing an approach for connection string storage, the two most important considerations are security and ease of configuration, closely followed by performance.

You can choose among the following locations for storing database connection strings:

- In an application configuration file; for example, Web.config for an ASP.NET Web application
- In a Universal Data Link (UDL) file (supported only by the OLE DB .NET Data Provider)
- In the Windows registry
- In a custom file
- In the COM+ catalog, by using construction strings (for serviced components only)

By using Windows authentication to access SQL Server, you can avoid storing user names and passwords in connection strings. If your security requirements demand more stringent measures, consider storing the connection strings in encrypted format.

For ASP.NET Web applications, storing the connection strings in encrypted format within the Web.config file represents a secure and configurable solution.

Note: You can set the **Persist Security Info** named value to false in the connection string to prevent security-sensitive details, such as the password, from being returned by means of the **ConnectionString** property of the **SqlConnection** or **OleDbConnection** objects.

The following subsections discuss how to use the various options to store connection strings, and they present the relative advantages and disadvantages of each approach. This will allow you to make an informed choice based on your specific application scenario.

Note: The Configuration Application Management block allows you to manage configuration settings — from database connections to complex hierarchical data. For more information, see *http://msdn.microsoft.com/practices.*

Using XML Application Configuration Files

You can use the **<appSettings>** element to store a database connection string in the custom settings section of an application configuration file. This element supports arbitrary key-value pairs, as illustrated in the following fragment:

```
<configuration>
 <appSettings>
  <add key="DBConnStr"
```

```
    value="server=(local);Integrated Security=SSPI;database=northwind"/>
 </appSettings>
</configuration>
```

Note: The **<appSettings>** element appears under the **<configuration>** element and not directly under **<system.web>**.

Advantages

- **Ease of deployment**. The connection string is deployed along with the configuration file through regular .NET **xcopy** deployment.
- **Ease of programmatic access**. The **AppSettings** property of the **ConfigurationSettings** class makes reading the configured database connection string an easy task at run time.
- **Support of dynamic update (ASP.NET only)**. If an administrator updates the connection string in a Web.config file, the change will be picked up the next time the string is accessed, which for a stateless component is likely to be the next time a client uses the component to make a data access request.

Disadvantages

- **Security**. Although the ASP.NET Internet Server Application Programming Interface (ISAPI) dynamic-link library (DLL) prevents clients from directly accessing files with a .config file extension and NTFS permissions can be used to further restrict access, you might still want to avoid storing these details in clear text on a front-end Web server. For added security, store the connection string in encrypted format in the configuration file.

More Information

- You can retrieve custom application settings by using the static **AppSettings** property of the **System.Configuration.ConfigurationSettings** class. This is shown in the following code fragment, which assumes the previously illustrated custom key called **DBConnStr**:

  ```
  using System.Configuration;
  private string GetDBaseConnectionString()
  {
    return ConfigurationSettings.AppSettings["DBConnStr"];
  }
  ```

- For more information about configuring .NET Framework applications, see *http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconconfiguringnetframeworkapplications.asp.*

Using UDL Files

The OLE DB .NET Data Provider supports Universal Data Link (UDL) file names in its connection string. You can pass the connection string by using construction arguments to the **OleDbConnection** object, or you can set the connection string by using the object's **ConnectionString** property.

Note: The SQL Server .NET Data Provider does not support UDL files in its connection string. Therefore, this approach is available to you only if you are using the OLE DB .NET Data Provider.

For the OLE DB provider, to reference a UDL file with the connection string, use "File Name=name.udl."

Advantages

- **Standard approach**. You might already be using UDL files for connection string management.

Disadvantages

- **Performance**. Connection strings that contain UDLs are read and parsed each time the connection is opened.
- **Security**. UDL files are stored as plain text. You can secure these files by using NTFS file permissions, but doing so raises the same issues as with .config files.
- **SqlClient does not support UDL files**. This approach is not supported by the SQL Server .NET Data Provider, which you use to access SQL Server 7.0 and later.

More Information

- To support administration, make sure that administrators have read/write access to the UDL file and that the identity used to run your application has read access. For ASP.NET Web applications, the application worker process runs by using the SYSTEM account by default, although you can override this by using the **<processModel>** element of the machine-wide configuration file (Machine.config). You can also impersonate, optionally with a nominated account, by using the **<identity>** element of the Web.config file.
- For Web applications, make sure that you do not place the UDL file in a virtual directory, which would make the file downloadable over the Web.
- For more information about these and other security-related ASP.NET features, see "Authentication in ASP.NET: .NET Security Guidance," at *http://msdn.microsoft.com/library/en-us/dnbda/html/authaspdotnet.asp.*

Using the Windows Registry

You can also use a custom key in the Windows registry to store the connection string, although this is not recommended due to deployment issues.

Advantages

- **Security**. You can manage access to selected registry keys by using access control lists (ACLs). For even higher levels of security, consider encrypting the data.
- **Ease of programmatic access**. .NET classes are available to support reading strings from the registry.

Disadvantages

- **Deployment**. The relevant registry setting must be deployed along with your application, somewhat defeating the advantage of **xcopy** deployment.

Using a Custom File

You can use a custom file to store the connection string. However, this technique offers no advantages and is not recommended.

Advantages

- None.

Disadvantages

- **Extra coding**. This approach requires extra coding and forces you to deal explicitly with concurrency issues.
- **Deployment**. The file must be copied along with the other ASP.NET application files. Avoid placing the file in the ASP.NET application directory or subdirectory to prevent it from being downloaded over the Web.

Using Construction Arguments and the COM+ Catalog

You can store the database connection string in the COM+ catalog and have it automatically passed to your object by means of an object construction string. COM+ will call the object's **Construct** method immediately after instantiating the object, supplying the configured construction string.

Note: This approach works only for serviced components. Consider it only if your managed components use other services, such as distributed transaction support or object pooling.

Advantages

- **Administration**. An administrator can easily configure the connection string by using the Component Services MMC snap-in.

Disadvantages

- **Security**. The COM+ catalog is considered a non-secure storage area (although you can restrict access with COM+ roles) and therefore must not be used to maintain connection strings in clear text.

- **Deployment**. Entries in the COM+ catalog must be deployed along with your .NET-based application. If you are using other enterprise services, such as distributed transactions or object pooling, storing the database connection string in the catalog presents no additional deployment overhead, because the COM+ catalog must be deployed to support those other services.
- **Components must be serviced**. You can use construction strings only for serviced components. You should not derive your component's class from **ServicedComponent** (making your component serviced) simply to enable construction strings.

Important: It is critical to secure connection strings. With SQL authentication, the connection contains a user name and password. If an attacker exploits a source code vulnerability on the Web server or gains access to the configuration store, the database will be vulnerable. To prevent this, connection strings should be encrypted. For descriptions of different methods available to encrypt plaintext connection strings, see *Improving Web Application Security: Threats and Countermeasures*, which will be available at *http://www.microsoft.com/practices.*

More Information

- For more information about how to configure a .NET class for object construction, see How To Enable Object Construction For a .NET Class in the appendix.
- For more information about developing serviced components, see *http://msdn.microsoft.com/library/en-us/cpguidnf/html/cpconwritingservicedcomponents.asp.*
- For general guidance on developing secure ASP.NET and Web applications, refer to the following Microsoft *patterns & practices* guides:
  - Volume I, *Building Secure ASP.NET Applications: Authentication, Authorization, and Secure Communication*, available at *http://www.microsoft.com/practices*
  - Volume II, *Improving Web Application Security: Threats and Countermeasures*, which will be available at *http://www.microsoft.com/practices*

Connection Usage Patterns

Irrespective of the .NET data provider you use, you must always:

- Open a database connection as late as possible.
- Use the connection for as short a period as possible.
- Close the connection as soon as possible. The connection is not returned to the pool until it is closed through either the **Close** or **Dispose** method. You should also close a connection even if you detect that it has entered the broken state. This ensures that it is returned to the pool and marked as invalid. The object pooler periodically scans the pool, looking for objects that have been marked as invalid.

To guarantee that the connection is closed before a method returns, consider one of the approaches illustrated in the two code samples that follow. The first uses a

finally block. The second uses a C# **using** statement, which ensures that an object's **Dispose** method is called.

The following code ensures that a **finally** block closes the connection. Note that this approach works for both Visual Basic .NET and C# because Visual Basic .NET supports structured exception handling.

```
public void DoSomeWork()
{
  SqlConnection conn = new SqlConnection(connectionString);
  SqlCommand cmd = new SqlCommand("CommandProc", conn );
  cmd.CommandType = CommandType.StoredProcedure;

  try
  {
    conn.Open();
    cmd.ExecuteNonQuery();
  }
  catch (Exception e)
  {
    // Handle and log error
  }
  finally
  {
    conn.Close();
  }
}
```

The following code shows an alternate approach that uses a C# **using** statement. Note that Visual Basic .NET does not provide a **using** statement or any equivalent functionality.

```
public void DoSomeWork()
{
  // using guarantees that Dispose is called on conn, which will
  // close the connection.
  using (SqlConnection conn = new SqlConnection(connectionString))
  {
    SqlCommand cmd = new SqlCommand("CommandProc", conn);
    fcmd.CommandType = CommandType.StoredProcedure;
    conn.Open();
    cmd.ExecuteQuery();
  }
}
```

You can also apply this approach to other objects—for example, **SqlDataReader** or **OleDbDataReader**—which must be closed before anything else can be done with the current connection.

Error Handling

ADO.NET errors are generated and handled through the underlying structured exception handling support that is native to the .NET Framework. As a result, you handle errors within your data access code in the same way that you handle errors elsewhere in your application. Exceptions can be detected and handled through standard .NET exception handling syntax and techniques.

This section shows you how to develop robust data access code and explains how to handle data access errors. It also provides specific exception handling guidance relating to the SQL Server .NET Data Provider.

.NET Exceptions

The .NET data providers translate database-specific error conditions into standard exception types, which you should handle in your data access code. The database-specific error details are made available to you through properties of the relevant exception object.

All .NET exception types ultimately are derived from the base **Exception** class in the **System** namespace. The .NET data providers throw provider-specific exception types. For example, the SQL Server .NET Data Provider throws **SqlException** objects whenever SQL Server returns an error condition. Similarly, the OLE DB .NET Data Provider throws exceptions of type **OleDbException**, which contain details exposed by the underlying OLE DB provider.

Figure 3 shows the .NET data provider exception hierarchy. Notice that the **OleDbException** class is derived from **ExternalException**, the base class for all COM Interop exceptions. The **ErrorCode** property of this object stores the COM HRESULT generated by OLE DB.

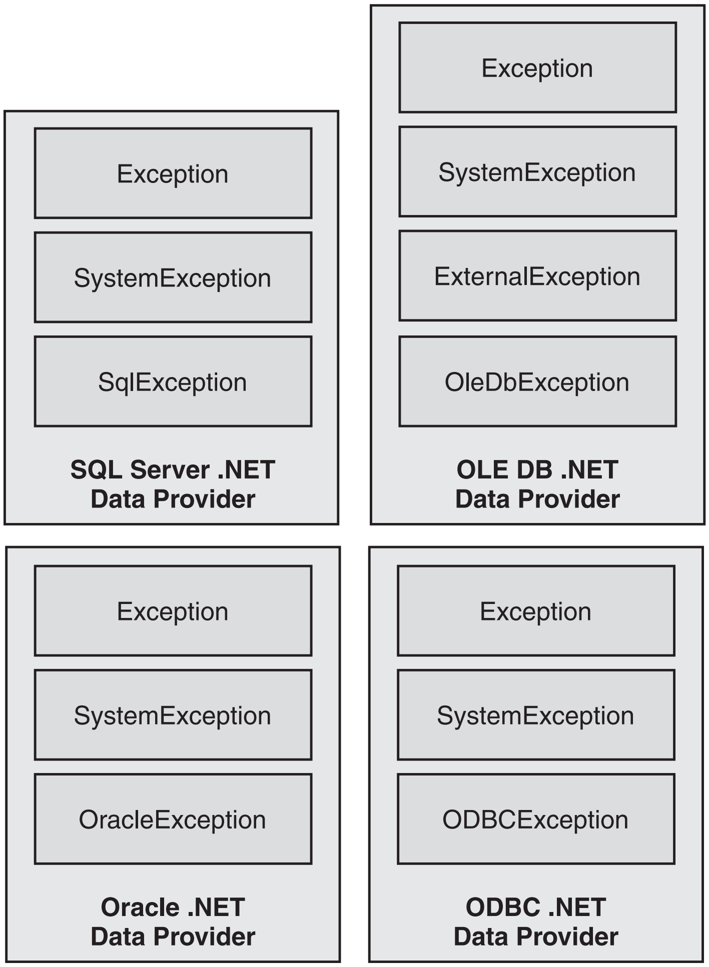

Figure 1.3
.NET Data Provider exception hierarchy

Catching and Handling .NET Exceptions

To handle data access exception conditions, place your data access code within a **try** block and trap any exceptions generated by using **catch** blocks with the appropriate filter. For example, when writing data access code by using the SQL Server .NET Data Provider, you should catch exceptions of type **SqlException**, as shown in the following code:

```
try
{
  // Data access code
}
catch (SqlException sqlex) // more specific
```

```
{
}
catch (Exception ex) // less specific
{
}
```

If you provide more than one **catch** statement with differing filter criteria, remember to order them from most specific type to least specific type. That way, the most specific type of **catch** block is executed for any given exception type.

This **SqlException** class exposes properties that contain details of the exception condition. These include:

- A **Message** property that contains text describing the error.
- A **Number** property that contains the error number, which uniquely identifies the type of error.
- A **State** property that contains additional information about the invocation state of the error. This is usually used to indicate a particular occurrence of a specific error condition. For example, if a single stored procedure can generate the same error from more than one line, the state should be used to identify the specific occurrence.
- An **Errors** collection, which contains detailed error information about the errors that SQL Server generates. The **Errors** collection will always contain at least one object of type **SqlError**.

The following code fragment illustrates how to handle a SQL Server error condition by using the SQL Server .NET Data Provider:

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

// Method exposed by a Data Access Layer (DAL) Component
public string GetProductName( int ProductID )
{
  SqlConnection conn = null;
  // Enclose all data access code within a try block
  try
  {
    conn = new SqlConnection(
        "server=(local);Integrated Security=SSPI;database=northwind");
    conn.Open();
    SqlCommand cmd = new SqlCommand("LookupProductName", conn );
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add("@ProductID", ProductID );
    SqlParameter paramPN =
         cmd.Parameters.Add("@ProductName", SqlDbType.VarChar, 40 );
    paramPN.Direction = ParameterDirection.Output;
    cmd.ExecuteNonQuery();
    // The finally code is executed before the method returns
    return paramPN.Value.ToString();
  }
  catch (SqlException sqlex)
  {
    // Handle data access exception condition
    // Log specific exception details
    LogException(sqlex);
    // Wrap the current exception in a more relevant
    // outer exception and re-throw the new exception
    throw new DALException(
                  "Unknown ProductID: " + ProductID.ToString(), sqlex );
  }
  catch (Exception ex)
  {
    // Handle generic exception condition . . .
    throw ex;
  }
  finally
  {
    if(conn != null) conn.Close(); // Ensures connection is closed
  }
}

// Helper routine that logs SqlException details to the
// Application event log
private void LogException( SqlException sqlex )
{
  EventLog el = new EventLog();
  el.Source = "CustomAppLog";
  string strMessage;
  strMessage = "Exception Number : " + sqlex.Number +
               "(" + sqlex.Message + ") has occurred";
  el.WriteEntry( strMessage );

  foreach (SqlError sqle in sqlex.Errors)
  {
    strMessage = "Message: " + sqle.Message +
                 " Number: " + sqle.Number +
                 " Procedure: " + sqle.Procedure +
                 " Server: " + sqle.Server +
                 " Source: " + sqle.Source +
                 " State: " + sqle.State +
                 " Severity: " + sqle.Class +
                 " LineNumber: " + sqle.LineNumber;
    el.WriteEntry( strMessage );
  }
}
```

Within the **SqlException catch** block, the code initially logs the exception details by using the **LogException** helper function. This function uses a **foreach** statement to enumerate the provider-specific details within the **Errors** collection and records the error details to the error log. The code within the **catch** block then wraps the SQL Server-specific exception within an exception of type **DALException**, which is more meaningful to the callers of the **GetProductName** method. The exception handler uses the **throw** keyword to propagate this exception back to the caller.

More Information

- For a full list of members of the **SqlException** class, see *http://msdn.microsoft.com/library/en-us/cpref/html/frlrfSystemDataSqlClientSqlExceptionMembersTopic.asp.*
- For further guidance about developing custom exceptions, logging and wrapping .NET exceptions, and using various approaches to exception propagation, see *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/exceptdotnet.asp.*

Generating Errors from Stored Procedures

Transact-SQL (T-SQL) provides a **RAISERROR** (note the spelling) function, which you can use to generate custom errors and return them to the client. For ADO.NET clients, the SQL Server .NET Data Provider intercepts these database errors and translates them to **SqlError** objects.

The simplest way to use the **RAISERROR** function is to include the message text as the first parameter, and then specify severity and state parameters, as shown in the following code fragment.

```
RAISERROR( 'Unknown Product ID: %s', 16, 1, @ProductID )
```

In this example, a substitution parameter is used to return the current product ID as part of the error message text. Parameter two is the message severity, and parameter three is the message state.

More Information

- To avoid hard coding message text, you can add your own message to the **sysmessages** table by using the **sp_addmessage** system stored procedure, or by using the SQL Server Enterprise Manager. You can then reference the message by using an ID passed to the **RAISERROR** function. The message IDs that you define must be greater than 50,000, as shown in the following code fragment.

  ```
  RAISERROR( 50001, 16, 1, @ProductID )
  ```

- For full details relating to the **RAISERROR** function, look up **RAISERROR** in the SQL Server Books Online index.

Using Severity Levels Appropriately

Choose your error severity levels carefully and be aware of the impact of each level. Error severity levels range from 0 to 25 and are used to indicate the type of problem that SQL Server 2000 has encountered. In client code, you can obtain an error's severity by examining the **Class** property of the **SqlError** object, within the **Errors** collection of the **SqlException** class. Table 1 indicates the impact and meaning of the various severity levels.

Table 1. Error Severity Levels – Impact and Meaning

Severity level	Connection is closed	Generates SqlException	Meaning
10 and below	No	No	Informational messages that do not necessarily represent error conditions.
11–16	No	Yes	Errors that can be corrected by the user — for example, by retrying the operation with amended input data.
17–19	No	Yes	Resource or system errors.
20–25	Yes	Yes	Fatal system errors (including hardware errors). Client's connection is terminated.

Controlling Automatic Transactions

The SQL Server .NET Data Provider throws a **SqlException** for any error encountered with a severity greater than 10. When a component that is part of an automatic (COM+) transaction detects a **SqlException**, the component must ensure that it votes to abort the transaction. This might or might not be an automatic process, and depends on whether or not the method is marked with the **AutoComplete** attribute.

For more information about handling **SqlException**s in the context of automatic transactions, see the Determining Transaction Outcome section in this document.

Retrieving Informational Messages

Severity levels of 10 and lower are used to represent informational messages and do not cause a **SqlException** to be raised.

To retrieve informational messages:

- Create an event handler and subscribe to the **InfoMessage** event exposed by the **SqlConnection** object. This event's delegate is shown in the following code fragment.

```
public delegate void SqlInfoMessageEventHandler( object sender,
                                                  SqlInfoMessageEventArgs e
);
```

Message data is available through the **SqlInfoMessageEventArgs** object passed to your event handler. This object exposes an **Errors** property, which contains a set of **SqlError** objects—one per informational message. The following code fragment illustrates how to register an event handler that is used to log informational messages.

```
public string GetProductName( int ProductID )
{
  SqlConnection conn = null;
  try
  {
    conn = new SqlConnection(
        "server=(local);Integrated Security=SSPI;database=northwind");
    // Register a message event handler
    conn.InfoMessage += new SqlInfoMessageEventHandler( MessageEventHandler );
    conn.Open();
    // Setup command object and execute it
    . . .
  }
  catch (SqlException sqlex)
  {
    // log and handle exception
    . . .
  }
  finally
  {
    if(conn != null) conn.Close();
  }
}
// message event handler
void MessageEventHandler( object sender, SqlInfoMessageEventArgs e )
{
  foreach( SqlError sqle in e.Errors )
  {
    // Log SqlError properties
    . . .
  }
}
```

Performance

This section introduces a number of common data access scenarios, and for each one, provides details about the most high-performance and scalable solution in terms of ADO.NET data access code. Where appropriate, performance, functionality, and development effort are compared. This section considers the following functional scenarios:

- **Retrieving Multiple Rows**. Retrieving a result set and iterating through the retrieved rows.
- **Retrieving a Single Row**. Retrieving a single row with a specified primary key.
- **Retrieving a Single Item**. Retrieving a single item from a specified row.
- **Determining the Existence of an Item of Data**. Checking to see whether or not a row with a particular primary key exists. This is a variation of the single item lookup scenario in which a simple Boolean return is sufficient.

Retrieving Multiple Rows

In this scenario, you want to retrieve a tabulated set of data and iterate through the retrieved rows to perform an operation. For example, you might want to retrieve a set of data, work with it in disconnected fashion, and pass it to a client application as an XML document (perhaps through a Web service). Alternatively, you might want to display the data in the form of a HTML table.

To help determine the most appropriate data access approach, consider whether you require the added flexibility of the (disconnected) **DataSet** object, or the raw performance offered by the **SqlDataReader** object, which is ideally suited to data presentation in business-to consumer (B2C) Web applications. Figure 4 on the next page shows the two basic scenarios.

Note: The **SqlDataAdapter** used to populate a **DataSet** internally uses a **SqlDataReader** to access the data.

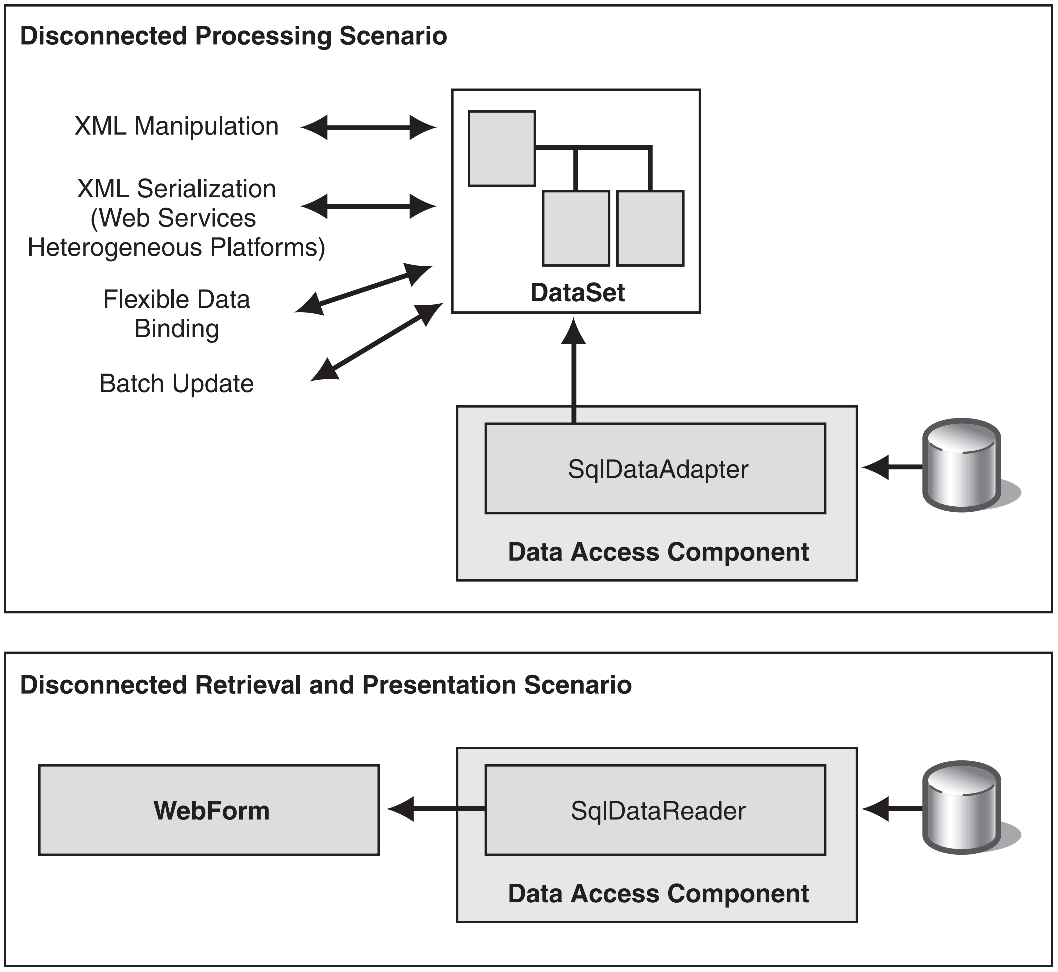
Disconnected Processing Scenario
XML Manipulation
XML Serialization (Web Services Heterogeneous Platforms)
Flexible Data Binding
Batch Update
DataSet
SqlDataAdapter
Data Access Component
Disconnected Retrieval and Presentation Scenario
WebForm
SqlDataReader
Data Access Component

Figure 1.4
Multiple row data access scenarios

Comparing the Options

You have the following options when you retrieve multiple rows from a data source:

- Use a **SqlDataAdapter** object to generate a **DataSet** or **DataTable**.
- Use a **SqlDataReader** to provide a read-only, forward-only data stream.
- Use an **XmlReader** to provide a read-only, forward-only data stream of XML data.

The choice between **SqlDataReader** and **DataSet**/**DataTable** is essentially one of performance versus functionality. The **SqlDataReader** offers optimum performance; the **DataSet** provides additional functionality and flexibility.

Data Binding

All three of these objects can act as data sources for data-bound controls, although the **DataSet** and **DataTable** can act as data sources for a wider variety of controls than the **SqlDataReader**. This is because the **DataSet** and **DataTable** implement **IListSource** (yielding **IList**), whereas the **SqlDataReader** implements **IEnumerable**. A number of WinForm controls capable of data binding require a data source that implements **IList**.

This difference is due to the type of scenario for which each object type is designed. The **DataSet** (which includes the **DataTable**) is a rich, disconnected structure suited to both Web and desktop (WinForm) scenarios. The data reader, on the other hand, is optimized for Web applications that require optimized forward-only data access.

Check the data source requirements for the particular control type that you want to bind to.

Passing Data Between Application Tiers

The **DataSet** provides a relational view of the data that can optionally be manipulated as XML, and allows a disconnected cached copy of the data to be passed between application tiers and components. The **SqlDataReader**, however, offers optimum performance because it avoids the performance and memory overhead associated with the creation of the **DataSet**. Remember that the creation of a **DataSet** object can result in the creation of multiple sub-objects—including **DataTable**, **DataRow**, and **DataColumn** objects—and the collection objects used as containers for these sub-objects.

Using a DataSet

Use a **DataSet** populated by a **SqlDataAdapter** object when:

- You require a disconnected memory-resident cache of data, so that you can pass it to another component or tier within your application.
- You require an in-memory relational view of the data for XML or non-XML manipulation.
- You are working with data retrieved from multiple data sources, such as multiple databases, tables, or files.
- You want to update some or all of the retrieved rows and use the batch update facilities of the **SqlDataAdapter**.
- You want to perform data binding against a control that requires a data source that supports **IList**.

Note: For detailed information, see "Designing Data Tier Components and Passing Data Through Tiers" on the MSDN Web site at *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/BOAGag.asp.*

More Information

If you use a **SqlDataAdapter** to generate a **DataSet** or **DataTable**, note the following:

- You do not need to explicitly open or close the database connection. The **SqlDataAdapter Fill** method opens the database connection and then closes the connection before it returns. If the connection is already open, **Fill** leaves the connection open.
- If you require the connection for other purposes, consider opening it prior to calling the **Fill** method. You can thus avoid unnecessary open/close operations and gain a performance benefit.
- Although you can repeatedly use the same **SqlCommand** object to execute the same command multiple times, do not reuse the same **SqlCommand** object to execute different commands.
- For a code sample that shows how to use a **SqlDataAdapter** to populate a **DataSet** or **DataTable**, see How to Use a **SqlDataAdapter** to Retrieve Multiple Rows in the appendix.

Using a SqlDataReader

Use a **SqlDataReader** obtained by calling the **ExecuteReader** method of the **SqlCommand** object when:

- You are dealing with large volumes of data—too much to maintain in a single cache.
- You want to reduce the memory footprint of your application.
- You want to avoid the object creation overhead associated with the **DataSet**.
- You want to perform data binding with a control that supports a data source that implements **IEnumerable**.
- You wish to streamline and optimize your data access.
- You are reading rows containing binary large object (BLOB) columns. You can use the **SqlDataReader** to pull BLOB data in manageable chunks from the database, instead of pulling all of it at once. For more details about handling BLOB data, see the Handling BLOBs section in this document.

More Information

If you use the **SqlDataReader**, note the following:

- The underlying connection to the database remains open and cannot be used for any other purpose while the data reader is active. Call **Close** on the **SqlDataReader** as soon as possible.
- There can be only one data reader per connection.
- You can close the connection explicitly when you finish with the data reader, or tie the lifetime of the connection to the **SqlDataReader** object, by passing the **CommandBehavior.CloseConnection** enumerated value to the **ExecuteReader** method. This indicates that the connection should be closed when the **SqlDataReader** is closed.
- When accessing data by using the reader, use the typed accessor methods (such as **GetInt32** and **GetString**) if you know the column's underlying data type because they reduce the amount of type conversion required when you read column data.
- To avoid unnecessary data being pulled from server to client, if you want to close the reader and discard any remaining results, call the command object's **Cancel** method before calling **Close** on the reader. **Cancel** ensures that the results are discarded on the server and are not pulled unnecessarily to the client. Conversely, calling **Close** on the data reader causes the reader to unnecessarily pull the remaining results to empty the data stream.
- If you want to obtain output or return values returned from a stored procedure and you are using the **ExecuteReader** method of the **SqlCommand** object, you must call the **Close** method on the reader before the output and return values are available.
- For a code sample that shows how to use a **SqlDataReader**, see How to Use a **SqlDataReader** to Retrieve Multiple Rows in the appendix.

Using an XmlReader

Use an **XmlReader** obtained by calling the **ExecuteXmlReader** method of the **SqlCommand** object when:

- You want to process the retrieved data as XML, but you do not want to incur the performance overhead of creating a **DataSet** and do not require a disconnected cache of data.
- You want to exploit the functionality of the SQL Server 2000 **FOR XML** clause, which allows XML fragments (that is, XML documents with no root element) to be retrieved from the database in a flexible manner. For example, this approach lets you specify precise element names, whether an element or attribute-centric schema should be used, whether a schema should be returned with the XML data and so on.

More Information

If you use the **XmlReader**, note the following:

- The connection must remain open while you read data from the **XmlReader**. The **ExecuteXmlReader** method of the **SqlCommand** object currently does not support the **CommandBehavior.CloseConnection** enumerated value, so you must explicitly close the connection when you finish with the reader.
- For a code sample that shows how to use an **XmlReader**, see How To Use an **XmlReader** to Retrieve Multiple Rows in the appendix.

Retrieving a Single Row

In this scenario, you want to retrieve a single row of data that contains a specified set of columns from a data source. For example, you have a customer ID and want to look up related customer details, or you have a product ID and want to retrieve product information.

Comparing the Options

If you want to perform data binding with a single row retrieved from a data source, you can use a **SqlDataAdapter** to populate a **DataSet** or **DataTable** in the same way that is described in the Multiple Row Retrieval and Iteration scenario discussed previously. However, unless you specifically require **DataSet**/**DataTable** functionality, you should avoid creating these objects.

If you need to retrieve a single row, use one of the following options:

- Use stored procedure output parameters.
- Use a **SqlDataReader** object.

Both options avoid the unnecessary overhead of creating a result set on the server and a **DataSet** on the client. The relative performance of each approach depends on stress levels and whether or not database connection pooling is enabled. When database connection pooling is enabled, performance tests have shown the stored procedure approach to outperform the **SqlDataReader** approach by nearly 30 percent under high-stress conditions (200+ simultaneous connections).

Using Stored Procedure Output Parameters

Use stored procedure output parameters when you want to retrieve a single row from a multitier Web application where you have enabled connection pooling.

More Information

For a code sample that shows how to use stored procedure output parameters, see How To Use Stored Procedure Output Parameters To Retrieve a Single Row in the appendix.

Using a SqlDataReader

Use a **SqlDataReader** when:

- You require metadata in addition to data values. You can use the **GetSchemaTable** method of the data reader to obtain column metadata.
- You are not using connection pooling. With connection pooling disabled, the **SqlDataReader** is a good option under all stress conditions; performance tests have shown it to outperform the stored procedure approach by around 20 percent at 200 browser connections.

More Information

If you use a **SqlDataReader**, note the following:

- If you know your query only returns a single row, use the **CommandBehavior.SingleRow** enumerated value when calling the **ExecuteReader** method of the **SqlCommand** object. Some providers such as the OLE DB .NET Data Provider use this hint to optimize performance. For example, this provider performs binding by using the **IRow** interface (if it is available) rather than the more expensive **IRowset**. This argument has no effect on the SQL Server .NET Data Provider.
- If your SQL Server command contains output parameters or return values, they will not be available until the **DataReader** is closed.
- When using the **SqlDataReader** object, always retrieve output parameters through the typed accessor methods of the **SqlDataReader** object, for example **GetString** and **GetDecimal**. This avoids unnecessary type conversions.
- .NET Framework version 1.1 includes an additional **DataReader** property called **HasRows**, which enables you to determine if the **DataReader** has returned any results before reading from it.
- For a code sample that shows how to use a **SqlDataReader** object to retrieve a single row, see How To use a SqlDataReader to Retrieve a Single Row in the appendix.

Retrieving a Single Item

In this scenario, you want to retrieve a single item of data. For example, you might want to look up a single product name, given its ID, or a single customer credit rating, given the customer's name. In such scenarios, you will generally not want to incur the overhead of a **DataSet** or even a **DataTable** when retrieving a single item.

You might also want simply to check whether a particular row exists in the database. For example, as a new user registers on a Web site, you need to check whether or not the chosen user name already exists. This is a special case of the single item lookup, but in this case, a simple Boolean return is sufficient.

Comparing the Options

Consider the following options when you retrieve a single item of data from a data source:

- Use the **ExecuteScalar** method of a **SqlCommand** object with a stored procedure.
- Use a stored procedure output or return parameter.
- Use a **SqlDataReader** object.

The **ExecuteScalar** method returns the data item directly because it is designed for queries that only return a single value. It requires less code than either the stored procedure output parameter or **SqlDataReader** approaches require..

From a performance perspective, you should use a stored procedure output or return parameter because tests have shown that the stored procedure approach offers consistent performance across low and high-stress conditions (from fewer than 100 simultaneous browser connections to 200 browser connections).

More Information

When retrieving a single item, be aware of the following:

- If a query normally returns multiple columns and/or rows, executing it through **ExecuteQuery** will return only the first column of the first row.
- For a code sample that shows how to use **ExecuteScalar**, see How To Use ExecuteScalar to Retrieve a Single Item in the appendix.
- For a code sample that shows how to use a stored procedure output or return parameter to retrieve a single item, see How To Use a Stored Procedure Output or Return Parameter To Retrieve a Single Item in the appendix.
- For a code sample that shows how to use a **SqlDataReader** object to retrieve a single item, see How To Use a **SqlDataReader** to Retrieve a Single Item in the appendix.

Connecting Through Firewalls

You will often want to configure Internet applications to connect to SQL Server through a firewall. For example, a key architectural component of many Web applications and their firewalls is the perimeter network (also known as DMZ or demilitarized zone), which is used to isolate front-end Web servers from internal networks.

Connecting to SQL Server through a firewall requires specific configuration of the firewall, client, and server. SQL Server provides the Client Network Utility and Server Network Utility programs to aid configuration.

Choosing a Network Library

Use the SQL Server TCP/IP network library to simplify configuration when connecting through a firewall. This is the SQL Server 2000 installation default. If you are using an earlier version of SQL Server, check that you have configured TCP/IP as the default network library on both the client and the server by using the Client Network Utility and Server Network Utility, respectively.

In addition to the configuration benefit, using the TCP/IP library results means that you:

- Benefit from improved performance with high volumes of data and improved scalability.
- Avoid additional security issues associated with named pipes.

You must configure the client and server computers for TCP/IP. Because most firewalls restrict the set of ports through which they allow traffic to flow, you must also give careful consideration to the port numbers that SQL Server uses.

Configuring the Server

Default instances of SQL Server listen on port 1433. UDP port 1434 is also used to allow SQL clients to locate other SQL servers on their network. Named instances of SQL Server 2000, however, dynamically assign a port number when they are first started. Your network administrator will not want to open a range of port numbers on the firewall; therefore, when you use a named instance of SQL Server with a firewall, use the Server Network Utility to configure the instance to listen on a specific port number. Your administrator can then configure the firewall to allow traffic to the specific IP address and port number that the server instance is listening on.

Note: The source port that the client network library uses is dynamically assigned in the range 1024–5000. This is standard practice for TCP/IP client applications, but it means your firewall must allow traffic from any port within this range. For more information about the ports that SQL Server uses, see Microsoft Knowledge Base article 287932, “INF: TCP Ports Needed for Communication to SQL Server Through a Firewall.”

Dynamic Discovery of Named Instances

If you change the default port number that SQL Server listens on, configure your client to connect to this port. For details, see the Configuring the Client section in this document.

If you change the port number for your default instance of SQL Server 2000, failure to modify the client will result in a connection error. If you have multiple SQL Server instances, the latest version of the MDAC data access stack (2.6) employs dynamic discovery and uses a User Datagram Protocol (UDP) negotiation (through UDP port 1434) to locate the named instances. Although this might work in a development environment, it is unlikely to work in a live environment because the firewall will typically block the UDP negotiation traffic.

To circumvent this, always configure your client to connect to the configured destination port number.

Configuring the Client

You should configure the client to use the TCP/IP network library to connect to SQL Server, and you should ensure that the client library uses the correct destination port number.

Using the TCP/IP Network Library

You can configure the client by using the SQL Server Client Network Utility. In some installations, you might not have this utility installed on the client (for example, your Web server). In this case, you can do either of the following:

- Specify the network library by using the "Network Library=dbmssocn" name-value pair supplied through the connection string. The string "dbmssocn" is used to identify the TCP/IP (sockets) library.

  Note: When using the SQL Server .NET Data Provider, the network library setting uses "dbmssocn" by default.

- Modify the registry on the client machine, to establish TCP/IP as the default library. For more information about configuring the SQL Server network library, see HOWTO: Change SQL Server Default Network Library Without Using Client Network Utility (Q250550).

Specifying a Port

If your instance of SQL Server is configured to listen on a port other than the default 1433, you can specify the port number to connect to by:

- Using the Client Network Utility.
- Specifying the port number with the "Server" or "Data Source" name-value pair supplied to the connection string. Use a string with the following format:

  ```
  "Data Source=ServerName,PortNumber"
  ```

  Note: *ServerName* might be an IP address or Domain Name System (DNS) name. For optimum performance, use an IP address to avoid a DNS lookup.

Distributed Transactions

If you have developed serviced components that use COM+ distributed transactions and the services of the Microsoft Distributed Transaction Coordinator (DTC), you might also need to configure your firewall to allow DTC traffic to flow between separate DTC instances, and between the DTC and resource managers such as SQL Server.

For more information about opening ports for the DTC, see INFO: Configuring Microsoft Distributed Transaction Coordinator (DTC) to Work Through a Firewall.

Handling BLOBs

Today, many applications need to deal with data formats such as graphic and sound—or even more elaborate data formats, such as video—in addition to more conventional character and numeric data. There are many different types of graphic, sound, and video formats. From a storage perspective, however, they can all be thought of as lumps of binary data, typically referred to as binary large objects, or BLOBs.

SQL Server provides the **binary**, **varbinary**, and **image** data types to store BLOBs. Despite the name, BLOB data can also refer to text-based data. For example, you might want to store an arbitrarily long notes field that can be associated with a particular row. SQL Server provides the **ntext** and **text** data types for this purpose.

In general, for binary data less than 8 kilobytes (KB), use the **varbinary** data type. For binary data exceeding this size, use **image**. Table 2 highlights the main features of each data type.

Table 2. Data Type Features

Data type	Size	Description
binary	Ranges from 1 to 8,000 bytes. Storage size is specified length plus 4 bytes.	Fixed-length binary data
varbinary	Ranges from 1 to 8,000 bytes. Storage size is actual length of supplied data plus 4 bytes.	Variable-length binary data
image	Variable-length binary data from 0 to 2 gigabytes (GB).	Large-size, variable-length binary data
text	Variable-length data from 0 to 2 GB.	Character data
ntext	Variable-length data from 0 to 2 GB.	Unicode character data

Note: Microsoft® SQL Server™ 2000 supports the ability to store small to medium **text**, **ntext**, and **image** values in a data row. The feature is best used for tables in which the data in **text**, **ntext**, and **image** columns is usually read or written in one unit and most statements referencing the table use the **text**, **ntext**, and **image** data. More information can be found in the SQL Server Books Online under "text in row".

Where to Store BLOB Data

SQL Server 7.0 and later have improved the performance of working with BLOB data stored in the database. One of the reasons for this is that the database page size has increased to 8 KB. As a result, text or image data less than 8 KB no longer needs to be stored in a separate binary tree structure of pages, but can be stored in a single row. This means that reading and writing **text**, **ntext**, or **image** data can be as fast as reading or writing character and binary strings. Beyond 8 KB, a pointer is maintained within the row, and the data itself is held in a binary tree structure of separate data pages—with an inevitable performance impact.

For more information about forcing **text**, **ntext**, and **image** data to be stored in a single row, see the "Using Text and Image Data" topic in SQL Server Books Online.

A commonly used alternative approach to handling BLOB data is to store the BLOB data in the file system, and store a pointer (preferably a Uniform Resource Locator [URL] link) in a database column to reference the appropriate file. For versions earlier than SQL Server 7.0, storing BLOB data externally to the database in the file system can improve performance.

However, improved BLOB support in SQL Server 2000, coupled with ADO.NET support for reading and writing BLOB data, makes storing BLOB data in the database a feasible approach.

Advantages of Storing BLOB Data in the Database

Storing BLOB data in the database offers a number of advantages:

- It is easier to keep the BLOB data synchronized with the remaining items in the row.
- BLOB data is backed up with the database. Having a single storage system can ease administration.
- BLOB data can be accessed through XML support in SQL Server 2000, which can return a base 64–encoded representation of the data in the XML stream.
- SQL Server Full Text Search (FTS) operations can be performed against columns that contain fixed or variable-length character (including Unicode) data. You can also perform FTS operations against formatted text-based data contained within **image** fields—for example, Microsoft Word or Microsoft Excel documents.

Disadvantages of Storing BLOB Data in the Database

Carefully consider what resources might be better stored on the file system rather than in a database. Good examples are images that are typically referenced via HTTP HREF. This is because:

- Retrieving an image from a database incurs significant overhead compared to using the file system.
- Disk storage on database SANs is typically more expensive than storage on disks used in Web server farms.

Note: A well thought out metadata strategy can remove the need for resources such as images, movies, and even Microsoft Office documents to be stored in the database. The metadata could be indexed and include pointers to resources stored on the file system.

Writing BLOB Data to the Database

The following code shows how to use ADO.NET to write binary data obtained from a file to an **image** field in SQL Server.

```
public void StorePicture( string filename )
{
  // Read the file into a byte array
  using(FileStream fs = new FileStream(filename, FileMode.Open, FileAccess.Read))
  {
    byte[] imageData = new Byte[fs.Length];
    fs.Read( imageData, 0, (int)fs.Length );
  }

  using( SqlConnection conn = new SqlConnection(connectionString) )
  {
    SqlCommand cmd = new SqlCommand("StorePicture", conn);
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add("@filename", filename );
    cmd.Parameters["@filename"].Direction = ParameterDirection.Input;
    cmd.Parameters.Add("@blobdata", SqlDbType.Image);
    cmd.Parameters["@blobdata"].Direction = ParameterDirection.Input;
    // Store the byte array within the image field
    cmd.Parameters["@blobdata"].Value = imageData;
    conn.Open();
    cmd.ExecuteNonQuery();
  }
}
```

Reading BLOB Data from the Database

When creating a **SqlDataReader** object through the **ExecuteReader** method to read rows that contain BLOB data, use the **CommandBehavior.SequentialAccess** enumerated value. Without this enumerated value, the reader pulls data from the server to the client one row at a time. If the row contains a BLOB column, this might represent a large amount of memory. By using the enumerated value, you have a finer degree of control because the BLOB data will be pulled only when referenced (for example, by means of the **GetBytes** method, which you can use to control the number of bytes read). This is illustrated in the following code fragment.

```
// Assume previously established command and connection
// The command SELECTs the IMAGE column from the table
conn.Open();
using(SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
{
  reader.Read();
  // Get size of image data - pass null as the byte array parameter
  long bytesize = reader.GetBytes(0, 0, null, 0, 0);
  // Allocate byte array to hold image data
  byte[] imageData = new byte[bytesize];
  long bytesread = 0;
  int curpos = 0;
  while (bytesread < bytesize)
  {
    // chunkSize is an arbitrary application defined value
    bytesread += reader.GetBytes(0, curpos, imageData, curpos, chunkSize);
    curpos += chunkSize;
  }
}
// byte array 'imageData' now contains BLOB from database
```

Note: Using **CommandBehavior.SequentialAccess** requires you to access column data in a strict sequential order. For example, if the BLOB data is in column 3, and you also require data from column 1 and column 2, you must read columns 1 and 2 prior to reading 3.

Performing Database Updates with DataSets

The architecture for performing database updates has changed significantly with the introduction of ADO.NET. ADO.NET has been designed to better facilitate development of multitier applications that scale to large database sizes and large numbers of clients. This has had some important consequences, in particular:

- ADO.NET applications usually segregate application logic on the client from business and data integrity computations on the middle and database tiers. In practical terms, this means that the typical application will have more of a batch or transactional nature with fewer (but larger) interactions between client application and database.
- ADO.NET applications have more control (in comparison to ADO and its predecessors) over exactly *how* updates are processed.

- ADO.NET allows applications to propagate changes by means of stored procedures that are stored in the backend database, instead of directly manipulating a row of a database table. This is a recommended practice.

Update Usage Patterns

The process of using ADO.NET to update data from a **DataSet** can be outlined as follows:

1. Create a **DataAdapter** object and fill a **DataSet** object with the results of a database query. The data will be cached locally.
2. Make changes to the local **DataSet** object. These changes can include updates, deletions, and insertions to one or more tables in the locally cached **DataSet**.
3. Initialize the **DataAdapter** update-related properties. This step configures exactly how updates, deletions, or insertions will be processed. Since there are several ways to handle this, recommended approaches are discussed below in "Initializing DataAdapters for Update."
4. Invoke the **DataAdapter Update** method to submit the pending changes. Each of the changed records of the locally cached **DataSet** will be processed. (Records with no changes will be automatically ignored by the **Update** method.)
5. Handle exceptions thrown by the **DataAdapter Update** method. Exceptions arise when the requested changes cannot be made in the database.

(There is one other way to perform updates. You can directly execute a SQL update query using the **ExecuteNonQuery** method. This technique is appropriate when you want to update specific rows programmatically, without using a **DataSet** object.)

Initializing DataAdapters for Update

In ADO.NET, you must add your own code for submitting database updates to the **DataAdapter** object. There are three ways of doing this:

- You can supply your own updating logic.
- You can use the Data Adapter Configuration Wizard to generate the updating logic.
- You can use the **CommandBuilder** object to generate the updating logic.

It is recommended that you supply your own updating logic. To save time, you can use the Data Adapter Configuration Wizard, but if you do, try not to generate the logic at run time. Do not rely on **CommandBuilder** objects unless you have to because your performance will suffer and you cannot control the updating logic the objects generate. In addition, a **CommandBuilder** will not help you submit updates using stored procedures.

You can use **CommandBuilder** with applications that dynamically generate data access logic, such as reporting or data extraction tools. Using the **CommandBuilder** eliminates the need for these tools to write their own code-generating modules.

Using Stored Procedures

Using stored procedures for updates allows the administrator of your database to implement more granular security than is available with dynamic SQL, as well as more sophisticated data integrity checks. For example, the stored procedure might insert an entry into an audit log as well as perform the requested update. Stored procedures can provide the best performance as well because of offline query optimization performed within the database on stored procedures. Finally, the insulation between the database structure and the application that stored procedures provide allows for easy maintenance.

Because ADO.NET applications that use stored procedures provide many benefits and are no more difficult to implement than those that make changes directly to the database, this approach is recommended in nearly every case. The exception to this is if you must work with multiple back ends or a database, such as Microsoft Access, that doesn't support them. In those cases, use query-based updates.

Managing Concurrency

The **DataSet** object is designed to encourage the use of optimistic concurrency for long-running activities, such as when you are remoting data and when users are interacting with data. When submitting updates from a **DataSet** to the database server, there are four main approaches to managing optimistic concurrency:

- Including only the primary key columns
- Including all columns in the WHERE clause
- Including unique key columns and the timestamp columns
- Including unique key columns and the modified columns

Note that the last three approaches maintain data integrity; the first does not.

Including Only the Primary Key Columns

This option creates a situation where the last update overrides all previous changes. The **CommandBuilder** does not support this option, but the Data Adapter Configuration Wizard does. To use it, go to the **Advanced Options** tab and clear the **Use Concurrency** check box.

This approach is not a recommended practice because it allows users to unknowingly overwrite other users' changes. It is never advisable to compromise the integrity of another user's update. (This technique is appropriate only for single-user databases.)

Including All Columns in the WHERE Clause

This option prevents you from overwriting changes made by other users between the time your code fetches the row and the time your code submits the pending change in the row. This option is the default behavior of both the Data Adapter Configuration Wizard and the SQL code generated by the SqlCommandBuilder.

This approach is not a recommended practice for the following reasons:

- If an additional column is added to the table, the query will need to be modified.
- In general, databases do not let you compare two BLOB values because their large sizes make these comparisons inefficient. (Tools such as the CommandBuilder and the Data Adapter Configuration Wizard should not include BLOB columns in the WHERE clause.)
- Comparing all columns within a table to all the columns in an updated row can create excessive overhead.

Including Unique Key Columns and the Timestamp Columns

With this option, the database updates the timestamp column to a unique value after each update of a row. (You must provide a timestamp column in your table.) Currently, neither the **CommandBuilder** nor the Data Adapter Configuration Wizard supports this option.

Including Unique Key Columns and the Modified Columns

In general, this option is not recommended because errors may result if your application logic relies on out-of-date data fields or even fields that it does not update. For example, if user A changes an order quantity and user B changes the unit price, it may be possible for the order total (quantity multiplied by price) to be incorrectly calculated.

Correctly Updating Null Fields

When fields in a database do not contain data values, it is often convenient to think of these empty fields as containing a special null value. However, this mental picture can be the source of programming errors because database standards require special handling for null values.

The core issue with null fields is that the ordinary SQL **=** operator will always return **false** if one or both of the operands is a null value. The operator **IS NULL** is the only correct way to check for the presence of a null field in a SQL query.

If your application uses the technique described above to manage concurrency by specifying a **WHERE** clause, you must include explicit **IS NULL** expressions wherever it is possible that a field could be null. For example, the following query will always fail if **OldLastName** is null:

```
SET LastName = @NewLastName WHERE StudentID = @StudentID AND
                                  LastName = @OldLastName
```

The query should be rewritten as:

```
SET LastName = @NewLastName WHERE (StudentID = @StudentID) AND
                                  ((LastName = @OldLastName) OR
                                   (OldLastName IS NULL AND LastName IS NULL))
```

A good way to understand how to write the kind of update logic shown above is to read the output generated by the **CommandBuilder** tool.

More Information

A complete treatment of database updates can be found in David Sceppa's *Microsoft ADO.NET* (Microsoft Press, 2002), chapters 11 and 12.

Using Strongly Typed DataSet Objects

Strongly typed **DataSet** objects present database tables and columns as objects and properties. Access is performed by name, not by indexing into a collection. This means you can recognize the difference between strongly typed and untyped **DataSet** objects in the way you access the fields:

```
string n1 = myDataSet.Tables["Students"].Rows[0]["StudentName"];  // untyped
string n2 = myDataSet.Students[0].StudentName;           // strongly typed
```

There are several benefits to using strongly typed **DataSet** objects:

- The code required to access fields is more readable and compact.
- The Intellisense capability within the Visual Studio .NET code editor can automatically complete lines as you type.
- The compiler can catch strongly typed **DataSet** type mismatch errors. It is better to detect type errors at compile time than at run time.

When to Use Strongly Typed DataSets

Strongly typed **DataSet**s are useful because they make application development easier and less error-prone. This is especially true for the client side of a multitiered application where the focus is on graphical user interfaces and data validation that require many field access operations.

However, strongly typed **DataSet**s can be cumbersome if the database structure changes, for example when field and table names are modified. In this case, the typed **DataSet** class has to be regenerated and all dependent classes must be modified.

It is possible to use both strongly typed and untyped approaches in the same application. For example, some developers use strongly typed **DataSet**s on the client side and untyped records on the server. The **.Merge** method of the strongly typed **DataSet** can be used to import data from an untyped **DataSet**.

Generating DataSet Classes

Both the .NET Framework SDK and Visual Studio.NET provide utilities for helping you generate the necessary **DataSet** subclasses. The .NET Framework SDK involves using a command-line tool and writing code. The Visual Studio .NET method, obviously, relies on the Visual Studio .NET development environment and doesn't require you to open a Command window.

Regardless of how the **DataSet** class is generated, the new class must also be deployed to all tiers that reference that typed **DataSet**. (This is not a typical scenario, but it is important to consider if passing a typed **DataSet** across tiers by using remoting.)

Using the .NET Framework Utility

The .NET Framework SDK includes a command-line utility called the XML Schema Definition Tool that helps you generate class files based on XML schema (.xsd) files. Use this utility in conjunction with the **DataSet** object's **WriteXmlSchema** method to translate your untyped **DataSet** into a strongly typed **DataSet**.

The following command illustrates how to generate a class file from an XML schema file. Open a Command window and type:

```
C:\>xsd MyNewClass.xsd /d
```

The first parameter in the command is the path to the XML schema file. The second parameter indicates that the class you want to create is derived from the **DataSet** class. By default, the tool generates Visual C# .NET class files, but it can also generate Visual Basic .NET class files by adding the appropriate options. To list the tool's available options, type:

```
xsd /?
```

Once you've created your new class file, add it to your project. You can now create an instance of your strongly typed **DataSet** class, as shown in the following Visual C# .NET code snippet:

```
MyNewClass ds = new MyNewClass();
```

Using Visual Studio .NET

To generate a strongly typed **DataSet** in Visual Studio .NET, right-click in the form designer window, and then click **Generate Dataset**. This creates an .xsd (XML Schema Definition) file, as well as a class file, and then adds them to your project. Be sure that one or more **DataAdapters** have been added to your Windows form before doing this. Note that the class file is hidden. To see it, click the **Show All Files** button located in the toolbar of the Solution Explorer window. The class file is associated with the .xsd file.

To add relations to a strongly typed **DataSet** , open the XML Schema Designer by double-clicking the schema file in the Solution Explorer window, and then right-click the table to which you want to add a constraint. On the shortcut menu, click **Add New Relation**.

An alternative method for generating a strongly typed **DataSet** in Visual Studio .NET is to right-click the project in the Project Explorer, choose **Add Files**, and then select **dataset**. A new .xsd file will be created. From there you can use Server Explorer to connect to a database and drag table(s) onto the xsd file.

Working with Null Data Fields

Here are a few tips to help you correctly use null field values in the .NET Data Architecture:

- Always set the value of a null field using the **System.DBNull** class. Do not use the null value provided by C# or Visual Basic .NET. For example:

  ```
  rowStudents["Nickname"] = DBNull.Value   // correct!
  ```

- Strongly typed **DataSets** contain two additional methods per **DataRow**—one to check if a column contains a null value, and another to set the column value to null. These are shown in the following code fragment:

  ```
  If (tds.rowStudent[0].IsPhoneNoNull()) {….}
  tds.rowStudent[0].SetPhoneNoNull()
  ```

- Always use the **IsNull** method of the **DataRow** class (or the strongly typed equivalent given in the previous bullet) to test for null values from a database. This method is the only supported way to test for null database values.
- If it is possible that a data field contains a null value, be sure to test for it (with the **IsNull** method) before using the value in a context that expects a non-null value. A typical example of this is **Integer**-valued data fields that may be null. Note that the .NET run time **Integer** datatype does not include the null value. Here is an example:

  ```
  int i = rowStudent["ZipCode"];         // throws exception if null!
  ```

- Use the **nullValue** annotation of a strongly typed **DataSet** .xsd file to configure how null values from a database are mapped. By default, an exception is thrown; however, for more fine-grained control, you can configure the class to replace the null value with a specified value, such as **String.Empty**.

Transactions

Virtually all commercially oriented applications that update data sources require transaction support. Transactions are used to ensure the integrity of a system's state contained within one or more data sources by providing the four basic guarantees of the well known ACID acronym: atomicity, consistency, isolation, and durability.

For example, consider a Web-based retail application that processes purchase orders. Each order requires three distinct operations that involve three database updates:

- The inventory level must be reduced by the quantity ordered.
- The customer's credit level must be debited by the purchase amount.
- A new order must be added to the orders database.

It is essential that these three distinct operations be performed as a unit and in an atomic fashion. They must all succeed, or none of them must succeed—any alternative would compromise data integrity. Transactions provide this and other guarantees.

For further background reading about transaction processing fundamentals, see *http://msdn.microsoft.com/library/en-us/cpguidnf/html /cpcontransactionprocessingfundamentals.asp.*

There are a number of approaches that you can adopt to incorporate transaction management into your data access code. Each approach fits into one of two basic programming models:

- **Manual transactions**. You write code that uses the transaction support features of either ADO.NET or Transact-SQL directly in your component code or stored procedures, respectively.
- **Automatic (COM+) transactions**. You add declarative attributes to your .NET classes that specify the transactional requirements of your objects at run time. This model allows you to easily configure multiple components to perform work within the same transaction.

Both techniques can be used to perform local transactions (that is, transactions performed against a single resource manager such as SQL Server 2000) or distributed transactions (that is, transactions performed against multiple resource managers located on remote computers), although the automatic transaction model greatly simplifies distributed transaction processing.

You might be tempted to use automatic (COM+) transactions to benefit from an easier programming model. This benefit is particularly apparent in systems where you have many components that perform database updates. In many scenarios, however, you should avoid the additional overhead and performance reduction that result from this style of transaction model.

This section provides guidance to help you choose the most appropriate model based on your particular application scenario.

Choosing a Transaction Model

Prior to choosing a transaction model, you should consider whether or not you require transactions at all. Transactions are the single most expensive resource consumed by server applications, and they reduce scalability when used unnecessarily. Consider the following guidelines governing the use of transactions:

- Perform transactions only when you need to acquire locks across a set of operations and need to enforce ACID rules.
- Keep transactions as short as possible to minimize the amount of time that you hold database locks.
- Never place a client in control of transaction lifetime.
- Don't use a transaction for an individual SQL statement. SQL Server automatically runs each statement as an individual transaction.

Automatic vs. Manual Transactions

Although the programming model is somewhat simplified for automatic transactions, particularly when multiple components are performing database updates, manual local transactions are always significantly faster because they do not require interaction with the Microsoft DTC. This is true (although the performance degradation is reduced) even if you are using automatic transactions against a single local resource manager (such as SQL Server), because a manual, local transaction avoids any unnecessary interprocess communication (IPC) with the DTC.

Use manual transactions when:

- You are performing a transaction against a single database.

Use automatic transactions when:

- You require a single transaction to span multiple remote databases.
- You require a single transaction to encompass multiple resource managers—for example, a database and a Windows 2000 Message Queuing (formerly known as MSMQ) resource manager.

Note: Avoid mixing your transaction models. Use one or the other.

In application scenarios in which performance is deemed sufficient, it is reasonable to opt for automatic transactions (even against a single database) to simplify the programming model. Automatic transactions make it easy for multiple components to perform operations that are part of the same transaction.

Using Manual Transactions

With manual transactions, you write code that uses the transaction support features of either ADO.NET or Transact-SQL directly in your component code or stored procedures, respectively. In most cases, you should opt for controlling transactions in your stored procedures because this approach offers superior encapsulation, and from a performance perspective is comparable to performing transactions with ADO.NET code.

Performing Manual Transactions with ADO.NET

ADO.NET supports a transaction object that you can use to begin a new transaction and then explicitly control whether it should be committed or rolled back. The transaction object is associated with a single database connection and is obtained by the **BeginTransaction** method of the connection object. Calling this method does not implicitly mean that subsequent commands are issued in the context of this transaction. You must explicitly associate each command with the transaction, by setting the **Transaction** property of the command. You can associate multiple command objects with the transaction object, thereby grouping multiple operations against the single database in a single transaction.

For an example of using ADO.NET transaction code, see How To Code ADO.NET Manual Transactions in the appendix.

More Information

- The default isolation level for an ADO.NET manual transaction is Read Committed, which means that the database holds shared locks while data is being read, but data can be changed before the end of the transaction. This can potentially result in non-repeatable reads, or phantom data. You can change the isolation level by setting the transaction object's **IsolationLevel** property to one of the enumerated values defined by the **IsolationLevel** enumerated type.
- You must give careful consideration to choosing an appropriate isolation level for your transactions. The tradeoff is one of data consistency versus performance. The highest isolation level (Serialized) offers absolute data consistency, but at the price of overall system throughput. Lower isolation levels can make an application more scalable, but at the same time, increase the possibility of errors resulting from data inconsistency. For systems that read data most of the time, and write data rarely, lower isolation levels might be appropriate.
- For valuable information about choosing an appropriate transaction isolation level, see the Microsoft Press® book *Inside SQL Server 2000*, by Kalen Delaney.

Performing Manual Transactions with Stored Procedures

You can also control manual transactions directly by using Transact-SQL statements in your stored procedures. For example, you could perform transactional operations by using a single stored procedure that employs Transact-SQL transaction statements such as **BEGIN TRANSACTION**, **END TRANSACTION**, and **ROLLBACK TRANSACTION**.

More Information

- If required, you can control the transaction isolation level by using the **SET TRANSACTION ISOLATION LEVEL** statement in the stored procedure. Read Committed is the SQL Server default. For more information about SQL Server isolation levels, see Isolation Levels within the "Accessing and Changing Relation Data" section of SQL Server Books Online.
- For a code sample that shows how to perform transactional updates using Transact-SQL transaction statements, see How To Perform Transactions With Transact-SQL in the appendix.

Using Automatic Transactions

Automatic transactions simplify the programming model because they do not require that you explicitly begin a new transaction or explicitly commit or abort the transaction. However, the most significant advantage of automatic transactions is that they work in conjunction with the DTC, which allows a single transaction to span multiple distributed data sources. In a large-scale distributed application, this advantage can be significant. Although it is possible to manually control distributed transactions by programming the DTC directly, automated transactions drastically simplify the task and are designed for component-based systems. For example, it is easy to declaratively configure multiple components to perform work that comprises a single transaction.

Automatic transactions rely on the distributed transaction support features provided by COM+, and as a result, only serviced components (that is, components that are derived from the **ServicedComponent** class) can use automatic transactions.

To configure a class for automatic transactions:

- Derive the class from the **ServicedComponent** class located within the **System.EnterpriseServices** namespace.
- Define the transaction requirements of the class by using the **Transaction** attribute. The value supplied from the **TransactionOption** enumerated type determines how the class will be configured in the COM+ catalog. Other properties that can be established with this attribute include the transaction isolation level and time-out.

- To avoid explicitly having to vote in the transaction outcome, you can annotate methods with the **AutoComplete** attribute. If these methods throw an exception, the transaction will be aborted automatically. Note that you can still directly vote toward transaction outcome if required. For further details, see the Determining Transaction Outcome section later in this document.

More Information

- For more information about COM+ automatic transactions, search for "Automatic Transactions Through COM+" within the Platform SDK documentation.
- For an example of a transactional .NET class, see How To Code a Transactional .NET Class in the appendix.

Configuring Transaction Isolation Levels

The transaction isolation level for COM+ version 1.0 — that is, COM+ running on Windows 2000—is Serialized. Although this offers the highest degree of isolation, such protection comes at the cost of performance. The overall throughput of your system is reduced because the resource managers (typically databases) involved must hold both read and write locks for the duration of the transaction. During this time, all other transactions are blocked, which can have a significant impact on your application's ability to scale.

COM+ version 1.5, which ships with Microsoft Windows .NET, allows the transaction isolation level to be configured in the COM+ catalog on a per component basis. The setting associated with the root component in the transaction determines the isolation level for the transaction. In addition, internal subcomponents that are part of the same transaction stream must not have a transaction level higher than that defined by the root component. If they do, an error will result when the subcomponent is instantiated.

For .NET managed classes, the **Transaction** attribute supports the public **Isolation** property. You can use this property to declaratively specify a particular isolation level, as illustrated in the following code.

```
[Transaction(TransactionOption.Supported,
Isolation=TransactionIsolationLevel.ReadCommitted)]
public class Account : ServicedComponent
{
  . . .
}
```

More Information

- For more information about configurable transaction isolation levels and other Windows .NET COM+ enhancements, see the MSDN Magazine article, "Windows XP: Make Your Components More Robust with COM+ 1.5 Innovations" at *http://msdn.microsoft.com/msdnmag/issues/01/08/ComXP/default.aspx.*

Determining Transaction Outcome

The outcome of an automatic transaction is governed by the state of the transaction abort flag, together with the consistent flags, in the context of all transactional components in a single transaction stream. Transaction outcome is determined at the point that the root component in the transaction stream is deactivated (and control is returned to the caller). This is illustrated in Figure 5, which shows a classic bank funds transfer transaction.

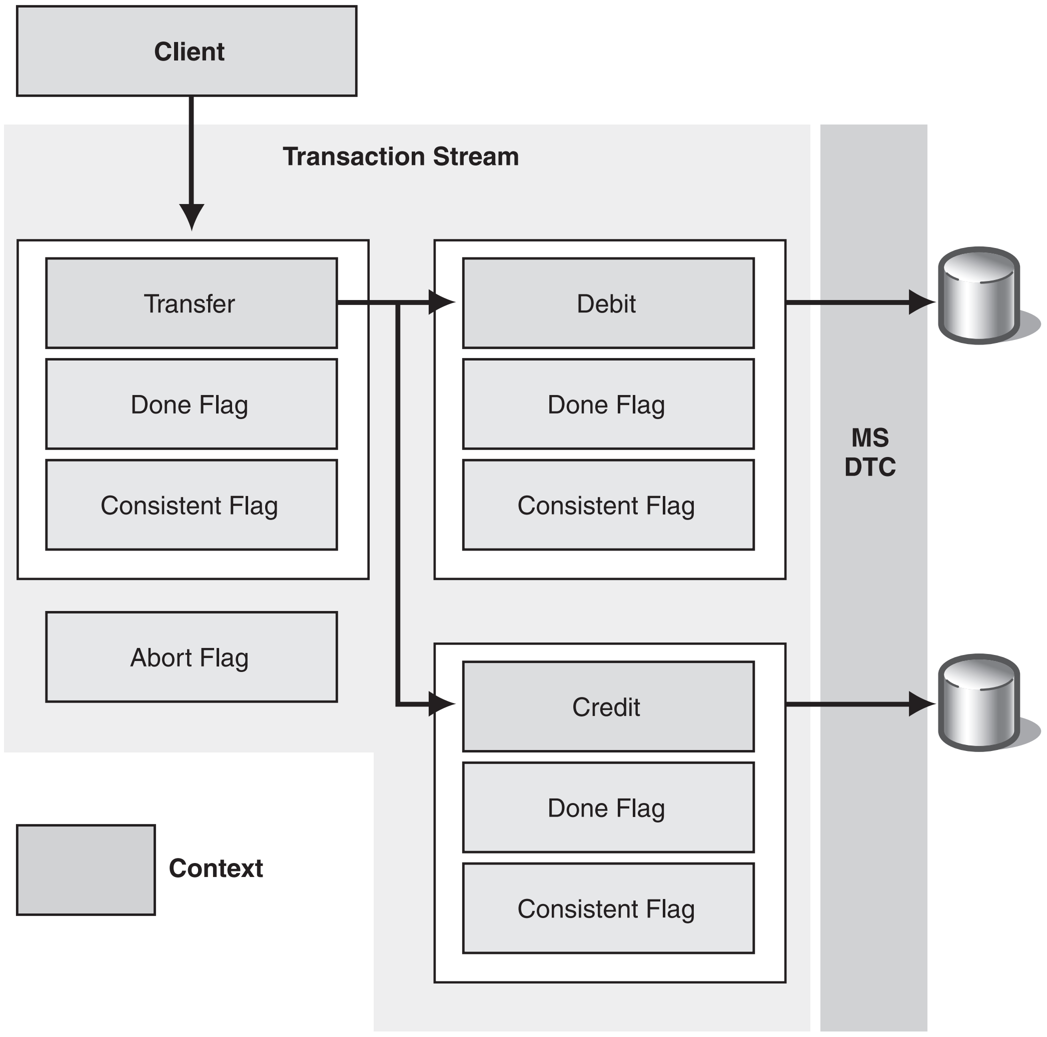

Figure 1.5
Transaction stream and context

The outcome of the transaction is determined when the root object (in this example, the **Transfer** object) is deactivated, and the client's method call returns. If any of the consistent flags within any context are set to false, or if the transaction abort flag is set to true, the underlying physical DTC transaction is aborted.

You can control transaction outcome from a .NET object in one of two ways:

- You can annotate methods with the **AutoComplete** attribute, and let .NET automatically place your vote governing the outcome of the transaction. With this attribute, if the method throws an exception, the consistent flag is automatically set to false (which ultimately causes the transaction to abort). If the method returns without throwing an exception, the consistent flag is set to true, which indicates that the component is happy for the transaction to commit. This is not guaranteed because it depends on the votes of other objects in the same transaction stream.
- You can call the static **SetComplete** or **SetAbort** method of the **ContextUtil** class, which sets the consistent flag to true or false, respectively.

SQL Server errors with a severity greater than 10 result in the managed data provider throwing exceptions of type **SqlException**. If your method catches and handles the exception, be sure to either manually vote to abort the transaction, or for methods flagged as **[AutoComplete]**, ensure that the exception is propagated to the caller.

[AutoComplete] Methods

For methods marked with the **AutoComplete** attribute, do either of the following:

- Propagate the **SqlException** back up the call stack.
- Wrap the **SqlException** in an outer exception and propagate that to the caller. You might want to wrap the exception in an exception type that is more meaningful to the caller.

Failure to propagate the exception will result in the object not voting to abort the transaction, despite the database error. This means that other successful operations made by other objects sharing the same transaction stream might be committed.

The following code catches a **SqlException** and then propagates it directly to the caller. The transaction will ultimately abort because this object's consistent flag will automatically be set to false when it is deactivated.

```
[AutoComplete]
void SomeMethod()
{
  try
  {
    // Open the connection, and perform database operation
    . . .
  }
  catch (SqlException sqlex )
  {
    LogException( sqlex ); // Log the exception details
    throw;                 // Rethrow the exception, causing the consistent
                           // flag to be set to false.
  }
  finally
  {
    // Close the database connection
    . . .
  }
}
```

Non-[AutoComplete] Methods

For methods not marked with the **AutoComplete** attribute, you must:

- Call **ContextUtil.SetAbort** within the **catch** block to vote to abort the transaction. This sets the consistent flag to false.
- Call **ContextUtil.SetComplete** if an exception doesn't occur to vote to commit the transaction. This sets the consistent flag to true (its default state).

The following code illustrates this approach.

```
void SomeOtherMethod()
{
  try
  {
    // Open the connection, and perform database operation
    . . .
    ContextUtil.SetComplete(); // Manually vote to commit the transaction
  }
  catch (SqlException sqlex)
  {
    LogException( sqlex );   // Log the exception details
    ContextUtil.SetAbort();  // Manually vote to abort the transaction
    // Exception is handled at this point and is not propagated to the caller
  }
  finally
  {
    // Close the database connection
    . . .
  }
}
```

Note: If you have multiple **catch** blocks, it is easier to call **ContextUtil.SetAbort** once at the start of a method, and call **ContextUtil.SetComplete** at the end of the **try** block. In this way, you do not need to repeat the call to **ContextUtil.SetAbort** within every **catch** block. The setting of the consistent flag determined by these methods has significance only when the method returns.

You must always propagate exceptions (or wrapped exceptions) back up the call stack because this makes the calling code aware that the transaction will fail. This allows the calling code to make optimizations. For example, in a bank funds transfer scenario, the transfer component could decide not to perform the credit operation if the debit operation has already failed.

If you set the consistent flag to false and then return without throwing an exception, the calling code has no way of knowing that the transaction is bound to fail. Although you could return a Boolean value or set a Boolean output parameter, you should be consistent and throw an exception to indicate an error condition. This leads to cleaner and more consistent code with a standard approach to error handling.

Data Paging

Paging through data is a common requirement in distributed applications. For example, the user might be presented with a list of books where it would be prohibitive to display the entire list at once. The user will want to perform familiar activities on the data, such as viewing the next or previous page of data or jumping to the beginning or end of the list.

This section discusses options for implementing this functionality, and the effect of each option on scalability and performance.

Comparing the Options

The options for data paging are:

- Using the **Fill** method of the **SqlDataAdapter** to fill a **DataSet** with a range of results from a query
- Using ADO through COM interoperability and use a server-side cursor
- Using stored procedures to implement data paging manually

The best option for paging your data is dependent on the factors listed below:

- Scalability requirements
- Performance requirements
- Network bandwidth
- Database server memory and power
- Middle-tier server memory and power
- Number of rows returned by a query you want to page
- Size of your data pages

Performance tests have shown that the manual approach using stored procedures offers the best performance across a wide range of stress levels. However, as the manual approach performs its work on the server, server stress levels can become a significant issue if a large proportion of your site's functionality relies upon data paging functionality. To ensure that this approach suits your particular environment, you should test all options against your specific requirements.

The various options are discussed below.

Using the Fill Method of SqlDataAdapter

As previously discussed, the **SqlDataAdapter** is used to fill a **DataSet** with data from a database. One of the overloaded **Fill** methods (shown in the following code) takes two integer index values.

```
public int Fill(
   DataSet dataSet,
   int startRecord,
   int maxRecords,
   string srcTable
);
```

The **startRecord** value indicates the zero-based index of the start record. The **maxRecords** value indicates the number of records, starting from **startRecord**, to copy into the new **DataSet**.

Internally, the **SqlDataAdapter** uses a **SqlDataReader** to execute the query and return the results. The **SqlDataAdapter** reads the results and creates a **DataSet** based on the data read from the **SqlDataReader**. The **SqlDataAdapter** copies all of the results through **startRecord** and **maxRecords** into a newly generated **DataSet** and discards the results that it doesn't need. This means that a lot of unnecessary data could potentially be pulled across the network to the data access client, which is the primary drawback to this approach.

For example, if you have 1,000 records and want records 900 through 950, the first 899 records are still pulled across the network and discarded on the client side. This overhead would probably be minimal for small result sets, but could be significant when you page through larger sets of data.

Using ADO

Another option for implementing paging is to use COM-based ADO to do the paging. The primary motivation behind this option is to gain access to server-side cursors, which are exposed through the ADO **Recordset** object. You can set the **Recordset** cursor location to **adUseServer**. If your OLE DB provider supports it (SQLOLEDB does), this will result in the use of a server-side cursor. You can then use this cursor to navigate to the starting record directly without having to pull all of the records across the network to the data access client code.

There are two primary drawbacks to this approach:

- In most cases, you will want to translate the records returned in the **Recordset** object into a **DataSet** for use in your client managed code. Although **OleDbDataAdapter** does overload the **Fill** method to take an ADO **Recordset** object and translate that into a **DataSet**, there is no facility to start and end with a particular record. The only realistic option is to move to the start record in the **Recordset** object, loop through each record, and manually copy the data to a new manually generated **DataSet**. It is possible that doing this, particularly through the overhead of COM Interop calls, will more than negate the benefits of not pulling extra data across the network, especially for small **DataSet**(s).
- During the time it takes to pull the data that you want from the server, you hold open a connection and a server-side cursor. Cursors are typically an expensive resource to open and maintain on a database server. Although this option might increase your performance, it is also likely to diminish your scalability by consuming valuable resources on the server for extended periods of time.

Using a Manual Implementation

The final option discussed in this section for paging through your data is to manually implement paging functionality for your application through the use of stored procedures. For tables that contain a unique key, you can implement the stored procedure relatively easily. For tables without a unique key (and you shouldn't have many of those), the process is more complicated.

Paging Against a Table with a Unique Key

If your table contains a unique key, you can use the key in a WHERE clause to create a result set starting from a specific row. This, coupled with the **SET ROWCOUNT** statement or the SQL Server TOP statement used to restrict the size of the result set, provides an effective paging mechanism. This approach is illustrated in the following stored procedure code:

```
CREATE PROCEDURE GetProductsPaged
@lastProductID int,
@pageSize int
AS
SET ROWCOUNT @pageSize
SELECT *
FROM Products
WHERE [standard search criteria]
AND ProductID > @lastProductID
ORDER BY [Criteria that leaves ProductID monotonically increasing]
GO
```

The caller of this stored procedure simply maintains the **lastProductID** value and increments or decrements it by the chosen page size between successive calls.

Paging Against a Table Without a Unique Key

If the table through which you want to page doesn't have a unique key, consider adding one—for example, by using an identity column. This will enable you to implement the paging solution discussed previously.

It is still possible to implement an effective paging solution for a table with no unique key, as long as you can generate uniqueness by combining two or more other fields that are part of the result set.

For example, consider the following table:

Col1	Col2	Col3	Other columns...
A	1	W	...
A	1	X	....
A	1	Y	....
A	1	Z	....
A	2	W	....
A	2	X	....
B	1	W	...
B	1	X	....

With this table, it is possible to generate uniqueness by combining **Col1**, **Col2**, and **Col3**. As a result, you can implement a paging mechanism by using the approach illustrated in the following stored procedure.

```
CREATE PROCEDURE RetrieveDataPaged
@lastKey char(40),
@pageSize int
AS
SET ROWCOUNT @pageSize
SELECT
Col1, Col2, Col3, Col4, Col1+Col2+Col3 As KeyField
FROM SampleTable
WHERE [Standard search criteria]
AND Col1+Col2+Col3 > @lastKey
ORDER BY Col1 ASC, Col2 ASC, Col3 ASC
GO
```

The client maintains the last value of the **KeyField** column returned by the stored procedure and plugs it back in to the stored procedure to control paging through the table.

Although the manual implementation increases the strain placed on the database server, it avoids passing unnecessary data over the network. Performance tests have shown this approach to work well across a range of stress levels. However, depending on how much of the work of your site involves data paging functionality, performing manual paging on the server might impact the scalability of your application. You should run performance tests in your own environment to find the optimum approach for you specific application scenario.

Appendix

How to Enable Object Construction for a .NET Class

You can enable a .NET managed class for object construction by using Enterprise (COM+) Services).

► To enable a .NET managed class

1. Derive your class from the **ServicedComponent** class located in the **System.EnterpriseServices** namespace.

```
using System.EnterpriseServices;
public class DataAccessComponent : ServicedComponent
```

2. Decorate your class with the **ConstructionEnabled** attribute, and optionally specify a default construction string. This default value is held in the COM+ catalog. Administrators can use the Component Services Microsoft Management Console (MMC) snap-in to maintain the value.

```
[ConstructionEnabled(Default="default DSN")]
public class DataAccessComponent : ServicedComponent
```

3. Provide an overridden implementation of the virtual **Construct** method. This method is called after the object's language-specific constructor. The construction string maintained in the COM+ catalog is supplied as the single argument to this method.

```
public override void Construct( string constructString )
{
  // Construct method is called next after constructor.
  // The configured DSN is supplied as the single argument
}
```

4. Provide a strong name for the assembly by signing it through the **AssemblyKey** file or **AssemblyKeyName** attribute. Any assembly registered with COM+ services must have a strong name. For more information about strong-named assemblies, see *http://msdn.microsoft.com/library/en-us/cpguidnf/html /cpconworkingwithstrongly-namedassemblies.asp.*

```
[assembly: AssemblyKeyFile("DataServices.snk")]
```

5. To support dynamic (lazy) registration, use the assembly-level attributes **ApplicationName** and **ApplicationActivation** to specify the name of the COM+ application used to hold assembly components and the application's activation type, respectively. For more information about assembly registration, see *http:// msdn.microsoft.com/library/en-us/cpguidnf/html /cpconregisteringservicedcomponents.asp.*

```
   // the ApplicationName attribute specifies the name of the
   // COM+ Application which will hold assembly components
   [assembly : ApplicationName("DataServices")]

   // the ApplicationActivation.ActivationOption attribute specifies
   // where assembly components are loaded on activation
   // Library : components run in the creator's process
   // Server : components run in a system process, dllhost.exe
   [assembly: ApplicationActivation(ActivationOption.Library)]
```

The following code fragment shows a serviced component called **DataAccessComponent**, which uses COM+ construction strings to obtain a database connection string.

```
using System;
using System.EnterpriseServices;

// the ApplicationName attribute specifies the name of the
// COM+ Application which will hold assembly components
[assembly : ApplicationName("DataServices")]

// the ApplicationActivation.ActivationOption attribute specifies
// where assembly components are loaded on activation
// Library : components run in the creator's process
// Server : components run in a system process, dllhost.exe
[assembly: ApplicationActivation(ActivationOption.Library)]

// Sign the assembly. The snk key file is created using the
// sn.exe utility
[assembly: AssemblyKeyFile("DataServices.snk")]

[ConstructionEnabled(Default="Default DSN")]
public class DataAccessComponent : ServicedComponent
{
    private string connectionString;
    public DataAccessComponent()
    {
      // constructor is called on instance creation
    }
    public override void Construct( string constructString )
    {
      // Construct method is called next after constructor.
      // The configured DSN is supplied as the single argument
      this.connectionString = constructString;
    }
}
```

How to Use a SqlDataAdapter To Retrieve Multiple Rows

The following code illustrates how to use a **SqlDataAdapter** object to issue a command that generates a **DataSet** or **DataTable**. It retrieves a set of product categories from the SQL Server Northwind database.

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

public DataTable RetrieveRowsWithDataTable()
{
  using ( SqlConnection conn = new SqlConnection(connectionString) )
  {
    conn.Open();
    SqlCommand cmd = new SqlCommand("DATRetrieveProducts", conn);
    cmd.CommandType = CommandType.StoredProcedure;
    SqlDataAdapter adapter = new SqlDataAdapter( cmd );
    DataTable dataTable = new DataTable("Products");
    adapter .Fill(dataTable);
    return dataTable;
  }
}
```

► **To use a SqlAdapter to generate a DataSet or DataTable**

1. Create a **SqlCommand** object to invoke the stored procedure and associate this with a **SqlConnection** object (shown) or connection string (not shown).
2. Create a new **SqlDataAdapter** object and associate it with the **SqlCommand** object.
3. Create a **DataTable** (or optionally, a **DataSet**) object. Use a constructor argument to name the **DataTable**.
4. Call the **Fill** method of the **SqlDataAdapter** object to populate either the **DataSet** or **DataTable** with the retrieved rows.

How to Use a SqlDataReader to Retrieve Multiple Rows

The **SqlDataReader** approach to retrieve multiple rows is illustrated in the following code fragment.

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

public SqlDataReader RetrieveRowsWithDataReader()
{
  SqlConnection conn = new SqlConnection(
         "server=(local);Integrated Security=SSPI;database=northwind");
  SqlCommand cmd = new SqlCommand("DATRetrieveProducts", conn );
  cmd.CommandType = CommandType.StoredProcedure;
  try
  {
    conn.Open();
    // Generate the reader. CommandBehavior.CloseConnection causes the
    // the connection to be closed when the reader object is closed
    return( cmd.ExecuteReader( CommandBehavior.CloseConnection ) );
  }
  catch
  {
    conn.Close();
    throw;
  }
}

// Display the product list using the console
private void DisplayProducts()
{
  SqlDataReader reader = RetrieveRowsWithDataReader();
  try
  {
    while (reader.Read())
    {
      Console.WriteLine("{0} {1} {2}",
                        reader.GetInt32(0).ToString(),
                        reader.GetString(1) );
    }
  }
  finally
  {
    reader.Close(); // Also closes the connection due to the
                    // CommandBehavior enum used when generating the reader
  }
}
```

► To retrieve rows with a SqlDataReader

1. Create a **SqlCommand** object used to execute the stored procedure and associate it with a **SqlConnection** object.
2. Open the connection.
3. Generate a **SqlDataReader** object by calling the **ExecuteReader** method of a **SqlCommand** object.
4. To read the data from the stream, call the **Read** method of the **SqlDataReader** object to retrieve rows and use the typed accessor methods (such as the **GetInt32** and **GetString** methods) to retrieve the column values.
5. When you finish with the reader, call its **Close** method.

How to Use an XmlReader to Retrieve Multiple Rows

You can use the **SqlCommand** object to generate an **XmlReader** object, which provides forward-only, stream-based access to XML data. The command (usually a stored procedure) must generate an XML-based result set, which for SQL Server 2000 usually consists of a **SELECT** statement with a valid **FOR XML** clause. The following code fragment illustrates this approach:

```
public void RetrieveAndDisplayRowsWithXmlReader()
{
  using( SqlConnection conn = new SqlConnection(connectionString) )
  {;
    SqlCommand cmd = new SqlCommand("DATRetrieveProductsXML", conn );
    cmd.CommandType = CommandType.StoredProcedure;
try
  {
      conn.Open();
    XmlTextReader xreader = (XmlTextReader)cmd.ExecuteXmlReader();
    while ( xreader.Read() )
    {
      if ( xreader.Name == "PRODUCTS" )
      {
        string strOutput = xreader.GetAttribute("ProductID");
        strOutput += " ";
        strOutput += xreader.GetAttribute("ProductName");
        Console.WriteLine( strOutput );
      }
    }
    xreader.Close();  // XmlTextReader does not support IDisposable so it can't be
                      // used within a using keyword
  }
}
```

The preceding code uses the following stored procedure:

```
CREATE PROCEDURE DATRetrieveProductsXML
AS
SELECT * FROM PRODUCTS
FOR XML AUTO
GO
```

► **To retrieve XML data with an XmlReader**

1. Create a **SqlCommand** object to invoke a stored procedure that generates an XML result set (for example, using the **FOR XML** clause on the **SELECT** statement). Associate the **SqlCommand** object with a connection.
2. Call the **ExecuteXmlReader** method of the **SqlCommand** object and assign the results to a forward-only **XmlTextReader** object. This is the fastest type of

XmlReader object that you should use when you do not require any XML-based validation of the returned data.

3. Read the data by using the **Read** method of the **XmlTextReader** object.

How to Use Stored Procedure Output Parameters to Retrieve a Single Row

You can call a stored procedure that returns the retrieved data items within a single row by means of named output parameters. The following code fragment uses a stored procedure to retrieve the product name and unit price for a specific product contained in the Products table in the Northwind database.

```
void GetProductDetails( int ProductID,
                        out string ProductName, out decimal UnitPrice )
{
  using( SqlConnection conn = new SqlConnection(
        "server=(local);Integrated Security=SSPI;database=Northwind") )
  {
    // Set up the command object used to execute the stored proc
    SqlCommand cmd = new SqlCommand( "DATGetProductDetailsSPOutput", conn )
    cmd.CommandType = CommandType.StoredProcedure;
    // Establish stored proc parameters.
    //  @ProductID int INPUT
    //  @ProductName nvarchar(40) OUTPUT
    //  @UnitPrice money OUTPUT

    // Must explicitly set the direction of output parameters
    SqlParameter paramProdID =
             cmd.Parameters.Add( "@ProductID", ProductID );
    paramProdID.Direction = ParameterDirection.Input;
    SqlParameter paramProdName =
             cmd.Parameters.Add( "@ProductName", SqlDbType.VarChar, 40 );
    paramProdName.Direction = ParameterDirection.Output;
    SqlParameter paramUnitPrice =
             cmd.Parameters.Add( "@UnitPrice", SqlDbType.Money );
    paramUnitPrice.Direction = ParameterDirection.Output;

    conn.Open();
    // Use ExecuteNonQuery to run the command.
    // Although no rows are returned any mapped output parameters
    // (and potentially return values) are populated
    cmd.ExecuteNonQuery( );
    // Return output parameters from stored proc
    ProductName = paramProdName.Value.ToString();
    UnitPrice = (decimal)paramUnitPrice.Value;
  }
}
```

► **To retrieve a single row with stored procedure output parameters**

1. Create a **SqlCommand** object and associate it with a **SqlConnection** object.
2. Set up the stored procedure parameters by calling the **Add** method of the **SqlCommand's Parameters** collection. By default, parameters are assumed to be input parameters, so you must explicitly set the direction of any output parameters.

   Note: It is good practice to explicitly set the direction of all parameters, including input parameters.

3. Open the connection.
4. Call the **ExecuteNonQuery** method of the **SqlCommand** object. This populates the output parameters (and potentially a return value).
5. Retrieve the output parameters from the appropriate **SqlParameter** objects by using the **Value** property.
6. Close the connection.

The preceding code fragment invokes the following stored procedure.

```
CREATE PROCEDURE DATGetProductDetailsSPOutput
@ProductID int,
@ProductName nvarchar(40) OUTPUT,
@UnitPrice money OUTPUT
AS
SELECT @ProductName = ProductName,
       @UnitPrice = UnitPrice
FROM Products
WHERE ProductID = @ProductID
GO
```

How to Use a SqlDataReader to Retrieve a Single Row

You can use a **SqlDataReader** object to retrieve a single row, and specifically the desired column values from the returned data stream. This is illustrated in the following code fragment.

```
void GetProductDetailsUsingReader( int ProductID,
                        out string ProductName, out decimal UnitPrice )
{
  using( SqlConnection conn = new SqlConnection(
         "server=(local);Integrated Security=SSPI;database=Northwind") )
  {
    // Set up the command object used to execute the stored proc
    SqlCommand cmd = new SqlCommand( "DATGetProductDetailsReader", conn );
    cmd.CommandType = CommandType.StoredProcedure;
    // Establish stored proc parameters.
    //  @ProductID int INPUT
    SqlParameter paramProdID = cmd.Parameters.Add( "@ProductID", ProductID );
    paramProdID.Direction = ParameterDirection.Input;
    conn.Open();
    using( SqlDataReader reader = cmd.ExecuteReader() )
    {
      if( reader.Read() ) // Advance to the one and only row
      {
        // Return output parameters from returned data stream
        ProductName = reader.GetString(0);
        UnitPrice = reader.GetDecimal(1);
       }
    }
  }
}
```

► **To return a single row with a SqlDataReader object**

1. Establish the **SqlCommand** object.
2. Open the connection.
3. Call the **ExecuteReader** method of the **SqlDataReader** object.
4. Retrieve output parameters through the typed accessor methods of the **SqlDataReader** object—in this case, **GetString** and **GetDecimal**.

The preceding code fragment invokes the following stored procedure.

```
CREATE PROCEDURE DATGetProductDetailsReader
@ProductID int
AS
SELECT ProductName, UnitPrice FROM Products
WHERE ProductID = @ProductID
GO
```

How to Use ExecuteScalar to Retrieve a Single Item

The **ExecuteScalar** method is designed for queries that return only a single value. In the event of the query returning multiple columns and/or rows, **ExecuteScalar** returns only the first column of the first row.

The following code shows how to look up the product name for a specific product ID:

```
void GetProductNameExecuteScalar( int ProductID, out string ProductName )
{
  using( SqlConnection conn = new SqlConnection(
         "server=(local);Integrated Security=SSPI;database=northwind") )
  {
    SqlCommand cmd = new SqlCommand("LookupProductNameScalar", conn );
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.Parameters.Add("@ProductID", ProductID );
    conn.Open();
    ProductName = (string)cmd.ExecuteScalar();
  }
}
```

► **To use ExecuteScalar to retrieve a single item**

1. Establish a **SqlCommand** object to call a stored procedure.
2. Open the connection.
3. Call the **ExecuteScalar** method. Notice that this method returns an object type. This contains the value of the first column retrieved and must be cast to the appropriate type.
4. Close the connection.

The preceding code uses the following stored procedure:

```
CREATE PROCEDURE LookupProductNameScalar
@ProductID int
AS
SELECT TOP 1 ProductName
FROM Products
WHERE ProductID = @ProductID
GO
```

How to Use a Stored Procedure Output or Return Parameter to Retrieve a Single Item

You can look up a single value by using a stored procedure output or return parameter. Use of an output parameter is illustrated in the following code:

```
void GetProductNameUsingSPOutput( int ProductID, out string ProductName )
{
  using( SqlConnection conn = new SqlConnection(
        "server=(local);Integrated Security=SSPI;database=northwind") )
  {
    SqlCommand cmd = new SqlCommand("LookupProductNameSPOutput", conn );
    cmd.CommandType = CommandType.StoredProcedure;

    SqlParameter paramProdID = cmd.Parameters.Add("@ProductID", ProductID );
    ParamProdID.Direction = ParameterDirection.Input;
    SqlParameter paramPN =
           cmd.Parameters.Add("@ProductName", SqlDbType.VarChar, 40 );
    paramPN.Direction = ParameterDirection.Output;

    conn.Open();
    cmd.ExecuteNonQuery();
    ProductName = paramPN.Value.ToString();
  }
}
```

► **To retrieve a single value with a stored procedure output parameter**

1. Establish a **SqlCommand** object to call the stored procedure.
2. Set up any input parameters and the single output parameter by adding **SqlParameters** to the **SqlCommand's Parameters** collection.
3. Open the connection.
4. Call the **ExecuteNonQuery** method of the **SqlCommand** object.
5. Close the connection.
6. Retrieve the output value by using the **Value** property of the output **SqlParameter**.

The preceding code uses the following stored procedure.

```
CREATE PROCEDURE LookupProductNameSPOutput
@ProductID int,
@ProductName nvarchar(40) OUTPUT
AS
SELECT @ProductName = ProductName
FROM Products
WHERE ProductID = @ProductID
GO
```

The following code illustrates the use of a return value to indicate whether a particular row exists. From a coding perspective, this is similar to using stored procedure output parameters, except that you must explicitly set the **SqlParameter** direction to **ParameterDirection.ReturnValue**.

```
bool CheckProduct( int ProductID )
{
  using( SqlConnection conn = new SqlConnection(
       "server=(local);Integrated Security=SSPI;database=northwind") )
  {
    SqlCommand cmd = new SqlCommand("CheckProductSP", conn );
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add("@ProductID", ProductID );
    SqlParameter paramRet =
           cmd.Parameters.Add("@ProductExists", SqlDbType.Int );
    paramRet.Direction = ParameterDirection.ReturnValue;
    conn.Open();
    cmd.ExecuteNonQuery();
  }
  return (int)paramRet.Value == 1;
}
```

► **To check whether a particular row exists by using a stored procedure return value**

1. Establish a **SqlCommand** object to call the stored procedure.
2. Set up an input parameter that contains the primary key value of the row to be accessed.
3. Set up the single return value parameter. Add a **SqlParameter** object to the **SqlCommand's Parameters** collection, and set its direction to **ParameterDirection.ReturnValue**.
4. Open the connection.
5. Call the **ExecuteNonQuery** method of the **SqlCommand** object.
6. Close the connection.
7. Retrieve the return value by using the **Value** property of the return value **SqlParameter**.

The preceding code uses the following stored procedure.

```
CREATE PROCEDURE CheckProductSP
@ProductID int
AS
IF EXISTS( SELECT ProductID
           FROM Products
           WHERE ProductID = @ProductID )
  return 1
ELSE
  return 0
GO
```

How to Use a SqlDataReader to Retrieve a Single Item

You can use a **SqlDataReader** object to obtain a single output value by calling the **ExecuteReader** method of the command object. This requires slightly more code because the **SqlDataReader Read** method must be called, and then the desired value is retrieved through one of the reader's accessor methods. Use of a **SqlDataReader** object is illustrated in the following code.

```
bool CheckProductWithReader( int ProductID )
{
  using( SqlConnection conn = new SqlConnection(
         "server=(local);Integrated Security=SSPI;database=northwind") )
  {
    SqlCommand cmd = new SqlCommand("CheckProductExistsWithCount", conn );
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.Add("@ProductID", ProductID );
    cmd.Parameters["@ProductID"].Direction = ParameterDirection.Input;
    conn.Open();
    using( SqlDataReader reader = cmd.ExecuteReader(
                                CommandBehavior.SingleResult ) )
    {
    if( reader.Read() )
    {
      return (reader.GetInt32(0) > 0);
    }
    return false;
  }
}
```

The preceding code assumes the following stored procedure.

```
CREATE PROCEDURE CheckProductExistsWithCount
@ProductID int
AS
SELECT COUNT(*) FROM Products
WHERE ProductID = @ProductID
GO
```

How to Code ADO.NET Manual Transactions

The following code shows how to take advantage of the transaction support offered by the SQL Server .NET Data Provider to protect a funds transfer operation with a transaction. This operation transfers money between two accounts located in the same database.

```
public void TransferMoney( string toAccount, string fromAccount, decimal amount )
{
  using ( SqlConnection conn = new SqlConnection(
            "server=(local);Integrated Security=SSPI;database=SimpleBank" ) )
  {
    SqlCommand cmdCredit = new SqlCommand("Credit", conn );
    cmdCredit.CommandType = CommandType.StoredProcedure;
    cmdCredit.Parameters.Add( new SqlParameter("@AccountNo", toAccount) );
    cmdCredit.Parameters.Add( new SqlParameter("@Amount", amount ));

    SqlCommand cmdDebit = new SqlCommand("Debit", conn );
    cmdDebit.CommandType = CommandType.StoredProcedure;
    cmdDebit.Parameters.Add( new SqlParameter("@AccountNo", fromAccount) );
    cmdDebit.Parameters.Add( new SqlParameter("@Amount", amount ));

    conn.Open();
    // Start a new transaction
    using ( SqlTransaction trans = conn.BeginTransaction() )
    {
      // Associate the two command objects with the same transaction
      cmdCredit.Transaction = trans;
      cmdDebit.Transaction = trans;
      try
      {
        cmdCredit.ExecuteNonQuery();
        cmdDebit.ExecuteNonQuery();
        // Both commands (credit and debit) were successful
        trans.Commit();
      }
      catch( Exception ex )
      {
        // transaction failed
        trans.Rollback();
        // log exception details . . .
        throw ex;
      }
    }
  }
}
```

How to Perform Transactions with Transact-SQL

The following stored procedure illustrates how to perform a transactional funds transfer operation within a Transact-SQL stored procedure.

```
CREATE PROCEDURE MoneyTransfer
@FromAccount char(20),
@ToAccount char(20),
@Amount money
AS
BEGIN TRANSACTION
– PERFORM DEBIT OPERATION
UPDATE Accounts
SET Balance = Balance - @Amount
WHERE AccountNumber = @FromAccount
IF @@RowCount = 0
BEGIN
  RAISERROR('Invalid From Account Number', 11, 1)
  GOTO ABORT
END
DECLARE @Balance money
SELECT @Balance = Balance FROM ACCOUNTS
WHERE AccountNumber = @FromAccount
IF @BALANCE < 0
BEGIN
  RAISERROR('Insufficient funds', 11, 1)
  GOTO ABORT
END
– PERFORM CREDIT OPERATION
UPDATE Accounts
SET Balance = Balance + @Amount
WHERE AccountNumber = @ToAccount
IF @@RowCount = 0
BEGIN
  RAISERROR('Invalid To Account Number', 11, 1)
  GOTO ABORT
END
COMMIT TRANSACTION
```

```
RETURN 0
ABORT:
  ROLLBACK TRANSACTION
GO
```

This stored procedure uses the **BEGIN TRANSACTION**, **COMMIT TRANSACTION**, and **ROLLBACK TRANSACTION** statements to manually control the transaction.

How to Code a Transactional .NET Class

The following example code shows three serviced .NET managed classes that are configured for automatic transactions. Each class is annotated with the **Transaction** attribute, whose value determines whether or not a new transaction stream should be started or whether the object should share the stream of its immediate caller. These components work together to perform a bank money transfer. The **Transfer** class is configured with the **RequiresNew** transaction attribute, and the **Debit** and **Credit** classes are configured with **Required**. As a result, at run time all three objects will share the same transaction.

```
using System;
using System.EnterpriseServices;

[Transaction(TransactionOption.RequiresNew)]
public class Transfer : ServicedComponent
{
  [AutoComplete]
  public void Transfer( string toAccount,
                        string fromAccount, decimal amount )
  {
    try
    {
      // Perform the debit operation
      Debit debit = new Debit();
      debit.DebitAccount( fromAccount, amount );
      // Perform the credit operation
      Credit credit = new Credit();
      credit.CreditAccount( toAccount, amount );
    }
    catch( SqlException sqlex )
    {
      // Handle and log exception details
      // Wrap and propagate the exception
      throw new TransferException( "Transfer Failure", sqlex );
    }
  }
}
[Transaction(TransactionOption.Required)]
public class Credit : ServicedComponent
{
  [AutoComplete]
  public void CreditAccount( string account, decimal amount )
  {
    try
    {
      using( SqlConnection conn = new SqlConnection(
              "Server=(local); Integrated Security=SSPI"; database="SimpleBank") )
      {
        SqlCommand cmd = new SqlCommand("Credit", conn );
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.Add( new SqlParameter("@AccountNo", account) );
        cmd.Parameters.Add( new SqlParameter("@Amount", amount ));
        conn.Open();
        cmd.ExecuteNonQuery();
      }
    }
  }catch( SqlException sqlex ){
     // Log exception details here
     throw; // Propagate exception
  }
}
[Transaction(TransactionOption.Required)]
public class Debit : ServicedComponent
{
  public void DebitAccount( string account, decimal amount )
  {
    try
    {
      using( SqlConnection conn = new SqlConnection(
              "Server=(local); Integrated Security=SSPI"; database="SimpleBank") )
      {
        SqlCommand cmd = new SqlCommand("Debit", conn );
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.Add( new SqlParameter("@AccountNo", account) );
        cmd.Parameters.Add( new SqlParameter("@Amount", amount ));
        conn.Open();
        cmd.ExecuteNonQuery();
      }
    }
    catch (SqlException sqlex)
    {
      // Log exception details here
      throw; // Propagate exception back to caller
    }
  }
}
```

Authors

Alex Mackman, Chris Brooks, Steve Busby, Ed Jezierski, Jason Hogg, Roberta Leibovitz (Modeled Computation) and Colin Campbell (Modeled Computation)

Collaborators

Many thanks to the following contributors and reviewers:

Bill Vaughn, Mike Pizzo, Pablo Castro, Doug Rothaus, Kevin White, Blaine Dokter, David Schleifer, Graeme Malcolm (Content Master), Bernard Chen (Sapient), Matt Drucker (Turner Broadcasting), Steve Kirk, David Sceppa, Scott Densmore, Diego González (Lagash Systems)

To learn more about .NET best practices, please visit the patterns & practices Web page.

To participate in an online collaborative development environment on this topic, join the GotDotNet workspace: Microsoft Patterns & Practices Data Access for .NET Workspace, at *http://www.gotdotnet.com/Community/Workspaces/workspace.aspx?id=c20d12b0-af52-402b-9b7c-aaeb21d1f431*. Please share your Data Access Block questions, suggestions, and customizations with the community in this workspace

Questions? Comments? Suggestions? For feedback on the content of this article, please e-mail us at devfdbck@microsoft.com.