

ADO.NET Fundamentals

Overview

A large number of computer applications—both desktop and web applications—are *data-driven*. These applications are largely concerned with retrieving, displaying, and modifying data. Retrieving and processing data seems like a fairly straightforward task, but over the past decade the way applications use data has changed repeatedly. Developers have moved from simple client applications that use local databases to distributed systems that rely on centralized databases on dedicated servers. At the same time, data access technologies have evolved. If you have worked with Microsoft languages for some time, you have most likely heard of (and possibly used) an alphabet soup of data access technologies that includes ODBC, DAO, RDO, RDS, and ADO. The .NET Framework includes its own data access technology, ADO.NET. ADO.NET consists of managed classes that allow .NET applications to connect to data sources (usually relational databases), execute commands, and manage disconnected data. The small miracle of ADO.NET is that it allows you to write more or less the same data access code in web applications that you write for client-server desktop applications, or even single-user applications that connect to a local database. This Supplementary Material describes the architecture of ADO.NET and the ADO.NET data providers. You will learn about ADO.NET basics such as opening a connection, executing a SQL statement or stored procedure, and retrieving the results of a query. You will also learn how to prevent SQL injection attacks and use transactions.

ADO.NET CHANGES IN .NET 2.0

If you are a seasoned .NET 1.x programmer, you are probably wondering what is new in the latest iteration of ADO.NET.

Without a doubt, the greatest change for ASP.NET applications is the new data binding model. The data binding model allows you to reduce the amount of code you write for data display, and it can even allow you avoid writing any data access code at

all (if you're willing to pay the price with pages that are less flexible and more difficult to optimize).

Even with the advent of the new data binding model, the underlying ADO.NET reality does not change that much.

Many of the changes are internal (such as a more compact DataSet serialization format that requires less memory) or involve frills that aren't of much use in the average web application (such as the new SQL bulk copy feature for rapidly transferring an entire table between two database servers).

Here are some genuinely interesting ADO.NET changes that are still around:

- **Provider factories:** The dream of generic data access code (code you can write once and use with multiple different databases) takes a giant leap forward in .NET 2.0 thanks to *provider factories*—new components that can create strongly typed Connection, Command, and DataAdapter objects on the fly.
- **Change notification:** To build truly scalable web applications, you need to cache data that is retrieved from a database so it can be reused without connecting to the data source each time. However, caching introduces the possibility of out-of-date information. ADO.NET includes a new change notification feature that you can use to automatically remove cached data when the related records in the database change.
- **Connection statistics:** It's a small frill, but the new connection-tracking features of the SqlConnection object might help you profile different data access strategies.
- **SQL Server 2005:** SQL Server 2005 introduces a whole set of new features, and ADO.NET 2.0 supports them seamlessly. These features include user-defined data types that are based on .NET classes, as well as stored procedures written with .NET languages.

The ADO.NET Architecture

ADO.NET uses a multilayered architecture that revolves around a few key concepts, such as Connection, Command, and DataSet objects.

ADO.NET Data Providers

A *data provider* is a set of ADO.NET classes that allows you to access a specific database, execute SQL commands, and retrieve data. Essentially, a data provider is a bridge between your application and a data source.

The classes that make up a data provider include the following:

- **Connection:** You use this object to establish a connection to a data source.
- **Command:** You use this object to execute SQL commands and stored procedures.
- **DataReader:** This object provides fast read-only, forward-only access to the data retrieved from a query.
- **DataAdapter:** This object performs two tasks. First, you can use it to fill a DataSet (a disconnected collection of tables and relationships) with information extracted from a data source. Second, you can use it to apply changes to a data source, according to the modifications you have made in a DataSet.

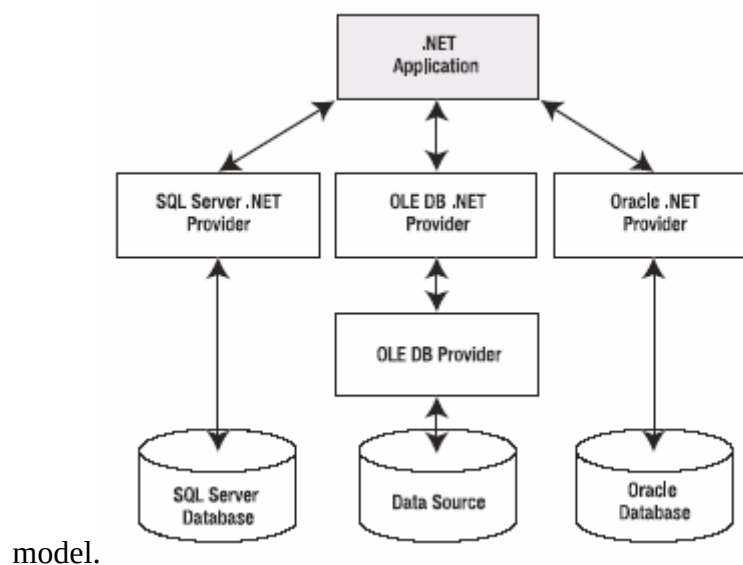
ADO.NET does not include generic data provider objects. Instead, it includes different data providers specifically designed for different types of data sources. Each data provider has a specific implementation of the Connection, Command, DataReader, and DataAdapter classes that is optimized for a specific RDBMS (relational database management system). For example, if you need to create a connection to a SQL Server database, you will use a connection class named SqlConnection.

One of the key underlying ideas of the ADO.NET provider model is that it is *extensible*. In other words, developers can create their own providers for proprietary data sources. In fact, numerous proof-of-concepts examples are available that show how you can easily create custom ADO.NET providers to wrap nonrelational data stores, such as the file system or a directory service. Some third-party vendors also sell custom providers for .NET.

The .NET Framework is bundled with a small set of four providers:

- **SQL Server provider:** Provides optimized access to a SQL Server database (version 7.0 or later).
- **OLE DB provider:** Provides access to any data source that has an OLE DB driver. This includes SQL Server databases prior to version 7.0.
- **Oracle provider:** Provides optimized access to an Oracle database (version 8i or later).
- **ODBC provider:** Provides access to any data source that has an ODBC driver.

Figure 1 shows the layers of the ADO.NET provider



When choosing a provider, you should first try to find a native .NET provider that's customized for your data source. If you cannot find a native provider, you can use the OLE DB provider, as long as you have an OLE DB driver for your data source. The OLE DB technology has been around for many years as part of ADO, so most data sources provide an OLE DB driver (including SQL Server, Oracle, Access, MySQL, and many more). In the rare situation when you cannot find a dedicated .NET provider or an OLE DB driver, you can fall back on the ODBC provider, which works in conjunction with an ODBC driver.

However, if you can find a provider that's customized specifically for your data source, you should use it instead.

For example, you can connect to a SQL Server database using either the SQL Server provider or the OLE DB provider, but the SQL Server provider will always perform best.

Standardization in ADO.NET

At first glance, it might seem that ADO.NET offers a fragmented model, because it does not include a generic set of objects that can work with multiple types of databases. As a result, if you change from one RDBMS to another, you will need to modify your data access code to use a different set of classes. But even though different .NET data providers use different classes, all providers are *standardized* in the same way. More specifically, each provider is based on the same set of interfaces and base classes. For example, every Connection object implements the IDbConnection interface, which defines core methods such as Open() and Close(). This standardization guarantees that every Connection class will work in the same way and expose the same set of core properties and methods. Behind the scenes, different providers use completely different low-level calls and APIs. For example, the SQL Server provider uses the proprietary TDS (Tabular Data Stream) protocol to communicate with the server. The benefits of this model are not immediately obvious, but they are significant:

- Because each provider uses the same interfaces and base classes, you can still write generic data access code (with a little more effort) by coding against the interfaces instead of the provider classes.
- Because each provider is implemented separately, it can use proprietary optimizations.

(This is different from the ADO model, where every database call needs to filter through a common layer before it reaches the underlying database driver.) In addition, custom providers can add nonstandard features that aren't included in other providers (such as SQL Server's ability to perform an XML query).

ADO.NET also has another layer of standardization: the DataSet. The DataSet is an all-purpose container for data that you have retrieved from one or more tables in a

data source. The DataSet is completely generic—in other words, custom providers do not define their own custom versions of the DataSet class. No matter which data provider you use, you can extract your data and place it into a disconnected DataSet in the same way. That makes it easy to separate data *retrieval* code from data *processing* code. If you change the underlying database, you will need to change the data retrieval code, but if you use the DataSet and your information has the same structure, you will not need to modify the way you process that data.

SQL Server 2005

ADO.NET 2.0 provides support for a few features that are limited to SQL Server 2005. These features include the following:

MARS (multiple active result sets): This allows you to have more than one query on the go at the same time. For example, you could query a list of customers and then query a list of orders without closing the first query. This technique is occasionally useful, but it is better if you can avoid the extra overhead.

User-defined data types: Using .NET code, you can define a custom class and then store instances of that class directly in a column of the database. This saves you the work of examining several fields in a row and then manually creating a corresponding data object to use in your application.

Managed stored procedures: SQL Server 2005 can host the CLR, which gives you the ability to write stored procedures in the database using pure C# code.

SQL notifications: Notifications allow your code to respond when specific changes are made in a database. In ASP.NET, this feature is most commonly used to invalidate a cached data object when one or more records are updated. This is the only SQL Server 2005 feature that is also supported in SQL Server 7 and SQL Server 2000, albeit through a different mechanism.

Snapshot transaction isolation: This is a new transaction level that allows you to improve concurrency. It allows transactions to see a slightly older version of data while it is being updated by another transaction.

Fundamental ADO.NET Classes

ADO.NET has two types of objects: *connection-based* and *content-based*.

Connection-based objects: These are the data provider objects such as Connection, Command, DataAdapter, and DataReader. They execute SQL statements, connect to a database, or fill a DataSet. The connection-based objects are specific to the type of data source.

Content-based objects: These objects are really just “packages” for data. They include the DataSet, DataColumn, DataRow, DataRelation, and several others. They are completely independent of the type of data source and are found in the System.Data namespace.

The ADO.NET classes are grouped into several namespaces. Each provider has its own namespace, and generic classes such as the DataSet are stored in the System.Data namespaces. Table 1 describes the namespaces.

Table 1. *The ADO.NET Namespaces*

<i>Namespace</i>	<i>Description</i>
System.Data	Contains the key data container classes that model columns, relations, tables, datasets, rows, views, and constraints. In addition, contains the key interfaces that are implemented by the connection-based data objects.
System.Data.Common	Contains base, mostly abstract classes that implement some of the interfaces from System.Data and define the core ADO.NET functionality. Data providers inherit from these classes to create their own specialized versions.

System.Data.OleDb	Contains the classes used to connect to an OLE DB provider, including OleDbCommand, OleDbConnection, and OleDbDataAdapter. These classes support most OLE DB providers but not those that require OLE DB version 2.5 interfaces.
System.Data.SqlClient	Contains the classes you use to connect to a Microsoft SQL Server database, including SqlCommand, SqlConnection, and SqlDataAdapter. These classes are optimized to use the TDS interface to SQL Server.
System.Data.OracleClient	Contains the classes required to connect to an Oracle database (version 8.1.7 or later), including OracleCommand, OracleConnection, and OracleDataAdapter. These classes are using the optimized Oracle Call Interface (OCI).
System.Data.Odbc	Contains the classes required to connect to most ODBC drivers. These classes include OdbcCommand, OdbcConnection, and OdbcDataAdapter. ODBC drivers are included for all kinds of data sources and are configured through the Data Sources icon in the Control Panel.
System.Data.SqlTypes	

	Contains structures that match the native data types in SQL Server. These classes aren't required but provide an alternative to using standard .NET data types, which require automatic conversion
--	--

The Connection Class

The Connection class allows you to establish a connection to the data source that you want to interact with. Before you can do anything else (including retrieving, deleting, inserting, or updating data), you need to establish a connection.

The core Connection properties and methods are specified by the IDbConnection interface, which all Connection classes implement.

Connection Strings

When you create a Connection object, you need to supply a *connection string*. The connection string is a series of name/value settings separated by semicolons (;). The order of these settings is unimportant, as is the capitalization. Taken together, they specify the basic information needed to create a connection.

Although connection strings vary based on the RDBMS and provider you are using, a few pieces of information are almost always required:

- **The server where the database is located:** In the examples in this material, the database server is always located on the same computer as the ASP.NET application, so the loopback alias localhost is used instead of a computer name.
- **The database you want to use:** Most of the examples in this material use the Northwind database, which is installed by default with most editions of SQL Server.
- **How the database should authenticate you:** The Oracle and SQL Server providers give you the choice of supplying authentication credentials or logging in as the current user. The latter choice is usually best, because you do not need to place password information in your code or configuration files.

For example, here's the connection string you would use to connect to the Northwind database on the current computer using integrated security (which uses the currently logged-in Windows user to access the database):

```
string connectionString = "Data Source=localhost;Initial Catalog=Northwind;" +  
"Integrated Security=SSPI";
```

If integrated security is not supported, the connection must indicate a valid user and password combination. For a newly installed SQL Server database, the sa (system administrator) account is usually present. Here's a connection string that uses this account:

```
string connectionString = "Data Source=localhost;Initial Catalog=Northwind;" +  
"user id=sa;password=opensesame";
```

If you're using the OLE DB provider, your connection string will still be similar, with the addition of a provider setting that identifies the OLE DB driver. For example, you can use the following connection string to connect to an Oracle database through the MSDAORA OLE DB provider:

```
string connectionString = "Data Source=localhost;Initial Catalog=Sales;" +  
"user id=sa;password=;Provider=MSDAORA";
```

And here's an example that connects to an Access database file:

```
string connectionString = "Provider=Microsoft.Jet.OLEDB.4.0;" +
@"Data Source=C:\DataSources\Northwind.mdb";
```

There's no reason to hard-code a connection string. The <connectionStrings> section of the web.config file is a handy place to store your connection strings. Here's an example:

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

<configuration>
  <connectionStrings>
    <add name="Northwind"
          connectionString="Data Source=.\sqlexpress;Initial
                           Catalog=Northwind;Integrated Security=True"/>
  </connectionStrings>
  ...
</configuration>
```

You can then retrieve your connection string by name from the WebConfigurationManager.ConnectionStrings collection, like so:

```
string connectionString = WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
```

Testing a Connection

Once you've chosen your connection string, managing the connection is easy—you simply use the Open() and Close() methods. You can use the following code in the Page.Load event handler to test a connection and write its status to a label (as shown in Figure 2):

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Import Namespace="System.Web.Configuration" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
  protected void Page_Load(object sender, EventArgs e)
  {
```

```

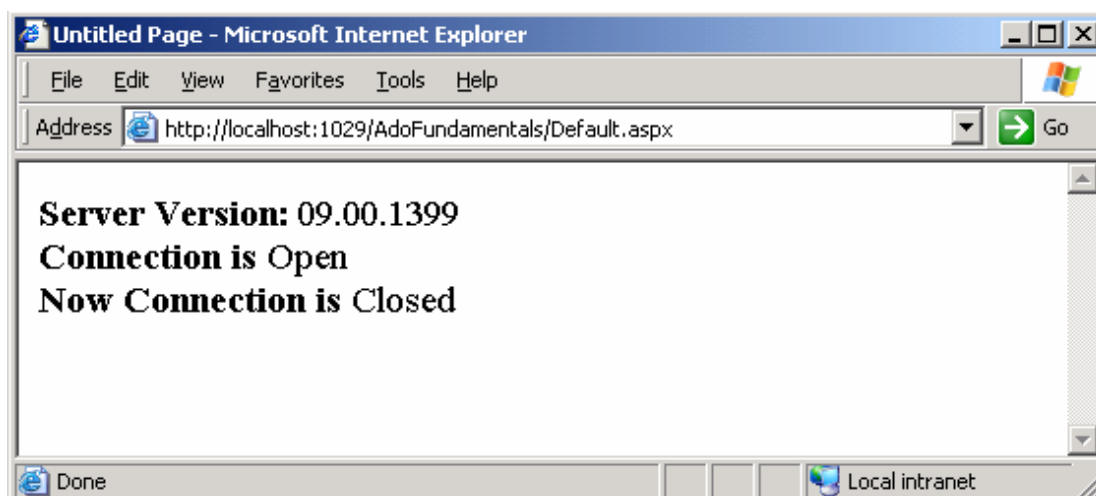
string connectionString =
WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
SqlConnection connection = new SqlConnection(connectionString);
try
{
    //Try to open the connection
    connection.Open();
    Response.Write("<b>Server Version: </b> "
        + connection.ServerVersion + "<br/>");
    Response.Write("<b>Connection is </b>"
        + connection.State.ToString() + "<br/>");
}
catch (Exception exp)
{
    Response.Write("Error Reading the database<br/>");
    Response.Write(exp.Message);
}
finally
{
    // Either way, make sure the connection is properly closed.
    // Even if the connection wasn't opened successfully,
    // calling Close() won't cause an error.
    connection.Close();
    Response.Write("<b>Now Connection is </b>" + connection.State.ToString());
}
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Connection Try Catch Finally</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>

        </div>
    </form>
</body>
</html>

```

Figure 2 shows the results of running this code



Connections are a limited server resource. This means it is imperative that you open the connection as late as possible and release it as quickly as possible. In the previous code sample, an exception handler is used to make sure that even if an unhandled error occurs, the connection will be closed in the finally block. If you do not use this design and an unhandled exception occurs, the connection will remain open until the garbage collector disposes of the SqlConnection object. An alternate approach is to wrap your data access code in a using block. The using statement declares that you are using a disposable object for a short period of time. As soon as the using block ends, the CLR releases the corresponding object immediately by calling its Dispose() method. Interestingly, calling the Dispose() method of a Connection object is equivalent to calling Close().

That means you can rewrite the earlier example in the following, more compact, form:

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Import Namespace="System.Web.Configuration" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        string connectionString =
            WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection connection = new SqlConnection(connectionString);
        using(connection)
        {
            try
            {
                //Try to open the connection
                connection.Open();
                Response.Write("<b>Server Version: </b> "
                    + connection.ServerVersion + "<br/>");
                Response.Write("<b>Connection is </b>"
                    + connection.State.ToString() + "<br/>");
            }
            catch (Exception exp)
            {
                Response.Write("Error Reading the database<br/>");
                Response.Write(exp.Message);
            }
        }
        // We do not need any more to call the Close method since the connection
        //object is now is disposable
        //connection.Close();
        Response.Write("<b>Now Connection is </b>" + connection.State.ToString());
    }
</script>
```

```
<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
  <title>Connection Disposal Using Block</title>
</head>
<body>
  <form id="form1" runat="server">
    <div>

    </div>
  </form>
</body>
</html>
```

The best part is that you do not need to write a finally block—the using statement releases the object you are using even if you exit the block as the result of an unhandled exception.

Connection Pooling

Acquiring a connection takes a short, but definite, amount of time. In a web application in which requests are being handled efficiently, connections will be opened and closed endlessly as new requests are processed. In this environment, the small overhead required to establish a connection can become significant and limit the scalability of the system.

One solution is *connection pooling*. Connection pooling is the practice of keeping a permanent set of open database connections to be shared by sessions that use the same data source. This avoids the need to create and destroy connections all the time. Connection pools in ADO.NET are completely transparent to the programmer, and your data access code does not need to be altered. When a client requests a connection by calling Open(), it is served directly from the available pool, rather than re-created. When a client releases a connection by calling Close() or Dispose(), it's not discarded but returned to the pool to serve the next request.

ADO.NET does not include a connection pooling mechanism. However, most ADO.NET providers implement some form of connection pooling. The SQL Server and Oracle data providers implement their own efficient connection pooling algorithms. For a connection to be reused with SQL Server or Oracle, the connection

string matches exactly. If it differs even slightly, a new connection will be created in a new pool. SQL Server and Oracle connection pooling use a full-text match algorithm. That means any minor change in the connection string will thwart connection pooling, even if the change is simply to reverse the order of parameters or add an extra blank space at the end. For this reason, it's imperative that you don't hard-code the connection string in different web pages. Instead, you should store the connection string in one place—preferably the <connectionStrings> section of the web.config file.

With both the SQL Server and Oracle providers, connection pooling is enabled and used automatically. However, you can also use connection string parameters to configure pool size settings.

Table 2 describes these parameters.

Table 2. Connection Pooling Settings

<i>Setting</i>	<i>Description</i>
Max Pool Size	The maximum number of connections allowed in the pool (defaults to 100). If the maximum pool size has been reached, any further attempts to open a connection are queued until a connection becomes available. (An error is raised if the Connection.Timeout value elapses before a connection becomes available.)
Min Pool Size	The minimum number of connections always retained in the pool (defaults to 0). This number of connections will be created when the first connection is opened, leading to a minor delay for the first request.

Pooling	When true (the default), the connection is drawn from the appropriate pool or, if necessary, is created and added to the appropriate pool.
Connection Lifetime	Specifies a time interval in seconds. If a connection is returned to the pool and its creation time is older than the specified lifetime, it will be destroyed. The default is 0, which disables this behavior. This feature is useful when you want to recycle a large number of connections at once.

Here's an example connection string that sets a minimum pool size:

```
string connectionString = "Data Source=localhost;Initial Catalog=Northwind;"
                          + "Integrated Security=SSPI;Min Pool Size=10";
SqlConnection con = new SqlConnection(connectionString);
// Get the connection from the pool (if it exists)
// or create the pool with 10 connections (if it doesn't).
con.Open();
// Return the connection to the pool.
con.Close();
//Remove The Connection Pool
SqlConnection.ClearPool(con);
```

Some providers include methods for emptying out the connection pool. For example, with the SqlConnection you can call the static ClearPool() and ClearAllPools() methods. When calling ClearPool(), you supply a SqlConnection, and all the matching connections are removed. ClearAllPools() empties out every connection pool in the current application domain. (Technically, these methods don't close the connections. They just mark them as invalid so that they will time out and be closed during the regular connection cleanup a few minutes later.) This functionality is

rarely used—typically, the only case it is useful is if you know the pool is full of invalid connections (for example, as a result of restarting SQL Server) and you want to avoid an error.

Connection Statistics

If you're using the SQL Server provider, you can retrieve some interesting statistics using the `SqlConnection.RetrieveStatistics()` method (new in .NET 2.0). `RetrieveStatistics` returns a hashtable with various low-level details that can help you analyze the performance of commands and the amount of work you've performed. Connection statistics are not often used in a deployed application, but they are useful for diagnosing performance during the testing and profiling stage. For example, they provide one tool that you can use to determine how various data access strategies perform (other tools include the SQL Server administrative utilities, such as the SQL Profiler and Query Analyzer).

By default, connection statistics are disabled to improve performance. To use connection statistics, you need to set the `SqlConnection.StatisticsEnabled` property to true. This tells the `SqlConnection` class to collect information about every action it performs. At any point after, you can call the `RetrieveStatistics()` method to examine this information, or you can use `ResetStatistics()` to clear it out and start from scratch.

Here's an example that displays the number of bytes received by the connection since you enabled statistics:

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<%@ Import Namespace="System.Web.Configuration" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        string connectionString =
            WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection connection = new SqlConnection(connectionString);
        using(connection)
        {
            try
            {
                //Try to open the connection
                connection.StatisticsEnabled = true;
```

```

connection.Open();
SqlCommand cmd = new SqlCommand();
cmd.CommandType = System.Data.CommandType.Text;
cmd.CommandText = "Select * from Employees";
cmd.Connection = connection;
SqlDataReader reader = cmd.ExecuteReader();
while (reader.Read())
{
    //We have just read all the rows in the Employees table
    //without any processing.
    IDictionary hash = connection.RetrieveStatistics();
    foreach(DictionaryEntry entry in hash)
    {
        Response.Write( entry.Key.ToString() + " = " +
            entry.Value.ToString() + "<br>");
    }
}
catch (Exception exp)
{
    Response.Write("Error Reading the database<br>");
    Response.Write(exp.Message);
}
}
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Connection Disposabel Using Block</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            </div>
        </form>
    </body>
</html>

```

Statistics are provided in a loosely typed name/value collection. That means you need to know the specific name of a statistic in order to retrieve it. Here are a few of the most useful:

- **ServerRoundtrips:** Indicates the number of times the connection has made a request to the database server. Typically, this value corresponds to the number of commands you've executed, but strategies such as command batching can affect it.
- **ConnectionTime:** Indicates the cumulative amount of time the connection has been open.
- **BytesReceived:** Indicates the total number of bytes retrieved from the database server (as a cumulative result of all the commands you have executed).
- **SumResultSets:** Indicates the number of queries you have performed.

- **SelectRows:** Records the total number of rows retrieved in every query you have executed.

The Command and DataReader Classes

The Command class allows you to execute any type of SQL statement. Although you can use a Command class to perform *data-definition* tasks (such as creating and altering databases, tables, and indexes), you are much more likely to perform *data-manipulation* tasks (such as retrieving and updating the records in a table).

The provider-specific Command classes implement standard functionality, just like the Connection classes. In this case, the IDbCommand interface defines the core set of Command methods that are used to execute a command over an open connection.

Command Basics

Before you can use a command, you need to choose the command type, set the command text, and bind the command to a connection. You can perform this work by setting the corresponding properties (CommandType, CommandText, and Connection), or you can pass the information you need as constructor arguments.

The command text can be a SQL statement, a stored procedure, or the name of a table. It all depends on the *type* of command you are using. Three types of commands exist, as listed in Table 3.

Table 3. *Values for the CommandType Enumeration*

Value	Description
CommandType.Text	The command will execute a direct SQL statement. The SQL statement is provided in the CommandText property. This is the default value.

CommandType.StoredProcedure	The command will execute a stored procedure in the data source. The CommandText property provides the name of the stored procedure.
CommandType.TableDirect	The command will query all the records in the table. The CommandText is the name of the table from which the command will retrieve the records. (This option is included for backward compatibility with certain OLE DB drivers only. It is not supported by the SQL Server data provider, and it won't perform as well as a carefully targeted query.)

For example, here's how you would create a Command object that represents a query:

```
SqlCommand cmd = new SqlCommand();
cmd.CommandType = System.Data.CommandType.Text;
cmd.CommandText = "Select * from Employees";
cmd.Connection = connection;
```

And here's a more efficient way using one of the Command constructors. Note that you don't need to specify the CommandType, because CommandType.Text is the default.

```
SqlCommand cmd = new SqlCommand("SELECT * FROM Employees", connection);
```

Alternatively, to use a stored procedure, you would use code like this:

```
SqlCommand cmd = new SqlCommand("GetEmployees", connection);
cmd.CommandType = CommandType.StoredProcedure;
```

These examples simply define a Command object; they do not actually execute it. The Command object provides **three** methods that you can use to perform the command, depending on whether you want to retrieve a full result set, retrieve a single value, or just execute a nonquery command. Table 4 lists these methods.

Table 4. Command Methods

<i>Method</i>	<i>Description</i>
ExecuteNonQuery()	Executes non-SELECT commands, such as SQL commands that insert, delete, or update records. The returned value indicates the number of rows affected by the command.
ExecuteScalar()	Executes a SELECT query and returns the value of the first field of the first row from the rowset generated by the command. This method is usually used when executing an aggregate SELECT command that uses functions such as COUNT() or SUM() to calculate a single value.
ExecuteReader()	Executes a SELECT query and returns a DataReader object that wraps a read-only, forward-only cursor.

The DataReader Class

A DataReader allows you to read the data returned by a SELECT command one record at a time, in a forward-only, read-only stream. This is sometimes called a *firehose cursor*. Using a DataReader is the simplest way to get to your data, but it lacks the sorting and relational abilities of the disconnected DataSet. However, the DataReader provides the quickest possible no-nonsense access to data.

Table 5 lists the core methods of the DataReader.

Table 5. DataReader Methods

<i>Method</i>	<i>Description</i>
Read()	Advances the row cursor to the next row in the stream. This method must also be called before reading the first row of data. (When the DataReader is first created, the row cursor is positioned just before the first row.) The Read() method returns true if there's another row to be read, or false if it's on the last row.
GetValue()	Returns the value stored in the field with the specified column name or index, within the currently selected row. The type of the returned value is the closest .NET match to the native value stored in the data source. If you access the field by index and inadvertently pass an invalid index that refers to a nonexistent field, you will get an IndexOutOfRangeException exception. You can also access the same value by name, which is slightly less efficient because the DataReader must perform a lookup to find the column with the specified name.
GetValues()	

	Saves the values of the current row into an array. The number of fields that are saved depends on the size of the array you pass to this method. You can use the <code>DataReader.FieldCount</code> property to determine the number of fields in a row, and you can use that information to create an array of the right size if you want to save all the fields.
<code>GetInt32(),GetChar()</code>	These methods return the value of the field with the specified index <code>GetDateTime()</code>, <code>GetXxx()</code> in the current row, with the data type specified in the method name. Note that if you try to assign the returned value to a variable of the wrong type, you'll get an <code>InvalidCastException</code> exception.
<code>NextResult()</code>	If the command that generated the <code>DataReader</code> returned more than one rowset, this method moves the pointer to the next rowset (just before the first row).
<code>Close()</code>	Closes the reader. If the originator command ran a stored procedure that returned an output value, that value can be read only from the respective parameter after the reader has been closed.

The ExecuteReader() Method and the DataReader

The following example creates a simple query command to return all the records from the Employees table in the Northwind database. The command is created when the page is loaded.

```
<%@ Page Language="C#" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<%@ Import Namespace="System.Web.Configuration" %>
<%@ Import Namespace="System.Data.SqlClient" %>

<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        string connectionString =
            WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection con = new SqlConnection(connectionString);
        string sql = "Select * from Employees";
        SqlCommand cmd = new SqlCommand(sql, con);
        con.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        StringBuilder sb = new StringBuilder();
        while (reader.Read())
        {
            sb.Append("<li>");
            sb.Append(reader["TitleOfCourtesy"]);
            sb.Append("<b>");
            sb.Append(reader.GetString(1));
            sb.Append("</b>, ");
            sb.Append(reader.GetString(2));
            sb.Append(" - Employee from ");
            sb.Append(reader.GetDateTime(6).ToString("d"));
            sb.Append("</li>");
        }
        reader.Close();
        con.Close();
        Message.Text = sb.ToString();
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml" >
<head runat="server">
    <title>Execute Reader Test</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label runat="server" ID="Message" />
        </div>
    </form>
</body>
</html>
```

The connection is then opened, and the command is executed through the ExecuteReader() method, which returns a SqlDataReader.

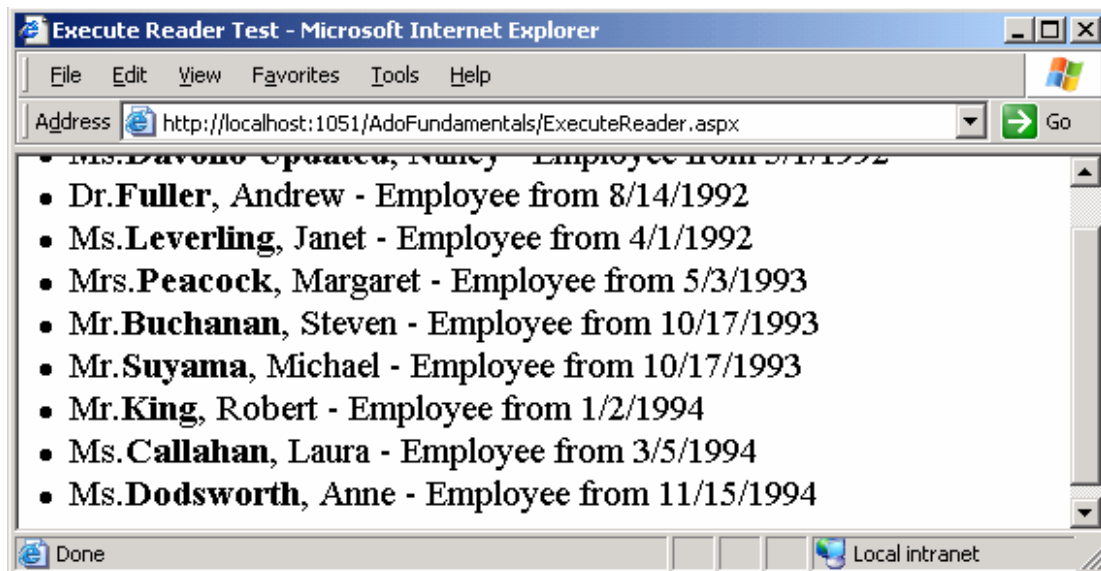
Once you have the DataReader, you can cycle through its records by calling the Read() method in a while loop. This moves the row cursor to the next record (which, for the first call, means to the first record). The Read() method also returns a Boolean value indicating whether there are more rows to read. In the following example the loop continues until Read() returns false, at which point the loop ends gracefully.

The information for each record is then joined into a single large string. To ensure that these string manipulations performed quickly, a StringBuilder (from the System.Text namespace) is used instead of ordinary string objects.

This code reads the value of the TitleOfCourtesy field by accessing the field by name through the Item indexer. Because the Item property is the default indexer, you don't need to explicitly include the Item property name when you retrieve a field value. Next, the code reads the LastName and FirstName fields by calling GetString() with the field index (1 and 2 in this case). Finally, the code accesses the HireDate field by calling GetDateTime() with a field index of 6. All these approaches are equivalent and included to show the supported variation.

The final step is to close the reader and the connection and show the generated text in a server control.

If you run the page, you'll see the output shown in Figure 3.



CommandBehavior

The `ExecuteReader()` method has an overloaded version that takes one of the values from the `CommandBehavior` enumeration as a parameter. One useful value is `CommandBehavior.CloseConnection`. When you pass this value to the `ExecuteReader()` method, the `DataReader` will close the associated connection as soon as you close the `DataReader`.

Using this technique, you could rewrite the code as follows:

```
SqlDataReader reader =  
    cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

This behavior is particularly useful if you retrieve a `DataReader` in one method and need to pass it to another method to process it. If you use the `CommandBehavior.CloseConnection` value, the connection will be automatically closed as soon as the second method closes the reader. Another possible value is `CommandBehavior.SingleRow`, which can improve the performance of the query execution when you're retrieving only a single row. For example, if you are retrieving a single record using its unique primary key field (`CustomerID`, `ProductID`, and so on), you can use this optimization. You can also use `CommandBehavior.SequentialAccess` to read part of a binary field at a time, which reduces the memory overhead for large binary fields.

Processing Multiple Result Sets

The command you execute does not have to return a single result set. Instead, it can execute more than one query and return more than one result set as part of the same command. This is useful if you need to retrieve a large amount of related data, such as a list of products and product categories that, taken together, represent a product catalog.

A command can return more than one result set in two ways:

- If you are calling a stored procedure, it may use multiple `SELECT` statements.

- If you are using a straight text command, you may be able to batch multiple commands by separating commands with a semicolon (;). Not all providers support this technique, but the SQL Server database provider does.

Here is an example of a string that defines a batch of three SELECT statements:

```
string sql = "Select top 5 * from employees;"
            + "Select top 5 * from Customers;"
            + "Select top 5 * from suppliers";
```

This string contains three queries. Together, they return the first five records from the Employees table, the first five from the Customers table, and the first five from the Suppliers table. Processing these results is fairly straightforward. Initially, the DataReader will provide access to the results from the Employees table. Once you've finished using the Read() method to read all these records, you can call NextResult() to the next result set. When there are no more result sets, this method returns false. You can even cycle through all the available result sets with a while loop, although in this case you must be careful not to call NextResult() until you finish reading the first result set.

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<%@ Import Namespace="System.Web.Configuration" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">
    protected void Page_Load(object sender, EventArgs e)
    {
        string strCon =
WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;

        SqlConnection con = new SqlConnection(strCon);

        string sql = "Select top 5 FirstName, LastName from employees;"
                    + "Select top 5 ContactName,ContactTitle from Customers;"
                    + "Select top 5 SupplierID,CompanyName,ContactName from
                        suppliers";

        SqlCommand cmd = new SqlCommand(sql, con);

        con.Open();
        SqlDataReader reader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

```

StringBuilder sb = new StringBuilder();
int i = 0;
do
{
    i++;
    sb.Append("<h2> Rowset " + i.ToString() + "</h2>");
    while (reader.Read())
    {
        sb.Append("<li>");
        for (int field = 0; field < reader.FieldCount; field++)
        {
            sb.Append(reader.GetName(field));
            sb.Append(" : ");
            sb.Append(reader.GetValue(field).ToString());
            sb.Append(" -- ");
        }
        sb.Append("</li>");
    }
    sb.Append("<br/>");
} while (reader.NextResult());

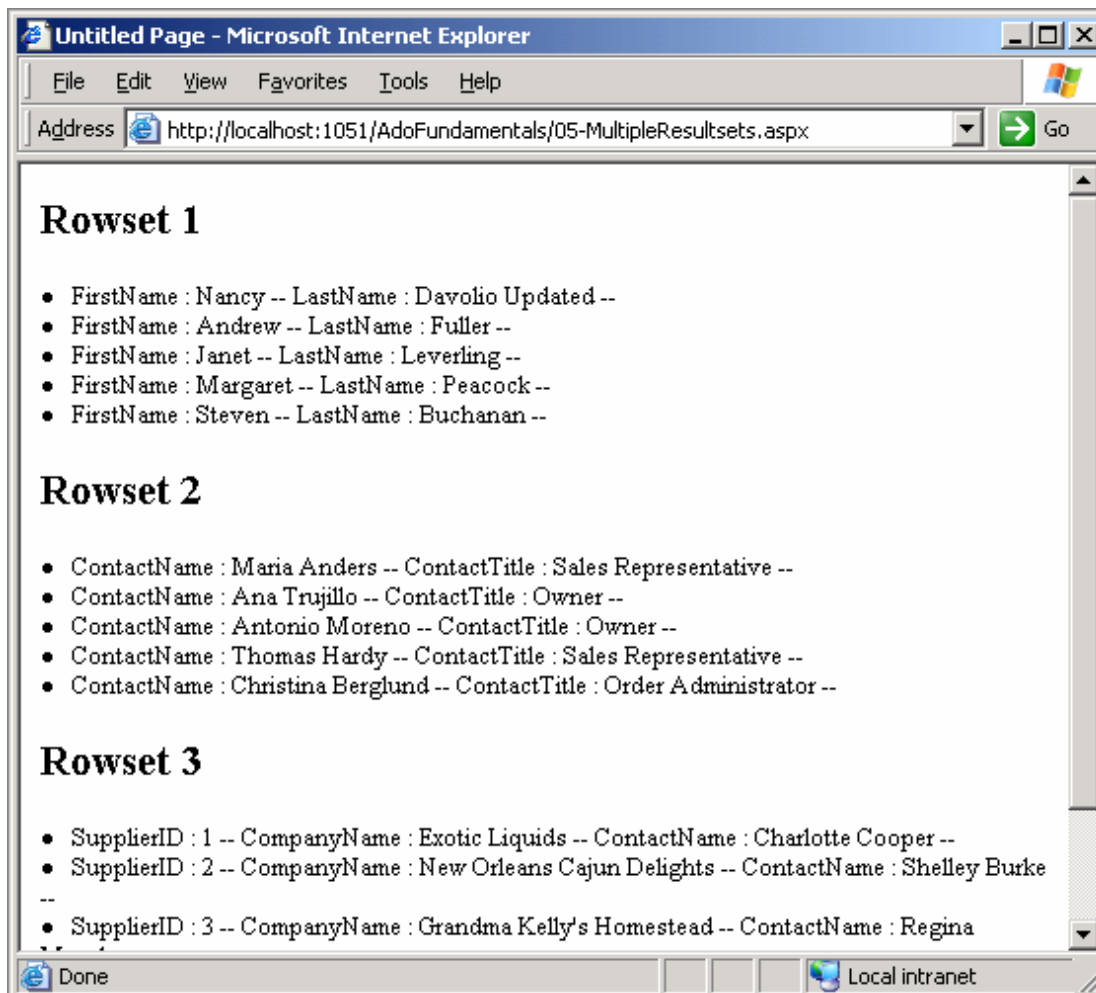
reader.Close();
//No need to con.Close since the Command Behavior is
//set to close connection upon completion
Message.Text = sb.ToString();
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Message" runat="server"
                Style="position: relative; top: 1px;"
                Width="100%">
            </asp:Label>
        </div>
    </form>
</body>
</html>

```

Note that in this case all the fields are accessed using the generic `GetValue()` method, which takes the index of the field to read. That is because the code is designed generically to read all the fields of all the returned result sets, no matter what query you use. However, in a realistic database application, you would almost certainly know which tables to expect as well as the corresponding table and field names.

Figure 4 shows the page output.



The ExecuteScalar() Method

The ExecuteScalar() method returns the value stored in the first field of the first row of a result set generated by the command's SELECT query. This method is usually used to execute a query that retrieves only a single field, perhaps calculated by a SQL aggregate function such as COUNT() or SUM().

The following procedure shows how you can get (and write on the page) the number of records in the Employees table with this approach:

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<%@ Import Namespace="System.Web.Configuration" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

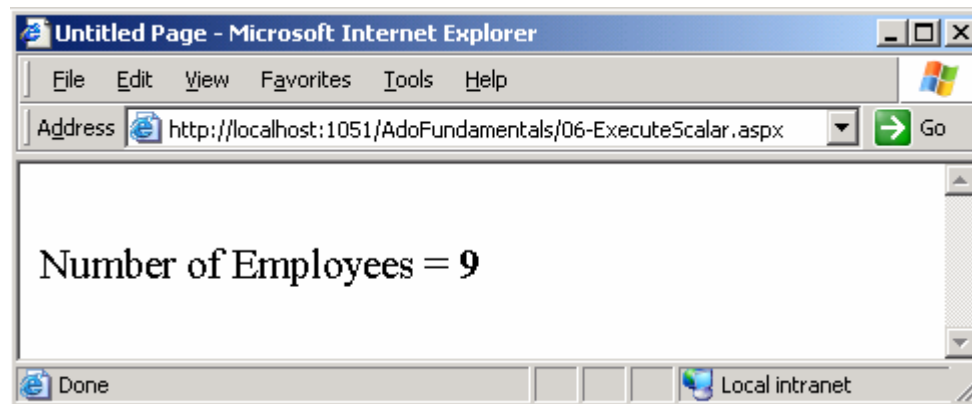
```
<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        string connectionString =
            WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection con = new SqlConnection(connectionString);
        string sql = "Select COUNT(*) FROM EMPLOYEES";
        SqlCommand cmd = new SqlCommand(sql, con);

        con.Open();
        int numberOfEmployees = (int)cmd.ExecuteScalar();
        con.Close();
        Message.Text = "<br/>Number of Employees = <b>" + numberOfEmployees +
"</b><br/>";
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Execute Scalar Demo</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Message" runat="server"
                Style="position: relative" Width="100%">
            </asp:Label>
        </div>
    </form>
</body>
</html>
```

The output of this page should look like the following:



The code is fairly straightforward, but it's worth noting that you must cast the returned value to the proper type because ExecuteScalar() returns an object.

The ExecuteNonQuery() Method

The ExecuteNonQuery() method executes commands that don't return a result set, such as INSERT, DELETE, and UPDATE. The ExecuteNonQuery() method returns a single piece of information—the number of affected records.

Here is an example that uses a DELETE command by dynamically building a SQL string:

```
<%@ Page Language="C#" %>

<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<%@ Import Namespace="System.Web.Configuration" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        string connectionString =
            WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection con = new SqlConnection(connectionString);

        string sql = "DELETE FROM [Order Details]";
        SqlCommand cmd = new SqlCommand(sql, con);
        SqlTransaction trans;
        try
        {
            con.Open();
            trans = con.BeginTransaction();
            cmd.Transaction = trans;

            int numberOfDeletedEmployees = (int)cmd.ExecuteNonQuery();

            trans.Rollback();

            Message.Text = string.Format("<br/>Number of Deleted Employees = <b>
                                     {0}</b><br/>",
                                     numberOfDeletedEmployees);
        }
        catch (Exception exp)
        {
            Message.Text = string.Format("<br/>Error:<br/> <b> {0}</b><br/>",
            exp.Message);
        }
        finally
        {
            con.Close();
        }
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Execute Non Query Demo</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Message" runat="server">
```

```

        Style="position: relative" Width="100%">
    </asp:Label>
</div>
</form>
</body>
</html>

```

Using Parameterized Commands

A parameterized command is simply a command that uses placeholders in the SQL text. The placeholders indicate dynamically supplied values, which are then sent through the Parameters collection of the Command object.

For example, this SQL statement:

```
SELECT * FROM Customers WHERE CustomerID = 'ALFKI'
```

would become something like this:

```
SELECT * FROM Customers WHERE CustomerID = @CustID
```

The placeholders are then added separately and automatically encoded.

The syntax for parameterized commands differs slightly for different providers. With the SQL Server provider, parameterized commands use named placeholders (with unique names). With the OLE DB provider, each hard-coded value is replaced with a question mark. In either case, you need to supply a Parameter object for each parameter, which you insert into the Command.Parameters collection. With the OLE DB provider, you must make sure you add the parameters in the same order that they appear in the SQL string. This isn't a requirement with the SQL Server provider, because the parameters are matched to the placeholders based on their names.

```

<%@ Page Language="C#" %>

<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<%@ Import Namespace="System.Web.Configuration" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">

    protected void Page_Load(object sender, EventArgs e)
    {
        string connectionString =
WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection con = new SqlConnection(connectionString);
        SqlDataReader reader=null ;
        string sql =
            "Select FirstName, LastName From Employees where
            EmployeeID = @EmployeeID";
        SqlCommand cmd = new SqlCommand(sql, con);
    }

```

```

string employeeIDStr = Request.QueryString["EmployeeID"];
if (employeeIDStr == null)
{
    Message.Text =
        "This page requires a query string parameter name EmployeeID";
    return;
}
cmd.Parameters.AddWithValue("@EmployeeID", employeeIDStr);
try
{
    con.Open();

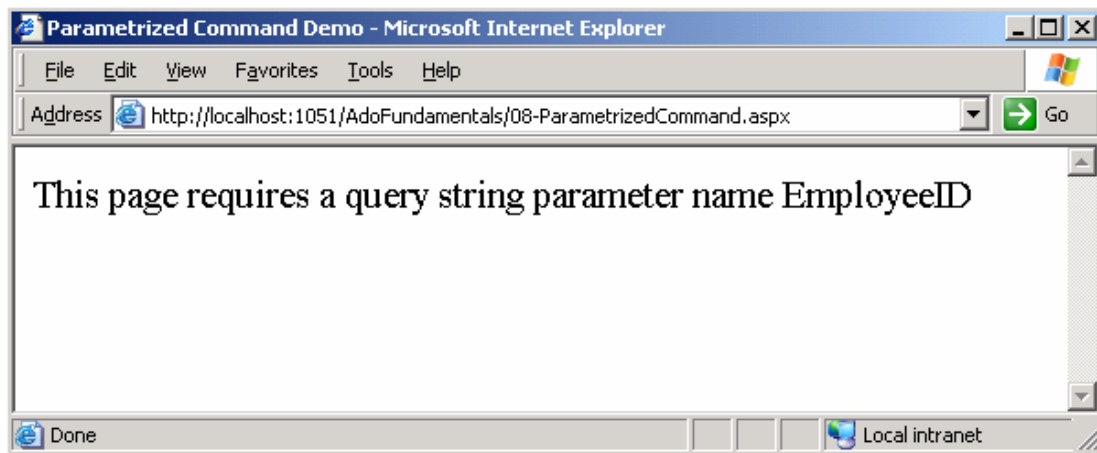
    reader = cmd.ExecuteReader();

    while (reader.Read())
    {
        Message.Text += string.Format("<br/>First Name: <b>{0}</b> Last Name:
                                     <b>{1}</b><br/>", reader[0], reader[1]);
    }
}
catch (Exception exp)
{
    Message.Text = string.Format("<br/>Error:<br/>  <b> {0}</b><br/>",
                                exp.Message);
}
finally
{
    reader.Close();
    con.Close();
}
}
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Parametrized Command Demo</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            <asp:Label ID="Message" runat="server"
                Style="position: relative" Width="100%">
            </asp:Label>
        </div>
    </form>
</body>
</html>

```

If you test your page, you would get the following output:

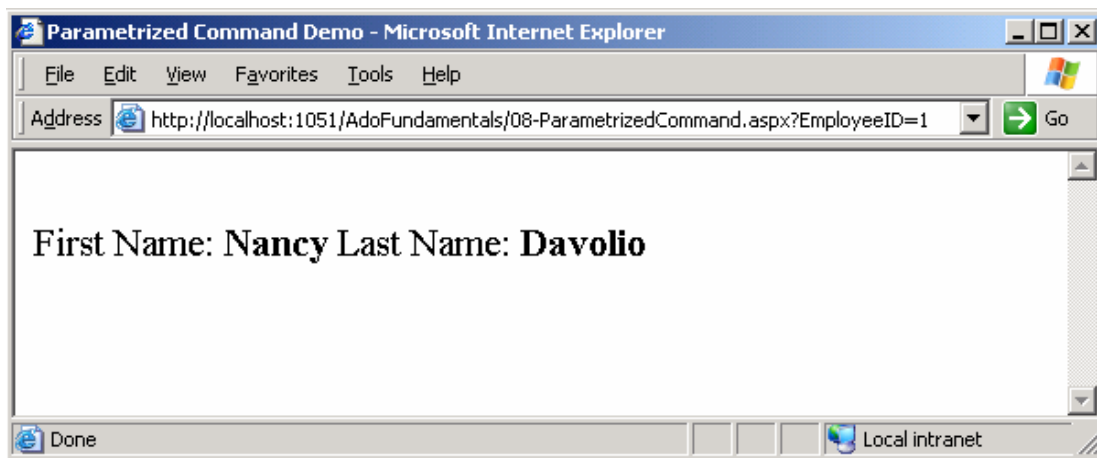


This message clearly indicates that you need to provide a query string parameter that is named EmployeeID.

To run the page successfully, add to the end of the URL the following string:

?EmployeeID=1

The output should look like the following:



Calling Stored Procedures

Parameterized commands are just a short step from commands that call full-fledged stored procedures.

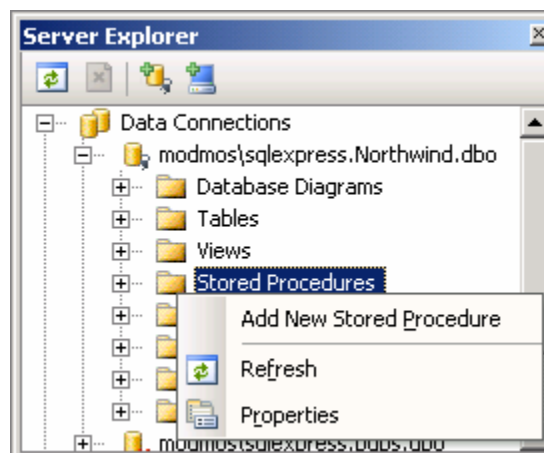
A stored procedure, of course, is a batch of one or more SQL statements that are stored in the database. Stored procedures are similar to functions in that they are well-encapsulated blocks of logic that can accept data (through input parameter) and return data (through result sets and output parameters). Stored procedures have many benefits:

- **They are easier to maintain.** For example, you can optimize the commands in a stored procedure without recompiling the application that uses it.
- **They allow you to implement more secure database usage.** For example, you can allow the Windows account that runs your ASP.NET code to use certain stored procedures but restrict access to the underlying tables.
- **They can improve performance.** Because a stored procedure batches together multiple statements, you can get a lot of work done with just one trip to the database server. If your database is on another computer, this reduces the total time to perform a complex task dramatically.

Here is the SQL code needed to create a stored procedure for inserting a single record into the Employees table. This stored procedure isn't in the Northwind database initially, so you'll need to add it to the database (using a tool such as Enterprise Manager or Query Analyzer) before you use it.

We will use Visual Studio 2005 to create a new stored procedure in the Northwind database.

1. If the Server Explorer window is not open, choose View→Server Explorer.
2. Click on the + symbol next to the Northwind Database.
3. Right Click on the + symbol next to the Stored Procedures folder as show below:



4. From the pop-up menu, choose Add New Stored Procedure
5. In the editor window that appears, type the following snippet:

```
CREATE PROCEDURE InsertEmployee
(
    @TitleOfCourtesy varchar(25),
    @LastName varchar(20),
    @FirstName varchar(10),
    @EmployeeID int OUTPUT
)
AS
BEGIN
    INSERT INTO Employees
        (TitleOfCourtesy, LastName, FirstName, HireDate)
    VALUES
        (@TitleOfCourtesy, @LastName, @FirstName, GETDATE());
    SET @EmployeeID = @@IDENTITY
END
```

Save the Stored procedure into the database by simply clicking on the Save Button.

This stored procedure takes three parameters for the employee's title of courtesy, last name, and first name. It returns the ID of the new record through the output parameter called @EmployeeID, which is retrieved after the INSERT statement using the @@IDENTITY function. This is one example of a simple task that a stored procedure can make much easier. Without using a stored procedure, it is quite awkward to try to determine the automatically generated identity value of a new record you've just inserted.

It is a very good idea to create a C# function that is responsible for, at least, creating and filling the parameters of the previous function as shown below:

```
private SqlCommand InsertEmployee(
    string titleOfCourtesy, string firstName, string lastName)
{
    SqlCommand cmd = new SqlCommand();
    cmd.CommandType = CommandType.StoredProcedure;
    cmd.CommandText = "InsertEmployee";

    cmd.Parameters.AddWithValue("@TitleOfCourtesy", titleOfCourtesy );
    cmd.Parameters.AddWithValue("@LastName", lastName);
    cmd.Parameters.AddWithValue("@FirstName", firstName);

    SqlParameter employeeID= new SqlParameter("@EmployeeID", SqlDbType.Int);
    employeeID.Direction = ParameterDirection.Output ;
    cmd.Parameters.Add (employeeID);
}
```

```
}
    return cmd;
}
```

As you can see, for each parameter of the stored procedure, we added a corresponding programmatic SqlParameter. By default, parameters are for input unless you specified otherwise as in the last parameter.

Now, we can simply call the procedure as described in the following page:

```
<%@ Page Language="C#" %>

<%@ Import Namespace="System.Data" %>
<%@ Import Namespace="System.Data.SqlClient" %>
<%@ Import Namespace="System.Web.Configuration" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<script runat="server">
    private SqlCommand InsertEmployee(string titleOfCourtesy, string firstName, string
lastName)
    {
        SqlCommand cmd = new SqlCommand();
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.CommandText = "InsertEmployee";

        cmd.Parameters.AddWithValue("@TitleOfCourtesy", titleOfCourtesy );
        cmd.Parameters.AddWithValue("@LastName", lastName);
        cmd.Parameters.AddWithValue("@FirstName", firstName);

        SqlParameter employeeID= new SqlParameter("@EmployeeID", SqlDbType.Int);
        employeeID.Direction = ParameterDirection.Output ;
        cmd.Parameters.Add (employeeID);

        return cmd;
    }

    protected void SaveEmployeeButton_Click(object sender, EventArgs e)
    {
        string connectionString =
WebConfigurationManager.ConnectionStrings["Northwind"].ConnectionString;
        SqlConnection con = new SqlConnection(connectionString);
        SqlCommand cmd =
            InsertEmployee(TitleTextBox.Text,
                           FirstNameTextBox.Text,
                           LastNameTextBox.Text);
        cmd.Connection = con;

        try
        {
            con.Open();

            int numOfRecords =(int) cmd.ExecuteNonQuery();

            if (numOfRecords == 1)
            {
                int employeeID =(int) cmd.Parameters["@EmployeeID"].Value;
                MessageLabel.Text = string.Format(
                    "The ID of the Inserted Employee: <b>{0}</b>", employeeID);
            }
        }
    }
}
```

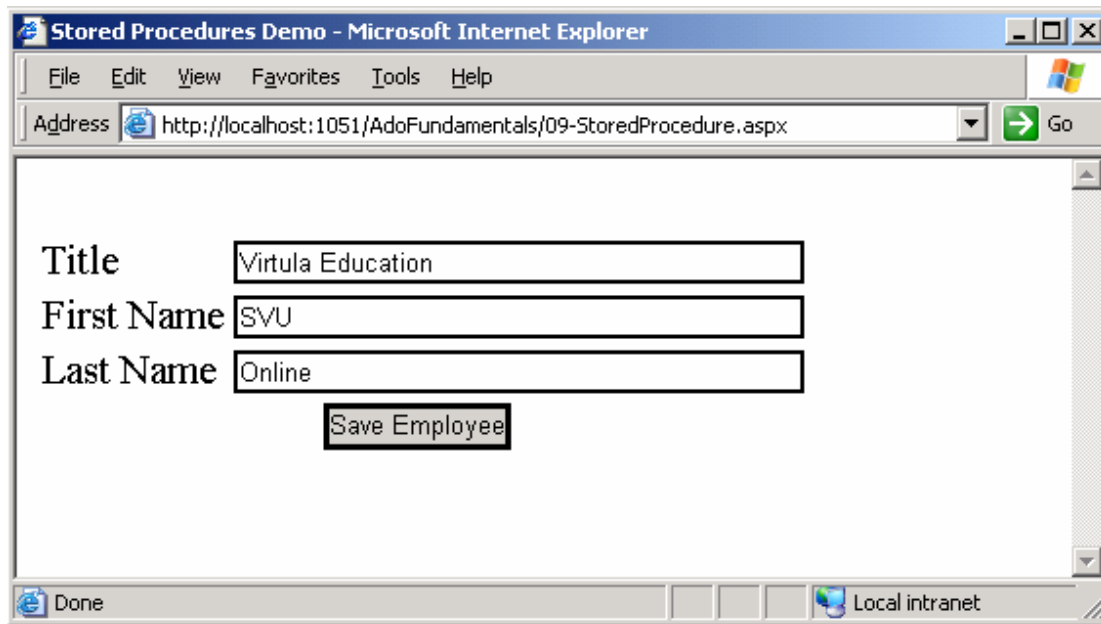
```

        catch (Exception exp)
        {
            MessageLabel.Text = string.Format("<br/>Error:<br/>  <b> {0}</b><br/>",
                exp.Message);
        }
        finally
        {
            con.Close();
        }
    }
</script>

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Stored Procedures Demo</title>
</head>
<body>
    <form id="form1" runat="server">
        <div>
            &nbsp;<table style="left: 0px; width: 75%; position: relative; top: 0px">
                <tr>
                    <td style="width: 25%">
                        Title</td>
                    <td style="width: 75%">
                        <asp:TextBox ID="TitleTextBox" runat="server"
                            BorderStyle="Solid" Style="position: relative"
                            Width="100%"></asp:TextBox></td>
                </tr>
                <tr>
                    <td style="width: 25%">
                        First Name</td>
                    <td style="width: 75%">
                        <asp:TextBox ID="FirstNameTextBox" runat="server"
                            BorderStyle="Solid" Style="position: relative"
                            Width="100%"></asp:TextBox></td>
                </tr>
                <tr>
                    <td style="width: 25%">
                        Last Name</td>
                    <td style="width: 75%">
                        <asp:TextBox ID="LastNameTextBox" runat="server"
                            BorderStyle="Solid" Style="position: relative"
                            Width="100%"></asp:TextBox></td>
                </tr>
                <tr>
                    <td colspan="2" style="text-align: center">
                        <asp:Button ID="SaveEmployeeButton" runat="server"
                            BorderStyle="Solid" Style="position: relative"
                            Text="Save Employee" Width="25%"
                            OnClick="SaveEmployeeButton_Click" /></td>
                </tr>
                <tr>
                    <td colspan="2" style="text-align: left">
                        <asp:Label ID="MessageLabel" runat="server"
                            Style="position: relative"
                            Width="100%"></asp:Label></td>
                </tr>
            </table>
        </div>
    </form>
</body>
</html>

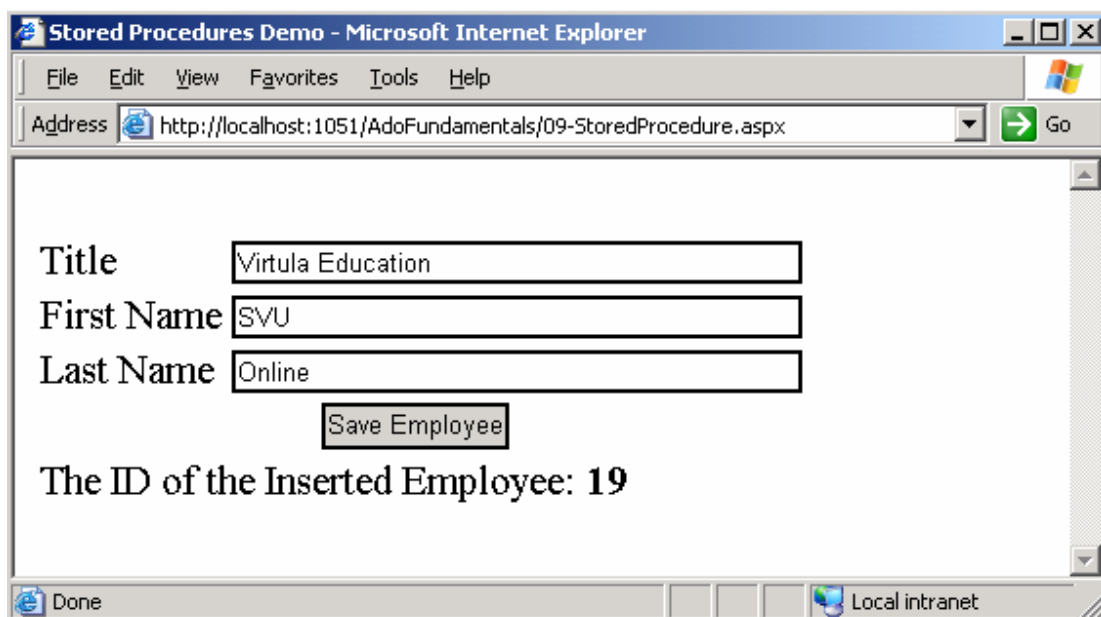
```

If you run this page, you should see the following output:



The screenshot shows a Microsoft Internet Explorer window titled "Stored Procedures Demo - Microsoft Internet Explorer". The address bar displays "http://localhost:1051/AdoFundamentals/09-StoredProcedure.aspx". The main content area contains a form with three text input fields: "Title" with the value "Virtula Education", "First Name" with the value "SVU", and "Last Name" with the value "Online". Below these fields is a button labeled "Save Employee". The status bar at the bottom shows "Done" and "Local intranet".

Enter the values shown in the previous figure, or any other values you would prefer, then click on the Save Employee Button. You should by now see the following figure:



The screenshot shows the same Microsoft Internet Explorer window. The form fields remain the same, but the "Save Employee" button is now disabled. Below the button, the text "The ID of the Inserted Employee: 19" is displayed. The status bar at the bottom remains the same.

The number 19 is actually the value of the output parameter returned by the stored procedure. This value may be different at your PC when you run the page.

You can check the table Employee through Visual Studio 2005 to ensure that a new row was added to the table employee:

Employees: Que...ss.Northwind)*				
SELECT EmployeeID, TitleOfCourtesy, FirstName, LastName				
FROM Employees				
	EmployeeID	TitleOfCourtesy	FirstName	LastName
►	1	Ms.	Nancy	Davolio
	2	Dr.	Andrew	Fuller
	3	Ms.	Janet	Leverling
	4	Mrs.	Margaret	Peacock
	5	Mr.	Steven	Buchanan
	6	Mr.	Michael	Suyama
	7	Mr.	Robert	King
	8	Ms.	Laura	Callahan
	9	Ms.	Anne	Dodsworth
	19	Virtula Education	SVU	Online
*	NULL	NULL	NULL	NULL

Conclusion

In this supplementary material, you have seen that dealing with database objects requires a few set of objects that represent the core components of any database provider.

Data retrieval and saving is actually independent of the presentation method you adopt. Data manipulation is actually the same for Standard Windows applications, Standard Web Application, as well as Mobile Web Applications.

Finish