



16212 NE 113th Ct.
Redmond, WA 98052
www.betav.com
(425) 556-9205

Beta V Corporation

Consistently Connecting with ADO.NET

*How to connect to your data source in
ADO.NET from Windows Forms, Web
Services, and ASP.NET Applications—
each time and every time.*



Consistently Connecting with ADO.NET

How to connect to your data source in ADO.NET from Windows Forms, Web Services, and ASP.NET Applications—each time and every time.

What's this all about?

If your application expects to access data (and what application doesn't), it's pretty likely that you'll want to open a connection to a database such as SQL Server, Jet, Oracle, or other data sources. If you've programmed in COM-based ADO "classic" (ADOC), you know that the secret to establishing a connection is in the `ConnectionString` property. ADO.NET is no different in that respect; however, as I discuss in this article, there are a differences between the ADOC and ADO.NET `ConnectionString` property values. Making sure that your data is secure is also something that you (or your company) are, or should be, vitally concerned with—I talk about that too.

This article endeavors to answer the following questions posed by various members of the list servers and newsgroups I frequent:

- “How can I leverage my existing ADOC connection strings to open ADO.NET connections—or can I?”
- “How do I access my ODBC data sources since the built-in .NET Data Providers don't support ODBC? Why doesn't the OleDb provider support ODBC anymore?”
- “Why can't ADO/ADO.NET open any more connections? It seems to lock up/slow down/die after about 100 connections.”
- “I want to be able to identify the user executing the code in the connection string, but if I do, I quickly run out of connections. How can I make sure only the 'right' people have access to the database?”

Before getting started, I'll assume you know a bit about the ADO.NET object model and are familiar with ADOC connection issues. After you read this article (extracted in part from my new book)¹ and experiment with the test applications, you'll know the answers to all of these questions and many others.

¹ *ADO.NET and ADO Examples and Best Practices for VB Programmers—Second Edition* (Apress ISBN 1-893115-68-2) and *ADO.NET Examples and Best Practices for C# Programmers* (Apress ISBN 1-590590-12-0)

Understanding the basic ADO.NET objects

Let's quickly review the ADO.NET objects you'll be using to get connected—starting with the .NET Data Providers. There are two providers built into the .NET Framework.

- **System.Data.SqlClient:** Used to access SQL Server 7.0 and later databases—including all versions of MSDE. This is a “managed” .NET Data Provider.
- **System.Data.OleDb:** Used to access selected OLE DB data sources—but *not ODBC*. As a rule of thumb, don't use this provider if there is an alternative. It's slower and won't leverage the features of “native” providers—such as `SqlConnection`—partly because it uses COM interop to access the OLE DB data providers. This provider is also used to access Jet/Access (.MDB) as well as FoxPro databases.

In addition, Microsoft has released several other .NET Data Providers, including `Microsoft.Data.Odbc`. This provider is your (only) path to ODBC drivers. Since this is a “managed” provider like `SqlConnection` (in contrast to `OleDb`), it's faster—and probably more stable. Remember, you can't use the `OleDb` provider to access ODBC data sources—it's simply not supported.

As I was writing this article, Microsoft released the first beta of the Oracle .NET Data Provider. We now know that it exposes the `System.Data.OracleClient` namespace. At first blush there appears to be a few new objects exposed to deal with special Oracle datatypes. Overall, though, the new provider looks pretty much like all of the other data providers—except that it expects the `SQL*NET DLL` (from Oracle) to be in place. But, remember that we saw quite a few namespace changes in the other providers as they progressed from beta to production; so, I would not be writing any production code using the new provider—not just yet.

Microsoft is not the only source for .NET Data Providers. DataDirect publishes their own set of “native” providers for Oracle, Sybase, and DB2 that don't require the (expensive) trip through unmanaged COM-interop layers to work.

Each of the .NET Data Providers exposes named instances of the following classes. These are used to connect, fetch data, and populate the `System.Data.DataSet` and `DataTable` classes.

- **Connection:** Used to get connected based on arguments provided in a connection string.
- **Command:** Used to manage the `SELECT` or action queries and their parameters and to construct a `DataReader`. Interconnects the `Connection` object with the `SQL` query.
- **DataReader:** Used to perform low-level data I/O. Returns a `RO/FO` (read-only/forward-only) data stream—not unlike the default `ADOc Recordset`—by executing the `Command` object.
- **DataAdapter:** Used to construct a `DataSet` and/or `DataTable` system objects and to manage updates to these objects. Addresses `Command` objects to query and manage the data.

These data classes are implemented by each .NET Data Provider, as shown in the following table:

Object Type	<code>SqlConnection</code>	<code>OleDb</code>	<code>Odbc</code>	<code>Oracle</code>
-------------	----------------------------	--------------------	-------------------	---------------------

Connection	SqlConnection	OleDbConnection	OdbcConnection	OracleConnection
Command	SqlCommand	OleDbCommand	OdbcCommand	OracleCommand
DataReader	SqlDataReader	OleDbDataReader	OdbcDataReader	OracleDataReader
DataAdapter	SqlDataAdapter	OleDbDataAdapter	OdbcDataAdapter	OracleDataAdapter

Table 1. ADO.NET objects as implemented by .NET Data Providers

This article (like many others) uses the term “Connection” object to refer to the equivalent object as implemented by the chosen .NET Data Provider. I tried to use “xxxConnection”, but my e-mail filters kept throwing out any mail that discussed this “xxx-rated” object; so, let’s just stick with “Connection” knowing that this really refers to the corresponding object as implemented by the chosen .NET Data Provider—such as SqlConnection.

While the various implementations are very similar, there are differences—but few within the scope of this article. I’ll note them as I go. Virtually all of the properties and methods and their constructors are implemented so that you can easily migrate code (or your skills) from one provider to another with few surprises.

How does the ADO.NET connection mechanism work?

Okay, you want to use ADO.NET to get connected. How can you accomplish this? As with ADOc, you have lots of options—each with its own benefits and issues.

- **Build a Connection object, set itsConnectionString property and use its Open method.** This seems the most like ADOc, and makes sense when creating a DataReader, but this approach does not make sense if you’re building a DataSet. The following routine uses the SqlClient .NET Data Provider:

```
Option Strict On
Imports System.Data.SqlClient
Imports System.Data.SqlClient
Imports Microsoft.Data.Odbc
Imports System.Data.OleDb
Public Class Form1
    Inherits System.Windows.Forms.Form
    Dim cnSql As SqlConnection
    Dim cnOdbc As OdbcConnection
    Dim cnOleDb As OleDbConnection
    Dim cmd As SqlCommand
    Dim da As OleDbDataAdapter
    Dim dr As OleDbDataReader
Private Sub OpenCn()
    Dim strConnect As String = "Data Source=betav9;initial catalog=biblio;" _
    & "connection timeout=2;UID=fred;pwd=xx"
    Try
        ' Trap problems during the open
        ' Instantiate the global SqlConnection object
        cnSql = New SqlConnection(strConnect)
        cnSql.Open()
        ' Open the global connection
    Catch exSql As SqlException
```

```

        MsgBox(exSql.Message)      ' Server down? LoginID not permitted?
    Catch ex As Exception
        MsgBox(ex.ToString)      ' Some other exception
    End Try
End Sub
End Class

```

In this example, the `ConnectionString` property arguments include a specific server name (Betav9) and point to the “Biblio” default database, a two-second connection timeout, and a specific SQL Server Login ID and password.

- **Build a Connection object, set its `ConnectionString` property, and point a `DataAdapter` at the Connection and let the `DataAdapter Fill` or `Update` method open and close it for you.** This seems more complicated, but is really pretty easy; again, this only applies when you need a `DataSet` (often you don’t). This example uses the `Odbc .NET Data Provider`.

```

Private Sub btnOpenOdbc ()
Dim strConnect As String = "DSN=Betav9DSN;UID=fred;pwd=xx"
    Try      ' Trap problems during the open
        ' Instantiate the global OdbcConnection object
        cnOdbc = New OdbcConnection(strConnect)
        cnOdbc.Open()      ' Open the connection
        cnOdbc.ChangeDatabase("biblio") ' Change default DB
    Catch exOdbc As OdbcException
        MsgBox(exOdbc.Message)      ' Trap ODBC-generated exceptions
    Catch ex As Exception
        MsgBox(ex.ToString)      ' Some other exception
    End Try
End Sub

```

In this case, the `Odbc .NET Data Provider` expects to see a “traditional” ODBC connection string. This connection string points to a registered DSN and specifies a specific Logon ID and password. I use the `ChangeDatabase` method to set the correct default database (“Biblio”). You can also use an ODBC “DSN-less” connection string, as in the following:

```

Dim strConnect As String = "SERVER=betav9;Driver={Sql Server};UID=fred;pwd=xx;"

```

Not much, if anything, has changed when you compare the new `.NET Odbc .NET Data Provider` with your existing ODBC connection strings.

- **Build a `DataAdapter` object constructor using a valid `ConnectionString` instead of a Connection object.** In this case, `ADO.NET` constructs a `Connection` object for you behind the scenes. This is a shortcut technique that might be easier to code. This example uses the `OleDb .NET Data Provider`. Remember, this should be the last `.NET Data Provider` you choose—when no native or ODBC providers are available.

```

Function GetDataSet(ByVal intYearBorn As Integer) As DataSet
    Dim strConnect As String = "Provider=SqlOleDb;Initial catalog=biblio;" _
        & "data source=betav9;integrated security=SSPI;"
    Dim cnOleDb As OleDbConnection
    Dim da As OleDbDataAdapter
    Dim ds As New DataSet()
    Try      ' Trap problems during the open
        ' Instantiate the local OleDbConnection and OleDbDataAdapter objects
        cnOleDb = New OleDbConnection(strConnect)
        da = New OleDbDataAdapter("SELECT Author, Year_Born " _
            & " FROM Authors where Year_Born = ?", cnOleDb)      ' Note:No named parms
    End Try

```

Consistently Connecting with ADO.NET

```
da.SelectCommand.Parameters.Add("@YearWanted", _
    OleDbType.Integer).Value = intYearBorn
da.Fill(ds, "Authors")
Return ds
Catch exoledb As OleDbException
    MsgBox(exoledb.ToString)      ' Could not find server (it's down?) or LoginID has no
permissions
Catch ex As Exception
    MsgBox(ex.ToString)          ' Some other exception
End Try
```

This example uses a connection string not unlike what you're used to seeing in an ADOc application. In this case, I pointed to the SQL Server OLE DB provider "SqlOleDb" that's accessed via COM interop. I pointed to the named SQL Server using the Data Source keyword. Note that the Integrated Security keyword *requires* "SSPI"—you can't use "True" or "Yes" as you can with other providers.

The remaining code in the example sets up an OleDbDataAdapter, its SelectCommand (with a simple SELECT query) and the input parameter for this query. I used the OleDbDataAdapter Fill method to open the connection, execute the query, populate the DataSet's DataTable, and close the connection.

Building a connection string

Regardless of the technique you use, you *must* construct a connection string—it's used to set the Connection (SqlConnection, OleDbConnection, OdbcConnection, OracleConnection...) object's ConnectionString property. No, you can't just set up a Connection object and change its properties as you can with ADOc. Most of the Connection object's properties are read-only and can *only* be set by setting the ConnectionString property, which is the only way to change the connection behavior of the .NET Data Provider including:

- **How is connection security to be managed and who (or what) is trying to connect?** Set the UID/PWD or Integrated Security keywords.
- **What server should be accessed?** Use the Data Source, DSN, or Server keywords.
- **Which network protocol should be used to access the server?** Use the Network keyword.
- **Should the connection be pooled?** With the SqlClient .NET Data Provider you can manage several connection pooling attributes.
- **Should the connection participate in transactions?** Set the Enlist keyword.

You'll discover that most of the Connection object properties are read-only—both before and after you use the Open method on the Connection object; so, unlike ADOc, you have to depend on the ConnectionString property to specify every characteristic of your connection, including how the connection pool is managed. No, you won't be able to pass in last-minute UserID and Password values in the Connection Open method as you did in ADOc. Security settings *must* be provided ahead of time in the ConnectionString property. After connecting, you can change the default database using the ChangeDatabase method.

The ConnectionString property is parsed at run time; when the provider finds an exact match for one of the recognized argument keywords, it assigns the value to the appropriate property. This means that if you don't spell a keyword correctly or try to set it to an invalid argument,

you'll generate an exception—even in a Dim statement. This also means you'll probably want to wrap your Dim statement constructors in Try/Catch blocks.

To make things easy, there are several alternative keywords which permit developers familiar with legacy ADOc (OLE DB/ODBC) connection strings to use many of the old keyword/value pairs. The following tables list these keywords. Table 1 lists the keywords exposed by specific Connection object properties—there aren't many. It also shows their default values and what effect they have on the connection itself. The keywords for the Odbc and OleDb providers are the same as you used in ADOc or other direct API applications.

Keyword	Default	Sets Connection Property	Function
Connect Timeout or Connection Timeout	15	ConnectionTimeout	How long (in seconds) the provider waits for a server to open a connection <i>after</i> network connectivity is established. If the connection pool is full, this is how long ADO.NET waits for a connection to free up.
Initial Catalog	(As set by UID account)	Database	Sets the default database. Overrides default set by UID account. No, "DBQ" is no longer supported.
Database	(As set by UID account)	Database	Also sets the default database. Overrides default set by UID account. (SqlClient only)
Packet Size	8192	PacketSize	Size in bytes of the network packets used to communicate with an instance of SQL Server. Smaller packets take less time, but carry less data.
Workstation ID	The local computer name	WorkstationID	The name of the workstation connecting to SQL Server. To help identify your computer in the sp_who report.

Table 2. ConnectionString keywords exposed as Connection object properties

There are a bevy of other keywords that ADO.NET recognizes that are not exposed as Connection object properties—as listed in Table 2:

Keyword	Default	Function
Application Name	.Net SqlClient Data Provider'	To identify the application's connection in sp_who (or sp_who2) or in the profiler.
AttachDBFilename, extended properties, or Initial File Name	N/A	The name of the primary file, including the full path name, of an attachable database. The database name must be specified with the keyword 'database'.
Current Language	N/A	The SQL Server Language record name.
Data Source, Server, Address, Addr, or Network Address	N/A	The name or network address of the instance of SQL Server to which to connect. This can include "." or "local" to refer to the local server. Be sure to include the server instance if you have multiple instances loaded. For example, "betav8\ss2k" refers to a named instance "ss2k" on the "betav8" server.
DSN	N/A	Points to a registered Data Source Name. (ODBC Only)
File name	N/A	Points to a Universal Data Link (UDL) file containing the connection string. (OleDb only)

Integrated Security, or Trusted_Connection	'false'	Whether the connection is to use "domain-managed" security. Recognized values include 'false', 'true', and 'sspi', (except in the OleDb provider which requires 'SSPI') which is equivalent to 'true'. If this option is true, any UID or PWD values are ignored and the connection is made using the process identity based on Windows authentication.
Isolation Level	ReadCommitted	The transaction isolation level for the connection. Can be: ReadCommitted, ReadUncommitted, RepeatableRead or Serializable.
Network Library, or Net	'dbmssocn'	The network library used to establish a connection to an instance of SQL Server.
Password, or Pwd	N/A	The password for the SQL Server account logging on. No, there is no Password property and it's removed from the ConnectionString unless Persist Security Info is set to "true". ²
Persist Security Info	'false'	When set to 'false', security-sensitive information, such as the password, is not returned as part of the connection if the connection is open or has ever been in an open State . Resetting the connection string resets all connection string values, including the Password.
Prompt	4	Specifies if a dialog should prompt the user for missing/incorrect UserID/Password values. 1: Always prompt, 2: Prompt if more information is needed, 3: Prompt if more info is needed but optional parameters are not permitted, 4: never prompt. (OleDb only)
Provider	(no default)	Specifies the OLE DB provider to use. Required for OleDb connections.
User ID	N/A	The SQL Server login account. This value is ignored if you use "trusted" connection security.

Table 3. General keywords used to establish and configure your Connection object

CAUTION *Until this problem is fixed (if ever), you should never (as in never) simply accept user input for UID, PWD, or any other ConnectionString argument without carefully filtering the value. Make sure that the user does not imbed extra connection string parameters in these values (for example, entering a password as "validpassword;database=somesecretdb" in an attempt to attach to a different database). Using "Integrated Security=true" can bypass this problem because it ignores any passed UID or PWD values; however, it does not stop users from passing in extra keyword/value pairs if you give them that opportunity.*

Closing connections

If you expect your application to scale, you had best build in code to doubly ensure that the connection(s) you open are actually closed—or at least released to the connection pool. If you don't close connections you open, ADO.NET and Visual Studio .NET won't do it for you—at least not always. Don't depend on connections getting closed when Connection objects fall out of scope, as was the case in Visual Basic 6.0. It does not work that way in Visual Studio .NET.

² See the caution regarding a security issue with user-supplied keyword values.

It turns out that, by default, ADO.NET does not really close connections when you use the Close method or when the Fill/Update methods are completed. When a connection is closed it is simply released back to the connection pool. Each process gets its own connection pool and so does each new connection string. Incidentally, in some cases, these pools can hang around indefinitely—until the system is rebooted. The connections managed by the pool are eventually closed by the pool manager, but *only* if you have released use of the connection. Table 4 shows how ADO.NET manages connections for you:

Object	Method	Operation
Connection	Close	Immediately releases the connection to the pool.
DataAdapter	Fill	Opens connection, runs query, returns DataSet/DataTable, and closes connection. Immediately releases connection to the pool.
DataReader	Close	Closes connection <i>only</i> if CommandBehavior.CloseConnection is specified.
DataReader	(Data bound)	Closes connection once complex bound control populated (Web Applications only)

Table 4. ADO.NET managing open connections

Leaving connections opened and orphaned in the connection pool is a far more common problem when using the DataReader where you have to open the Connection object manually. Even if you use the CloseConnection CommandBehavior when creating the DataReader, you'll still have to either close the connection, close the DataReader, or bind the DataReader to a complex bound control—such as a DataGrid (which closes the connection for you after population).

Activating, tuning and disabling the connection pool

When using the new SqlClient .NET Data Provider to access (just) Microsoft SQL Server 7.0 and 2000 (and later), you can turn connection pooling off (it's on by default), change the size of the pool, and tune its operations—at least to some extent. Managing and monitoring the connection pool is a broad, deep river of issues that's beyond the scope of this article. I discuss these issues in my books and lectures.

Creating connection strings for new data sources

One other tip: When working with a new data source, try to get the Visual Studio .NET IDE to create the connection string for you. Using the Server Explorer window, create a new Data Connection. By following the dialogs, you'll get a new connection string to appear in the property window once you select the Data Connection. While this string is somewhat verbose (it has all of the default settings as well), it can help get you connected to a data source you haven't worked with before.