

## The Connections of ADO.NET Connection Pooling

by Dino Esposito

Just as an oven is necessary to a pizza maker, database connections are essential to data-driven applications. The startup of the oven—especially a wood-burning one—is relatively expensive, at least compared to what it takes to bake a single pizza. Can you imagine how inefficient a pizzeria would be if they restarted the oven for each order? Similarly, your application cannot reasonably afford to start a new "true" connection object for each command or operation to execute.

That's where connection pooling fits in.

Connection pooling is a technique for sharing a set of predefined connection objects among clients. Each object in the pool is instantiated upon startup and remains idle waiting for a request to serve. Just like firing up the pizza oven, the creation of the connection object is the most expensive part; pooling a set of similar objects reduces the total cost and guarantees better performance to the application.

ADO.NET and .NET data providers make connection pooling mostly transparent to developers. However, the benefits of connection pooling can be dramatically cut down, or even cancelled out, by poorly chosen connection string parameters or poor Data Access Layer (DAL) design.

In this article, I'll review the characteristics of the main connection classes in ADO.NET and examine some DAL programming patterns to ensure a good balance of pooling and security.

### The Connection Model

The connection represents an open and unique link to a data source. In a distributed system, this often involves a network connection. Depending on the underlying data source, the programming interface of the various connection objects may differ quite a bit. A connection object is specific to a particular type of data source, such as SQL Server and Oracle. Connection objects can't be used interchangeably across different data sources, but all share a common set of methods and properties grouped in the **IDbConnection** interface.

In ADO.NET, connection objects are implemented within data providers as sealed classes (that is, they are not further inheritable). This means that the behavior of a connection class can never be modified or overridden, just configured through properties and attributes.

In ADO.NET, all connection classes support connection pooling, although each class may implement it differently. Connection pooling is implicit, meaning that you don't need to enable it because the provider manages this automatically.

The connection object is much simpler in ADO.NET than it was in earlier versions like ADO 2.x. It drops some of the functions like the capability of executing commands (method `Execute`) and reading schema information (method `OpenSchema`).

## ADO.NET Connection Classes

ADO.NET comes with four data providers: SQL Server 7.0 and beyond, Oracle, ODBC data sources, and OLE DB data sources. Each of these has its own connection class with a common layer of functions plus specific features. The **SqlConnection** class represents a connection to a SQL Server database, version 7.0 or higher. In ADO.NET 2.0, this class also represents a connection to the newest version-SQL Server 2005. The **OracleConnection** class connects to an Oracle database, while **OdbcConnection** represents an open connection to a data source set through an ODBC driver. Finally, **OleDbConnection** establishes a link with a data source via an OLE DB provider.

All connection classes use the **ConnectionString** property to specify the data source of choice and configure the runtime environment. The connection string can be set either through the aforementioned string property or passed on to the class constructor. Each connection class supplies a bunch of read-only properties to extract details out of the connection string. The **Database** property gets the name of the database to be used after the connection is opened. The **DataSource** property indicates the database server through any of the following: file name, server name, or network path. It is important to notice that these detail properties are read-only; their value can only be set through the connection string.

The property **ConnectionTimeout** gets the timeout value for the physical connection to take place. The default timeout is 15 seconds. A value of 0 means that the application never times out and will indefinitely wait for the connection to open; while acceptable, this setting should be avoided in practice to prevent the application from hanging. The connection timeout can be set through the **ConnectionString** property. The Oracle connection class doesn't support any timeout and always returns 0-meaning that no maximum wait time is set on an attempt of connection before an error is generated.

Likewise, the Oracle connection class doesn't support the **ChangeDatabase** method, which other classes use to dynamically change the database for open connections.

In addition to the properties and methods defined by the IDbConnection interface-all classes must implement it-each connection class can provide custom members exposing functionalities specific to the underlying DBMS. The SQL Server provider has three properties: **PacketSize**, **WorkStationId**, and **ServerVersion**.

**ServerVersion** gets the *major.minor.build version* number of the current instance of SQL Server. **WorkstationId** contains the network name of the client accessing SQL Server. **PacketSize** gets the size in bytes of the network packets used to communicate with an instance of SQL Server. The default size is 8192 bytes. A different value can be set through the connection string using the Packet Size attribute. Feasible values range from 512 to 32767 bytes.

The SQL Server .NET data provider manages pooling using internal classes, as does the Oracle data provider. As you'd expect, OLE DB and ODBC data providers implement connection pooling using the ODBC and OLE DB infrastructure.

## Managing a Connection

Each connection pool is associated with *one distinct* connection string; the connection string is parsed using an exact-match algorithm. This means that an additional white space, or even a different order of the same attributes, leads to distinct pools. The following code creates two distinct pools.

```
// When Open is called, the first pool is created
SqlConnection conn = new SqlConnection();
conn.ConnectionString = "Integrated
Security=SSPI;Initial Catalog=northwind";
conn.Open();

// When Open is called, a second pool is created
SqlConnection conn = new SqlConnection();
conn.ConnectionString = " Initial
Catalog=northwind;Integrated Security=SSPI;";
conn.Open();
```

Once created, a connection pool is not destroyed until the active process ends. The reason is that maintenance of inactive or empty pools involves minimal system overhead, and this is preferable to recreating the pool.

Upon creation, the pool is filled until the maximum size is reached. Connections are created and opened sequentially. To open a connection, you use the `Open` method. It takes no arguments and draws a connection object from the pool that matches the connection string, if any. (If not, a new pool is created.) The `Open` method looks for a usable connection. A connection is reckoned usable if it is currently unused, has a matching transaction context or is not associated with any transaction context, and has a valid link to the server.

If no usable connection is found, the request is queued and served as soon as a valid connection is returned to the pool. Connections return to the pool only if explicitly closed. To close a connection, you can either use the **Close** or **Dispose** method. (Dispose ends up calling `Close` to cancel pending commands and roll transactions back.) The **Close** method releases the connection object to the pool. If connection pooling is disabled, any call to `Close` terminates the connection instead.

Connections are removed from the pool in case of a timeout or if a severe error occurs. You should never rely on the garbage collector to return a connection to the pool. If the connection object goes out of scope, the underlying connection to the server is not automatically closed, meaning that any connection must always be explicitly closed

Returning a connection to the pool makes the object reusable for another call; the underlying connection to the database is not severed, though. As mentioned, this happens only if a severe error occurs. With SQL Server, if an exception is thrown while a command is being executed, the connection remains open only if the severity level is low (less than 20). Beyond level 20, the connection gets automatically closed. Next, the user can reopen the connection and continue with the application.

As mentioned, connections are assigned also based on the transaction context. The context of the requesting thread and the assigned connection must match. Each pool is subdivided into connections with no associated transaction context, and into various groups each filled with connections with a particular transaction context. When a connection is closed, it returns into the appropriate group based on its transaction context. This mechanism allows you to close the connection even though a distributed transaction is still pending. Later, you retrieve the connection with the proper transaction context and can commit or rollback the distributed transaction.

Golden rules for effectively using connections and pooling can be summarized as follows:

- Enable connection pooling
- Store connection strings securely
- Open connections late and close them early
- Design your data access code thinking in terms of user roles, not user accounts

Let's delve into these a bit more deeply.

### **Enable Pooling**

You enable and disable connection pooling through the connection string used to connect to the server database. The table below lists the attributes you can take advantage of.

| <b>Attribute</b>    | <b>Description</b>  |
|---------------------|---|
| Connection Lifetime | Sets the maximum duration in seconds of the connection object in the pool. When the object is returned to the pool, the pooler makes a simple calculation. If the creation time, plus the lifetime, is earlier than the current time, the object is destroyed. This is useful in clustered configurations to force load balancing between a running server and a new server that just showed up online. The default value is 0, meaning that pooled connections never time out. |
| Connection Reset    | Determines whether the database connection is reset to the state of login time when being drawn from the pool before use. Default is true. If false, by calling <code>ChangeDatabase</code> you can have the same physical connection serve different databases at different times.   |
| Enlist              | Indicates that the pooler automatically enlists the connection in the creation thread's current transaction context. Default is true. If enlistment is disabled, you can manually enlist the connection by using the method <code>EnlistDistributedTransaction</code> . (Requires ADO.NET 1.1.)   |
| Max Pool Size       | Maximum number of connections allowed in the pool. Default is 100.  |

|               |  |
|---------------|--|
| Min Pool Size | Minimum number of connections allowed in the pool. Default is 0.   |
| Pooling       | Indicates that the connection object is drawn from the appropriate pool, or if necessary, is created and added to the appropriate pool. Default is true. |

To disable connection pooling, you set **Pooling** to false. Attributes that require a Boolean value also accept *yes-no* in addition to *true-false*.

### Store Connection Strings

There are at least two good reasons to store connection strings outside the application—flexibility and security. As expected, though, these two parameters clash with raw performance and require you to write extra code. In ADO.NET 1.x, here are your options:

| Option            | Description  |
|-------------------|--|
| Hardcoded strings | Maximum performance, but significantly reduced flexibility. In general, external storage offers you great flexibility with an overall negligible performance hit but potentially severe security issues.   |
| .config files     | A form of external storage, these are easy to deploy and access programmatically. In .NET 1.x, strings are normally stored as clear text (including any passwords) unless you take care of encryption/decryption yourself. Security of sensitive data is greatly enhanced in .NET 2.0 thanks to protected sections in .config files. |
| UDL files         | Only supported by the OLE DB managed provider. It poses the same security issues as .config files but without the improvements brought by .NET 2.0.  |
| Registry          | Stores the strings in the system registry. It provides a <i>naturally</i> secure approach (even more if encrypted) but poses some deployment issues and requires custom coding.  |

In .NET 2.0 the favorite solution is to use .config files from both Windows Forms and ASP.NET applications. This approach weds the flexibility of external storage with the security guaranteed by protected sections. In addition, .NET 2.0 offers a transparent API that automatically reads and decrypts and a tool to encrypt required sections.

### Open and Close

The main advantage of connection pooling is the frequent reusability of ready-to-use connections. The longer the connection object waits unused in the pool, the more potential scalability your application gains. For this reason, each application should acquire any connection objects as late as possible to return them to the pool as soon as possible. This simple rule is fundamental to any high-performance application. Following this guideline doesn't guarantee anything about the final performance; if you don't, though, you're certainly on the way to a significant performance hit.

## User Roles

To make connection pooling boost an application, you must use the same connection string for all data access tasks. This approach clearly complicates security management at the database level, if that is what you really want. If impersonation of user accounts is necessary to access the database, you end up having many small pools with limited reuse of connections.

A good way to combine pool reusability and data access security is to use a common set of accounts, one per each role you recognize to users of your system.

Generally speaking, you can perform data access using any of the following identities:

- The process identity of the calling process
- One or more service identities (roles)
- The original caller's identity (impersonation)

Using the worker process identity is common in ASP.NET applications. This takes advantage of the Windows authentication model in which the account of the currently logged user is silently and invisibly passed to SQL Server. The application's account must be mirrored and secured at the database level. In this way, data tier components use trusted connections to perform operations. SQL Server needs more time to handle a trusted connection, but the performance hit is negligible compared to the execution time of the command. Note that SQL Server must be configured to run either in Windows Authentication Mode or Mixed Mode.

If one fixed identity can't just serve any database-related requests in the application, you need to look elsewhere. Consider untrusted connections, where SQL Server receives the user ID and password and authenticates them. This approach assumes a middle tier that guarantees the rights of the caller to execute the commands. Each user is authenticated at the gate (that is, IIS/ASP.NET) and assigned a role. The middle tier then checks the user against role membership and allows the request to reach data tier components. These components look at the role and impersonate the account specific for the role. Of course, the account must have any database permissions needed to operate according to its privileges. Few accounts are used by all users, which ensures both connection pooling and security.

Finally, if your database requires user-level authorization, you will need to just impersonate the original caller. In this case, database objects determine what a user can do and cannot do. This solution affects scalability, because it defeats connection pooling, but the hit can be alleviated in some way. You can use a unique identity (or a few identities) and modify your stored procedures to make them accept an extra parameter-the user ID. Internally, the stored procedure will check the ID and implement some logic to block or allow certain operations. It goes without saying that this approach requires a lot of T-SQL coding or extended stored procedures, both of which are not exactly pleasant techniques for most developers. The embedded CLR of SQL Server 2005 makes these sort of tasks simpler, but will likely put you in contrast (so to speak) with the database administrator. To sum up, avoid database impersonation whenever possible.



## Summary

Connection pooling is a fundamental and essential part of distributed applications. ADO.NET makes pooling transparent, but the underlying mechanism is not fully disconnected from the upper-level code. In other words, the way you design your code can influence (usually poorly) the behavior of the pooler. Be aware of this and apply the few golden rules listed here to get the most out of this valuable and mostly invisible infrastructure.