See This in MSDN Library

Generic Coding with the ADO.NET 2.0 Base Classes and Factories

Bob Beauchemin
DevelopMentor

July 2004

Page Options

Average rating:
7 out of 9

Rate this page

Print this page

E-mail this page

Applies to:
  Microsoft ADO.NET 2.0
  Microsoft Visual Studio 2005 and previous versions
  C# programming language
  Visual Basic programming language

Summary: Learn more about the new functionality in ADO.NET 2.0 that enables you to create generic data access code. (14 printed pages)

Download the associated VSGeneric.exe sample code.

Contents

Introduction

Microsoft ADO.NET is a database-general API. It is an API in which database vendors write plug-in data providers for their databases, such as OLE DB/ADO (providers), ODBC (drivers), or JDBC (drivers). It is not an API with a database-specific set of functions such as dbLib, or a database-specific object model such as OO4O (Oracle Objects for OLE).

Predecessors of ADO.NET factored their providers into sets of interfaces, defined by a strict specification. For example, each OLE DB provider writer implemented a standard set of interfaces, and may have included both required and optional interfaces. Required interfaces specified the lowest common denominator; optional interfaces were defined for advanced database/provider features. Although the OLE DB specification permitted providers to add new interfaces to encapsulate database-specific functionality, the ADO library that programmers used with OLE DB providers did not generally use those interfaces.

ADO.NET was designed from the beginning to allow the provider writer the space to support database-specific features. Provider writers implemented a set of classes, like a **Connection** class, a **Command** class, and a **DataReader** class. Because classes and interfaces are visible in Microsoft .NET (in ADO, only interfaces were visible to the programmer), ADO.NET provider writers implement a set of provider-specific classes, exposing generic functionality via interfaces, and database-specific functions as class properties and methods. Provider writers implemented their own parallel hierarchy of classes. Table 1 shows the parallel class hierarchies for the SqlClient and OracleClient data providers, as well as the interfaces implemented by both data providers. These interfaces are defined in the **System.Data** namespace.

Table 1. Provider-specific classes and generic interfaces in ADO.NET 1.0/1.1

SqlClient class	Oracle class	Generic interface
SqlConnection	OracleConnection	IDbConnection
SqlCommand	OracleCommand	IDbCommand
SqlDataReader	OracleDataReader	IDataReader/IDataRecord
SqlTransaction	OracleTransaction	IDbTransaction
SqlParameter	OracleParameter	IDbDataParameter
SqlParameterCollection	OracleParameterCollection	IDataParameterCollection
SqlDataAdapter	OracleDataAdapter	IDbDataAdapter

In ADO.NET 1.0 and 1.1, programmers had two choices. They could code to the provider-specific classes or the generic interfaces. If there was the possibility that the company database could change during the projected lifetime of the software, or if the product was a commercial package intended to support customers with different databases, they had to program with the generic interfaces. You can't call a constructor on an interface, so most generic programs included code that accomplished the task of obtaining the original IDbConnection by calling "new" on the appropriate provider-specific class, like this.

```
enum provider {sqlserver, oracle, oledb, odbc};
public IDbConnection GetConnectionInterface()
{
// determine provider from configuration
provider prov = GetProviderFromConfigFile();
IDbConnection conn = null;
switch (prov) {
  case provider.sqlserver:
    conn = new SqlConnection(); break;
  case provider.oracle:
    conn = new OracleConnection(); break;
 // add new providers as the application supports them
 }
return conn;
}
```

The **GetProviderFromConfigFile** method referred to above was hand-coded by each company, as was the mechanism for storing the provider information in the configuration file. In addition, vendors that supported multiple databases had to write vendor-specific code to allow the user or software programmer to choose which database to use at installation time.

ADO.NET 2.0 codifies configuration, setup, and generic coding with a prescribed set of base classes for providers to implement, connection string setup and handling, and factory classes to standardize obtaining a provider-specific Connection, Command, and so on. These features make it easier for software companies that support running on the user's choice of database, as well as programmers who think that the database might change (from Microsoft Access to Microsoft SQL Server, for example) during the lifetime of the project.

Base Classes

New features have been added to the "base provider profile" with each release of ADO.NET. Version 1.1 added a property to the DataReader to allow the programmer to determine if the DataReader contained a nonzero number of rows (the **HasRows** property) without calling **IDataReader.Read()**. Because interfaces are immutable, this property could not just be added to the **IDataReader** interface; this new property had to be added to each provider-specific class (i.e., **SqlDataReader**, **OleDbDataReader**, etc.). The usual way to use versioning with interfaces is to define a new interface. This new interface could either extend the original interface (i.e., IDataReader2 inherits from IDataReader and adds HasRows) or include only the new functionality (i.e., IDataReaderNewStuff contains only HasRows).

Another way to version functionality is to use base classes. Although using base classes restricts the class inheritance model (each class can only inherit from a single base class in .NET), this is the preferred mechanism if you expect changes to your set of functionality in the future. Lots of new functionality is included in ADO.NET 2.0, and so the model uses base classes instead of interfaces. The ADO.NET 1.0/1.1 interfaces are retained for backward compatibility, however. To use our original example, ADO.NET 2.0 contains a DbDataReader class that includes a HasRows property. Additional properties, methods, and events can be added in the future. The only ADO.NET 1.0/1.1 provider class to be based on the base-class concept was the DataAdapter; all provider-specific base classes, such as SqlDataAdapter, derived from DbDataAdapter, which derived from DataAdapter. In ADO.NET 2.0, all of the data classes derive from base classes. The new ADO.NET provider base classes are listed in Table 2. These classes are defined in **System.Data.Common**. All new generic functionality is included in the base classes (but not the interfaces) in ADO.NET 2.0.

Table 2. Generic base classes and Generic interfaces in ADO.NET 2.0

SqlClient class	**Base class**	**Generic interface**
SqlConnection	DbConnection	IDbConnection
SqlCommand	DbCommand	IDbCommand
SqlDataReader	DbDataReader	IDataReader/IDataRecord
SqlTransaction	DbTransaction	IDbTransaction
SqlParameter	DbParameter	IDbDataParameter
SqlParameterCollection	DbParameterCollection	IDataParameterCollection
SqlDataAdapter	DbDataAdapter*****	IDbDataAdapter
SqlCommandBuilder	DbCommandBuilder	
SqlConnectionStringBuilder	DbConnectionStringBuilder	
SqlPermission	DBDataPermission*****	

***** These base classes existed in ADO.NET 1.0.

In addition to these "main" base classes, many new base classes were added in ADO.NET 2.0, including some that we'll be talking about later in this article. The provider base classes in ADO.NET are abstract, however, meaning that they can't be instantiated directly. Our interface-based code above would change to:

```
enum provider {sqlserver, oracle, oledb, odbc};
public DbConnection GetConnectionBaseClass()
```

```
{
// determine provider from configuration
provider prov = GetProviderFromConfigFile();
DbConnection conn = null;
switch (prov) {
  case provider.sqlserver:
    conn = new SqlConnection(); break;
  case provider.oracle:
    conn = new OracleConnection(); break;
 // add new providers as the application supports them
 }
return conn;
}
```

It isn't much of an improvement from a programmer usability standpoint, so next I will discuss the provider factories.

Provider Factories

Rather than rely on the case statements above, it would be nice to have a class that gave out a **DbConnection** based on instantiating "the correct" provider-specific connection. But how to know whether to instantiate **SqlConnection** or **OracleConnection**? The solution to this is to use a Provider Factory class to give out the right type of concrete class. Each provider implements a provider factory class, e.g., **SqlClientFactory**, **OracleClientFactory**, and **OleDbFactory**. These classes all derive from **DbProviderFactory** and contain static methods (shared in Visual Basic .NET) to distribute classes that can be created. Here's the list of DbProviderFactory methods:

Table 3. DbProviderFactory Methods

CreateConnection
CreateCommand
CreateCommandBuilder
CreateConnection
CreateConnectionStringBuilder
CreateDataAdapter
CreateDataSourceEnumerator
CreateParameter
CreatePermission

Note that we don't need a CreateDbDataReader class, for example, because **DbDataReader** isn't directly created with the "new" operator; instead it is always created via **DbCommand.ExecuteReader()**.

Each data provider that exposes a DbProviderFactory-based class registers configuration information in machine.config. Data providers can also be added to program-specific configuration files like web.config. A typical entry in machine.config for the SqlClient data provider looks like this:

```
<system.data>
 <DbProviderFactories>
<add name="SqlClient Data Provider"
 invariant="System.Data.SqlClient"
 support="FF"
 description=".Net Framework Data Provider for SqlServer"
 type="System.Data.SqlClient.SqlClientFactory, System.Data,
  Version=2.0.3600.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />
 <!-- other provider entries elided -->
 </DbProviderFactories>
</system.data>
```

Notice that because the **DbProviderFactories** element uses **<add>** sub-elements, you can <add> or **<remove>** providers to override machine.config. In my sample code, I've used <remove> to remove the SqlServer CE provider, because it wasn't installed on the system I was working on. One of the samples includes a sample provider that can be added in at the machine.config level, or in the application or Web-specific configuration file. The provider itself must be installed in the Global Assembly Cache (GAC) or in the working directory of the application for DbProviderFactories to work with it.

Although some of the attributes in the config files can be used for displaying a list of available data providers by "friendly name", the two most important attributes are **invariant** and **support**. Invariant is the provider-invariant name that can be passed as string to the DbProviderFactories class mentioned in the next paragraph. Support is a bitmask of the types that the DbProviderFactory class can create using the DbProviderFactory.CreateXXX methods described above.

The DbProviderFactories class, which could be thought of as "a factory for factories", has static methods to accommodate choosing a provider and instantiating its DbProviderFactory. These are listed in Table 4.

Table 4. DbProviderFactories methods

DbProviderFactories Method	Purpose
GetFactoryClasses()	Returns a DataTable of provider information from the information in machine.config

GetFactory(DataRow)	Returns the correct DbProviderFactory instance given a DataRow from the DataTable produced by GetFactoryClasses
GetFactory(string)	Returns the correct DbProviderFactory instance given a provider-invariant name string that identifies the provider

The methods of DbProviderFactories could be used to produce a drop-down list box with a list of .NET data providers (via **GetFactoryClasses**), wherein the user chooses one and instantiates the correct DbProviderFactory via **GetFactory(DataRow)**. Or you can just get the correct DbProviderFactory with a string. So now, if we choose to prompt the user for a provider to use, our code to select a provider and instantiate a DbConnection looks like this:

```
public DbConnection GetConnectionBaseClass2()
{
DbProviderFactory f = GetProviderFactoryFromUserInput();
// if our factory supports creating a DbConnection, return it.
if ((f.SupportedClasses & DbProviderSupportedClasses.DbConnection) > 0)
  return f.CreateConnection();
else
  return null;
}

public DbProviderFactory GetProviderFactoryFromUserInput()
{
DataTable t = DbProviderFactories.GetFactoryClasses();
DataRow r;
// user selects a DataRow r from DataTable t
r = t.Rows[0]; // row 0 is the selection DataRow
return DbProviderFactories.GetFactory(r);
}
```

Once we have the appropriate DbProviderFactory, we can instantiate any of the provider-supported classes that can be created—the DbConnection is just the most common example.

This takes care of allowing the user to configure a provider, but this is usually done once, during setup of a particular application. After setting up the database and choosing a database instance, the information should be stored in a user's config file and only changed if the company changes database products, or if the location of the database instance changes (i.e., the application moves from a test server to a production server). ADO.NET 2.0 accommodates both of these with built-in functionality; you don't have to roll your own.

Specifying Configuration Information

In addition to the DbProviderFactories section of configuration files, there is also a standard config file location and APIs for manipulating connection string information. This requires a little more handling than you might expect. OLE DB/ADO, ODBC, and JDBC connection strings actually contain two discrete types of information: information identifying the provider/driver and name/value pairs of information describing how to connect to the provider. Here's a typical ADO (classic) connection string:

```
"provider=sqloledb;data source=mysvr;integrated security=sspi;initial catalog=pubs"
```

When the Visual Basic 6.0 programmer codes the statement:

```
Dim conn as New Connection
conn.Open connstr 'use the connection string defined above
```

The connection string information is used by the OLE DB service components to:

1. Search the registry and load the correct provider by calling **CoCreateInstance** on the OLE DB data source class for the provider name specified.
2. Create an ADO Connection instance that encapsulates the OLE DB Data Source (and Session) objects.
3. Return the correct ADO interface pointer (**_Connection** in this case) to the program.
4. Call **Open()** on the ADO _Connection interface, which calls the underlying OLE DB provider code using OLE DB interfaces.

ADO.NET connection strings do not contain provider information. Programmers must decide which provider to use at program coding time (and use the SqlConnection, for example) or invent a mechanism to store that information. ADO.NET 2.0 connection configuration information takes care of this by including provider information. A typical ADO.NET 2.0 connection string in an app.config file looks like this:

```
<configuration>
 <connectionStrings>
    <add name="Publications" providerName="System.Data.SqlClient"
      connectionString="Data Source=MyServer;Initial Catalog=pubs;
                        integrated security=SSPI" />
 </connectionStrings>
</configuration>
```

... and contains a connection string name and a **providerName**, as well as the string itself.

This information is exposed through the **ConnectionStringsSettingsCollection** collection class of **ConnectionStringSettings**. You can fetch a connection string by name and retrieve all of the needed information with very little code. Using the connection string information above, my generic DbConnection fetching program becomes:

```
public DbConnection GetInitializedConnectionBaseClass()
{
DbConnection conn = null;
ConnectionStringSettings s =
  ConfigurationSettings.ConnectionStrings["Publications"];
DbProviderFactory f = DbProviderFactories.GetFactory(
  s.ProviderName);

if ((f.SupportedClasses & DbProviderSupportedClasses.DbConnection) > 0)
 {
  conn =  f.CreateConnection();
  conn.ConnectionString = s.ConnectionString;
 }
return conn;
}
```

Storing named connection string information is just as easy using the **ConnectionStringSettingsCollection**. Using the Configuration classes in .NET works in .NET 2.0 Beta 1, but in later versions the functionality of loading and saving connection strings will likely be built into the provider-specific DbConnectionStringBuilder class. We'll be discussing this class in the next section.

Enumerating Data Sources

Now we can configure and load connection strings and get a connection to the database using ADO.NET 2.0 in a completely provider-independent manner. But what if the database that I'm connecting to changes? The database administrators (DBAs) have moved the data I'm using from the "Publications" instance of SQL Server to the "Bookstore" instance, and I've received e-mail that my configuration should be changed. Although Systems Management Server or another automatic deployment could be used to automate the process of pushing the new configuration, a program could be written to list all of the SQL Server instances on the network and allow me to reconfigure it myself. ADO.NET 2.0 introduces Data Source Enumerator classes just for that purpose.

The ADO.NET data source enumerators derive from a common base class (of course), **DbDataSourceEnumerator**. You can retrieve the appropriate DbDataSourceEnumerator through the DbProviderFactory using the pattern described above. DbDataSourceEnumerator classes expose a single static method, **GetDataSources**. If my database or data source changes, the code to change it would look like this:

```
public void ChangeDataSourceOrProvider()
{
// see this method in the example above
DbProviderFactory f = GetProviderFactoryFromUserInput();
// if our factory supports creating a DbConnection, return it.
if ((f.SupportedClasses &
    DbProviderSupportedClasses.DbDataSourceEnumerator) > 0)
 {
    DbDataSourceEnumerator e = f.CreateDataSourceEnumerator();
    DataTable t = e.GetDataSources();
    // uses chooses a Data Row r
    DataRow r = t.Rows[0];
    string dataSource = (string)r["ServerName"];
    if (r[InstanceName] != null)
      dataSource += ("\\" + r["InstanceName"]);
    // this method is defined below
    RewriteConnectionStringAndUpdateConfigFile(f, dataSource);
 }
else
  Console.WriteLine("Source must be changed manually");
}
```

Note that this was pretty simple generic code, with the exception of the last **RewriteConnectionStringAndUpdateConfigFile** method. Because connection strings are name/value pairs, we'll have to do some string manipulation to ensure that we're replacing the right value. And what if not only the instance of the database changes, but other connection string parameters like the User ID or Initial Catalog (Database) changes as well? It would be nice to have a built-in class to help with this task. ADO.NET 2.0 provides just such a class, the **ConnectionStringBuilder**. Of course, this is based on a generic base class, **DbConnectionStringBuilder**.

Building Connection Strings

In ADO.NET, connection strings are name/value pairs, as with other generic APIs. Unlike OLE DB and ODBC, ADO.NET does not mandate what those name keywords should be, though it is suggested that they follow the OLE DB conventions. The **OleDb** and **Odbc** bridge data providers each follow their own convention. The Microsoft **OracleClient** follows OLE DB convention and **SqlClient** supports either keyword when the OLE DB and ODBC keywords differ. Here are some examples:

Table 5. Connection string names and values in different data providers

Meaning	Odbc	OleDb	SqlClient	OracleClient

Source to connect to	Server	Data Source	Server or Data Source	Server or Data Source
User	UID	User ID	UID or User ID	User ID
Password	PWD	Password	PWD or Password	Password
Is a Windows login used?	Trusted_Connection	Integrated Security	Trusted_Connection or Integrated Security	Integrated Security
Database to connect to	Database	Initial Catalog	Database or Initial Catalog	N/A
Connection Pooling		OLE DB Services	Pooling	Pooling

The DbConnectionStringBuilder is a base class for provider-specific connection string builders. Because it cannot be guaranteed that an ADO.NET data provider supports a specific connection string parameter, the DbConnectionStringBuilder just keeps a dictionary of name/value pairs. All of the ordinary collection methods are available, including a **GetKeys** and **GetValues** method to get the entire list from the dictionary. DbConnectionStringBuilder does support specific properties for connection string parameters, such as **ShouldSerialize** (the user may not want the password value serialized into a file, for example). You can get an instance of the generic DbConnectionStringBuilder through the DbProviderFactory class. Specific ConnectionStringBuilder classes have convenience properties, should as a **DataSource** property that refers to the database server to connect to. In general, the properties are data provider-specific, as are the keywords.

To implement our RewriteConnectionStringAndUpdateConfigFile method mentioned above using the DbConnectionStringBuilder would look like this:

```
public void RewriteConnectionStringAndUpdateConfigFile(
  DbProviderFactory f, string dataSource)
{
  Configuration config = Configuration.GetExeConfiguration(_progname,
     ConfigurationUserLevel.None);
  ConnectionStringSettingsCollection cs =
     config.ConnectionStrings.ConnectionStrings;

  DbConnectionStringBuilder b = f.CreateConnectionStringBuilder();

  b.ConnectionString = cs["Publications"].ConnectionString;
  if (b.ContainsKey("Data Source"))
  {
    b.Remove("Data Source");
    b.Add("Data Source", dataSource);

    // Update ConfigurationSettings connection string
    cs["Publications"].ConnectionString = b.ConnectionString;
    config.Update();
  }
}
```

One last thing to mention about building connection strings: both ODBC and OLE DB provide graphic editors for configuring connection information. The ODBC editor is contained in a Control Panel applet (Configure ODBC Data Sources). The OLE DB editor is invoked whenever a file with the suffix .UDL is double-clicked. The OLE DB UDL editor is used in Visual Studio .NET to configure .NET data providers; the connection strings for SqlClient must be post-processed by Visual Studio itself. Visual Studio 2005 contains a graphic editor that uses the ConnectionStringBuilder and ProviderFactories class. To have a look at it, configure a new data source in Visual Studio 2005 Beta 1 using Server Explorer. It lists the .NET data providers, not the OLE DB providers any more! In the Beta 1 release, this component is tightly coupled with Visual Studio and is not available through public classes yet.

Other Generic Coding Considerations

Using ADO.NET 2.0 common base classes and factories, we can write almost completely generic code. However, databases by their very nature aren't completely generic. Programmers never have been able to write one program that will work on every database and have it perform the same on each one. One prominent example of this is that strategies that work well on client-server databases, such as SQL Server and Oracle, wouldn't work as well on file-based data stores, such as Access and FoxPro. We found out in the ADO era that porting between file-based data stores and client-server (network) databases based involves a little more than changing the Provider and Connection String.

To address data source differences, each data provider may support some properties, methods, and events that are not supported by other data providers. The premise behind the base classes is that they expose generic functionality; provider-specific functionality is exposed on the provider-specific classes. These extras are always available through casting using the cast syntax, "as" or "is" in C# or the **CType** method in Visual Basic .NET. Here's an example of using the SqlConnection-specific method **RetrieveStatistics** from C# code.

```
public void GetStatsIfPossible(DbConnection conn)
{
  if (conn is SqlConnection)
    Hashtable h = ((SqlConnection)conn).RetrieveStatistics();
}
```

Another place where providers differ is in their usage of parameters, in parameterized queries or stored procedures. In ADO.NET, data providers can either support named parameters (the parameter name is significant, the order in which parameters are specified is irrelevant) or positional parameters (the parameter order is significant, the name is irrelevant), or both. The SqlClient and the Microsoft OracleClient provider insist on named parameters; OleDb and Odbc use positional parameters. Another difference is parameter markers if you are using parameterized queries. Each provider has its own idea of what the parameter marker should be. Here's a short list of differences, using the providers that ship with ADO.NET as an example. If you obtain providers from DataDirectTechnologies or Oracle Corporation, these can differ.

Table 6. Parameter usage styles in different ADO.NET data providers

Provider	**Named/Positional**	**Parameter Marker**
SqlClient	Named	@parmname
OracleClient	Named	:parmname (or parmname)
OleDb	Positional	?
Odbc	Positional	?

Other differences are based on how the database's network protocol works. The network protocol defines the way that result packets are returned from database to client: SQL Server uses the TDS (tabular data stream) protocol, and Oracle uses TNS (transparent network substrate), for example. Because of how the database and protocol works, versions of SQL Server prior to SQL Server 2005 can only have one active resultset (DataReader, in ADO.NET terms) at a time. SQL Server 2005 does not have this limitation; the feature that permits multiple active resultsets on a single connection is known as MARS. In SQL Server, output parameters are only available after resultsets are returned and the DataReader is closed. Oracle, on the other hand, returns resultsets from stored procedures as a special type of output parameter, **REFCURSOR**, and has different behaviors with respect to DataReaders and output parameters. Nothing is completely generic.

Wrap-up

With the introduction of the ProviderFactory, DataSourceEnumerator, and ConnectionStringBuilder classes, ADO.NET 2.0 makes it easy to switch between providers at configuration time or at runtime. There are helper classes to help configure connection information, and it is stored in a standard format in the configuration files. Base classes in **System.Data.Common** make it easy to add functionality to the base provider profile without adding new interfaces in future releases of ADO.NET. On my Web site, I've posted an updated version of my "world's simplest" ADO.NET data provider that uses base classes and implements a provider factory. See my *MSDN Magazine* article, ADO.NET: Building a Custom Data Provider for Use with the .NET Data Access Framework for information about the original provider.

There is one provider-specific item we didn't cover relating to how a database and a provider expose database metadata (the **INFORMATION_SCHEMA** in SQL-92/99). Each database has a slightly different way to expose lists of information about tables, columns, primary keys, and so on. These may even vary between versions of the same database product. In the next article of this series, we'll look at the mechanism that ADO.NET 2.0 provides to expose metadata in a database/version independent way. See you then.

Bob Beauchemin is an instructor, course author, and database curriculum course liaison for DevelopMentor. He has over twenty-five years of experience as an architect, programmer, and administrator for data-centric distributed systems. He's written articles on ADO.NET, OLE DB, and SQL Server for Microsoft Systems Journal and SQL Server Magazine and others, and is the author of A First Look at SQL Server 2005 for Developers *and* Essential ADO.NET.

Top of Page