CHAPTER 2

ADO.NET from 3,048 meters

It is a rare opportunity to get to build something from scratch. When Microsoft chose the approach for data access from .NET, the company could have gone a number of different directions. One possibility would have been to wrap OLE DB, or provide a slightly updated version of ADO (which, of course, many people believe ADO.NET to be). However, the world has changed since these older technologies were developed, and they didn't necessarily have the same goals as .NET. This chapter talks about the goals of ADO.NET, and then provides a high-level overview of its major pieces. For those not familiar with the metric system, 3,048 meters is the 10,000-foot view. Much of the rest of the book is dedicated to exploring these pieces in great detail.

2.1 THE GOALS OF ADO.NET

Rather than just putting a new face on an older technology, Microsoft took the opportunity to develop a new access mechanism that learned from previous technologies, but was a better fit for the .NET world.

The new access technology had a number of goals:

- *Multi-tier*—Although you can use most previous Microsoft data access technologies in a multi-tier environment, it is usually quite difficult to move code from, for example, a 2-tier client/server application to a true 3-tier application. ADO.NET was built from the ground up to allow code to be used on different tiers easily without having to be significantly modified.
- *Disconnected*—Part of this multi-tier support is based on the fact that ADO.NET assumes a temporary connection to the database. Data is read from the database and then can be transported and changed without touching the database. Only when the system needs to commit the changes to the database must the system touch the database again.
- *XML-based*—It is probably no surprise that, like much of the technology coming from Microsoft today, ADO.NET is steeped in XML. XML is used to transport data between tiers; it is also very easy to convert data back and forth between relational data and XML data.
- *Scalable*—ADO.NET, coupled with other features of .NET, makes it easy to build stateless applications, which are generally far more scalable than applications that remember information on the server about each user. You can also build applications that maintain state fairly efficiently, which, although not quite as scalable, *can* provide better performance.
- *Fast*—It is exceedingly important to Microsoft that applications written using ADO.NET (and .NET in general) perform well, in order for the technology to become accepted. For this reason, a lot of effort has gone into performance tuning. It is especially advantageous when used with SQL Server, because the SQL Server data provider can talk natively to the engine. In fact, the SQL Server ADO.NET provider has shown itself to be 10 to 15 percent faster than the native OLE DB provider in some of Microsoft's early tests.

One thing is something of a departure from some of the previous technologies. Although you can use ADO.NET to talk to a variety of different back-end databases (SQL Server, Oracle, and so forth), ADO.NET has not focused to the same extent on allowing movement between different engines without changing code.

That doesn't mean you can't write code that is *fairly* generic, but it is not the highest priority of ADO.NET. I would say there are several reasons behind Microsoft's direction here:

- Providing a thinner layer between the caller and the database engine provides performance advantages, and the caller is free to take advantage of the special features offered by the engine.
- Most development is done against a single engine, and providing overly generic functionality can easily complicate what would otherwise be simple. OLE DB is an excellent example of this problem.

- Microsoft was working against an aggressive schedule to make the first release of .NET. Presumably, later versions of ADO.NET will provide more functionality for gathering information about the engine, and so forth.

That all being said, you may notice that a number of sections within this book talk about building generic code that can be swapped between engines fairly easily. I do this for two reasons:

- I assume that a fair number of developers want to support different engines easily. I know that the current project on which I am working, written using .NET and ADO.NET, is designed to work with a large number of different back ends.
- It is not uncommon for a decision to be made to switch to a different database without necessarily considering the impact on existing code. By remaining fairly generic, that impact can be significantly reduced.

2.2 ZOOMING IN ON ADO.NET

Although a number of different classes make up ADO.NET, the pieces can be broken down into several fairly distinct components, as shown in figure 2.1.

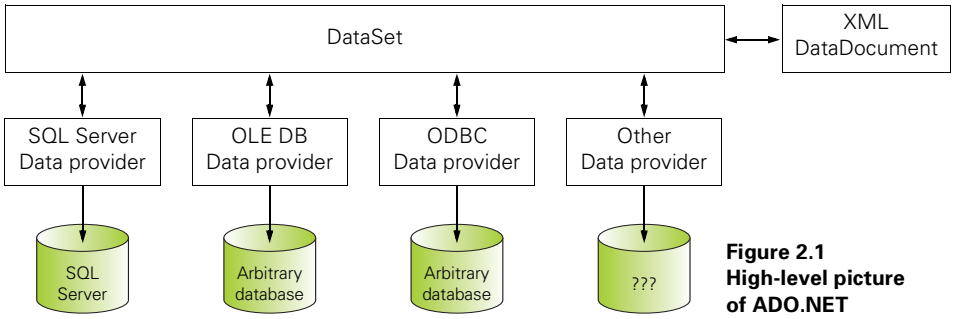

Figure 2.1 High-level picture of ADO.NET

2.2.1 .NET data providers

Just as with OLE DB, ADO.NET has data providers for each targeted engine. A data provider wraps all the functionality required to talk to a particular database engine or data access technology. Version 1 of .NET includes two data providers:

- *SQL Server data provider*—As the name implies, this data provider is specific to the SQL Server engine (version 7.0 or higher). It also works with MSDE, the lower-end version of SQL Server.
- *OLE DB data provider*[1]—Just as OLE DB provided a wrapper for ODBC to ease transitioning, so ADO.NET provides a wrapper for OLE DB. This provider is used to talk to engines that have OLE DB drivers.

A third data provider has also been created, which wraps ODBC. It did not ship with the first release of .NET, but is available for download from the Microsoft web site. This data provider gives you access to hundreds of existing ODBC drivers. In addition, it is assumed that other engines such as Oracle, DB2, and Sybase will have native .NET data providers over time. These data providers will most likely be built by the vendors of those engines, rather than by Microsoft.

Each data provider contains a number of objects to provide various capabilities, such as connections and commands. Like OLE DB, these objects derive from specific interfaces; but unlike OLE DB, the objects can be used directly, and different data providers can have substantially different functionality.

Figure 2.2 shows the major objects in each data provider.

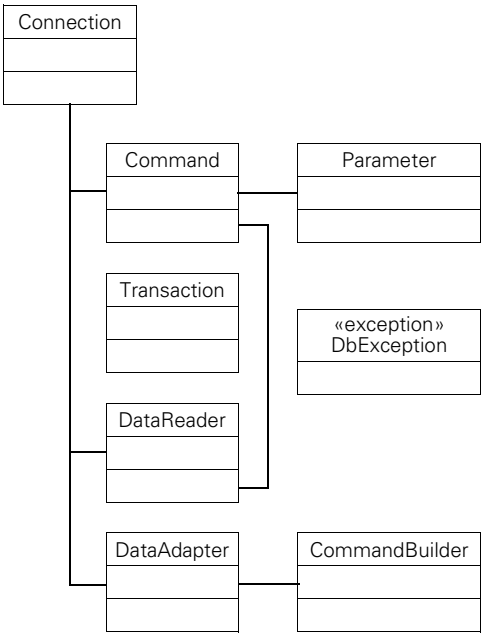

Figure 2.2
Principal objects within a data provider

No classes in ADO.NET have the *exact* names shown in figure 2.2. Usually, though, an interface presents the general functionality and a class that implements that interface within each separate data provider. For example:

- *IDbCommand*—The interface that provides the functionality for a database command, such as an `Update` or a `Select`. It exists in the `System.Data` namespace. An interface defines a series of properties and methods that must exist in any object that *implements* the interface. If an object implements an interface, it is derived from that interface, and can be cast to that interface.

[1] Quite some confusion will probably exist between OLE DB *data providers* and the ADO.NET OLE DB *data provider*!

- *SqlCommand*—The SQL Server–specific object for commands, which implements the IDbCommand interface. It exists in the `System.Data.SqlClient` namespace.
- *OleDbCommand*—The OLE DB–specific object for commands, which implements the IDbCommand interface. It exists in the `System.Data.OleDb` namespace.

The objects in figure 2.2 are all discussed in detail in chapters 5 through 14. The following is a brief description of the purpose of each object:

- *Connection*—A connection, as the name implies, wraps the concept of a connection with a database. You must have a connection before you can execute commands and read data from the database.
- *Command*—A command represents an instruction being sent to the database, usually in SQL, such as an `Insert`, `Update`, `Delete`, `Select`, or the name of a stored procedure. Commands can either return data (such as from a `Select` command) or just a changed row-count for operations like an `Update`.
- *Parameter*—You can specify parameters to a command that need to be "inserted" appropriately by the database engine. This functionality is provided by the Parameter class. In addition, a ParameterCollection object holds a collection of parameters.
- *Transaction*—Rather than just tying transactions to a connection, as with earlier data access technologies, ADO.NET separates out the concept of a transaction and allows it to be associated with some number of commands. This provides much finer control.
- *DataReader*—The DataReader is like a forward-only cursor into a result set. For example, if a command is used to select a number of rows, it is possible to move through each row, one at a time, and retrieve the data from that row. The DataReader is a fast access mechanism for retrieving data.
- *DataAdapters*—DataAdapters are primarily designed to provide a liaison between a data provider and a DataSet. DataSets are discussed in more detail in the next section, but they are basically independent collections of data. To maintain that independence, they cannot know anything specific about how they get their data. A DataAdapter allows for a DataSet to be filled from a database query.
- *CommandBuilder*—When a DataSet needs to write its changes back to a database, it does so using the DataAdapter. The DataAdapter must know how to do `Deletes`, `Updates`, and `Inserts`, and this functionality can be customized. However, for *simple* applications, you can use the CommandBuilder to provide default commands for these operations. The commands that are generated, though, are not high-performance and can have other problems.
- *DbException*—As with most of .NET, when an error occurs, an exception is thrown that must be caught. This includes data errors (for example, if a `Select` references a nonexistent column).

2.2.2 The DataSet

At the most fundamental level, a DataSet is simply a container for data. However, it is a very elaborate container that provides a large amount of functionality for working with the data it holds. It is conceptually similar to an ADO Recordset, but with several key differences:

- A DataSet can contain more than one table. A DataSet contains a collection of DataTables (discussed later), each of which represents a single table.
- A DataSet is not tied in any way to a database or data table. Although it is possible (and quite likely) that a DataSet will be filled by the results of database queries, the DataSet itself is just the container for the data. You certainly can fill a DataSet manually. This ability is quite important, because it allows DataSets to operate in a completely disconnected manner.
- When the data within a DataSet is changed, those changes are not automatically made to the underlying database. Rather, the DataSet keeps track of the changes—`Inserts`, `Updates`, and `Deletes`. Only when a DataSet is given to a DataAdapter to update the database are the changes actually written. This behavior is not unlike a disconnected ADO Recordset.
- You can query against a DataSet in a number of ways, including having it execute simple SQL statements without touching the database at all.
- DataSets can be responsible for data integrity, preventing illegal data from being put in place, again without touching the underlying database.
- DataSets can very easily be converted back and forth to XML. This feature is especially useful because .NET uses this mechanism to transport DataSets between tiers. A client can request a DataSet object from a server, and the server will just return that DataSet object. .NET will automatically convert it to XML, and then convert it back into a DataSet on the client.

This process is more efficient than disconnected ADO Recordsets. The main reason is that, because ADO uses COM, every element in the Recordset (every field in every row) must be marshaled into a COM type, and later marshaled back. With a DataSet, only a string is being sent. There is some overhead in converting to and from XML, but that code is highly optimized.

DataSets have quite a lot of functionality. Figure 2.3 shows the main classes that make up a DataSet.

Although these classes are discussed in detail in chapters 15 through 20, here is a brief explanation of each of the objects:

- *DataSet*—This is the principal class; but, perhaps surprisingly, it's little more than a container for the other objects. It does, however, have some properties that control how the other objects operate.

- *DataTable*—A DataTable represents a single table's results or, more accurately, represents the results of a single query. The distinction is that `Select`s that cross table boundaries (for example, by using a join), will still end up in a single DataTable. Think of a DataTable as the result set from a single `Select` statement.
- *DataColumn*—The DataColumn contains information about the column of results, which—at least if the DataSet is filled from a database—is synonymous with a field. Information in a DataColumn includes a name, data type, and length. You can also calculate a DataColumn's value using an expression.
- *DataRow*—There is a DataRow for each row of data in the DataTable. You can ask the DataRow for the value in a particular column. To add data to a DataTable, a new DataRow is created and added. A DataRow can also be changed or deleted.
- *DataRelation*—A DataRelation relates two different DataTables to one another. For example, consider a `Teacher` DataTable and a `Classes` DataTable. Both tables have a `TeacherID` field, so the two tables can be related via that field.
- *Constraint*—Constraints are used to prevent certain types of changes to the data in a DataSet. There are currently two different types of constraints: the ForeignKeyConstraint enforces that a value entered into a field in one table *must* exist in a column in another table, and the UniqueConstraint enforces that a value entered into a field is unique within that field for all records in the DataSet.

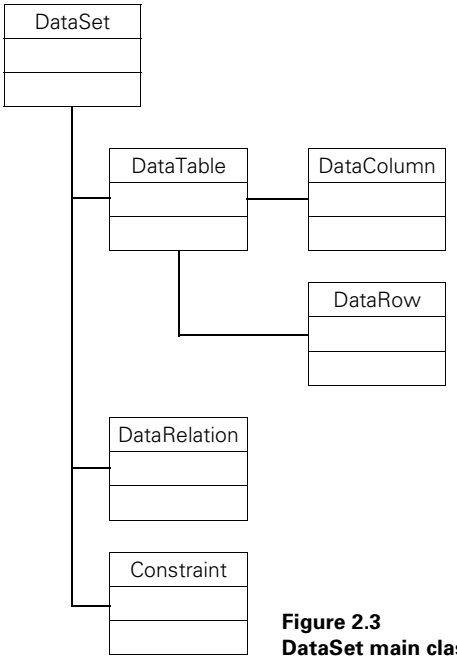

Figure 2.3
DataSet main classes

A DataSet must have all this functionality (which somewhat duplicates the database) because it must be self-sufficient. All the work is done on a DataSet without any connection to the data source.

As changes are made to the data, the DataSet keeps track. The fact that it does this means you can roll back changes, as well. At some point, all the changes will be written to the database (or somewhere), at which time the DataSet can be told to incorporate all the changes (or roll them all back).

How much data in a DataSet?

You may have noticed that much of the functionality of a DataSet relies on it containing *all* the appropriate data. For example, a UniqueConstraint would not be very useful if it was being compared to only half the data in the actual table.

Because the DataSet has no connection to the database, it has no way of looking at data it does not contain. This simplifies things, especially in a multi-tier model, because it means the client has all the information it needs. But it can also cause problems when you have a lot of data. Imagine a customer list containing a million customers (or even 100,000 customers). It would be impractical to read them in to a DataSet, let alone to transmit that DataSet across the wire to a client, even if that client had a fairly good connection and was not dialed in at modem speeds.

Of course, the data that fills the DataSet can easily be limited based on a criterion (an SQL `where` clause), so that only an appropriate amount of data will be sent. The problem is that then the client is only dealing with a subset of the data and cannot rely on the built-in functionality (or any client-based functionality) to enforce constraints such as the UniqueConstraint.

There are ways around these problems, but they largely come down to design decisions in your application.

2.3 SUMMARY

I am one of those people who really wants to understand the reason behind the way things work, as well as understand how to *make* them work. There are a couple of reasons—first of all, I am naturally nosy; and second, it has saved me a lot of pain trying to use something in a way other than intended (the mental equivalent of trying to use a can opener as a hammer)!

It is not that I am adverse to stretching technologies and using them in ways never dreamt of by their designers;[2] but all things being equal, I generally try to begin by seeing if the technology I plan to use is really designed to work the way I intend to use it. If nothing else, I have a better chance of finding useful documentation.

[2] I once spent an hour on the phone with an ODBC engineer explaining how our (shipping) application worked, only to be told that ODBC couldn't be used that way!

That is the reason why I spent so much time in this chapter discussing the philosophies behind ADO.NET. Understanding the disconnected model might, I hope, prevent you from trying to use a DataSet as a front-end to a live database cursor (or at least make you realize that making it happen will hurt a lot). The other goals of ADO.NET are important, but assuming the disconnected data model from the start requires a fundamental design shift from older technologies.

That is not to say the breakdown of the technology is not important, although the decisions for that breakdown make more sense when you understand the underlying philosophy. For example, the fact that the DataSet is completely disconnected from the data providers is a good coding practice, but it's critical to the disconnected model.

Like OLE DB and COM, ADO.NET is interface based, and interfaces exist for all the important pieces of a data provider—IDbConnection, IDbCommand, and so forth. One of the great things about .NET versus COM is that you can get beyond these interfaces to provide additional functionality. This ability is cool for two reasons—first, the interfaces do not have to be loaded with all sorts of arbitrary functionality that is specific to one or two implementers; and second, data provider *providers* don't have to jump through all sorts of hoops to provide extra functionality.

If you don't fight the disconnected model, and instead choose to embrace it, then ADO.NET gives you a great tool for simplifying your life—the DataSet, which is actually a collection of classes working together to provide database-like functionality without the database. That is not to say the DataSet is a panacea—there are always trade-offs you must understand, such as knowing how much data it's practical to send over the wire. However, it is a great tool in a lot of situations.

Chapter 3 will talk about working with data and XML; although not strictly part of ADO.NET, XML is still very much related to working with data. As you will see, ADO.NET has a lot of tools for interacting with XML.