

Using ADO.NET Databases

With the advent of .NET, Microsoft decided to update ActiveX Data Objects (ADO) and created ADO.NET. ADO.NET contains several enhancements over the original ADO architecture, providing improved interoperability and performance. If you are already familiar with ADO, you will notice that the object model of ADO.NET is a little different. For one thing, the RecordSet type no longer exists - Microsoft has created the DataAdapter and DataSet classes that support disconnected data access and operations, allowing greater scalability because you no longer have to be connected to the database all the time. (To be fair, ADO also provided disconnected RecordSets, but they were the exception rather than the rule when used by programmers.) Therefore, your applications can consume fewer resources. With the connection pooling mechanisms of ADO.NET, database connections can be reused by different applications, thereby reducing the need to continually connect to and disconnect from the database, which can be a time-consuming operation.

Visual Studio .NET provides access to databases through the set of tools and namespaces collectively referred to as Microsoft ADO.NET. Data access in ADO.NET is standardized to be mostly independent of the source of the data - once you've established a connection to a database, you use a consistent set of objects, properties, and methods, regardless of the type of database you are using.

Comparison of ADO.NET and ADO: these are following comparison:

In-memory Representations of Data: In ADO, the in-memory representation of data is the RecordSet. In ADO.NET, it is the dataset. There are important differences between them.

Number of Tables: A RecordSet looks like a single table. If a RecordSet is to contain data from multiple database tables, it must use a JOIN query, which assembles the data from the various database tables into a single result table. In contrast, a dataset is a collection of one or more tables. The tables within a dataset are called data tables; specifically, they are DataTable objects. If a dataset contains data from multiple database tables, it will typically contain multiple **DataTable** objects. That is, each **DataTable** object typically corresponds to a single database table or view. In this way, a dataset can mimic the structure of the underlying database. A dataset usually also contains relationships. A relationship within a dataset is analogous to a foreign-key relationship in a database —that is, it associates rows of the tables with each other. For example, if a dataset contains a table about investors and another table about each investor's stock purchases, it could also contain a relationship connecting each row of the investor table with the corresponding rows of the purchase table.

Data Navigation and Cursors: In ADO you scan sequentially through the rows of the RecordSet using the ADO MoveNext method. In ADO.NET, rows are represented as collections, so you can loop through a table as you would through any collection, or access particular rows via ordinal or primary key index. A **cursor** is a database element that controls record navigation, the ability to update data, and the visibility of changes made to the database by other users. ADO.NET does not have an inherent cursor object, but instead includes data classes that provide the functionality of a traditional cursor. For example, the functionality of a forward-only, read-only cursor is available in the ADO.NET DataReader object.

Minimized Open Connection: In ADO.NET you open connections only long enough to perform a database operation, such as a Select or Update. You can read rows into a dataset and then work with them without staying connected to the data source. In ADO the RecordSet can provide disconnected access, but ADO is designed primarily for connected access. There is one significant difference between disconnected processing in ADO and ADO.NET. In ADO you communicate with the database by making calls to an OLE DB provider. In ADO.NET you communicate with the database through a data adapter (an OleDbDataAdapter, SqlDataAdapter, OdbcDataAdapter, or OracleDataAdapter object), which makes calls to an OLE DB provider or the APIs provided by the underlying data source. The important difference is that in ADO.NET the data adapter allows you to control how the changes to the dataset are transmitted to the database — by optimizing for performance, performing data validation checks, or adding any other extra processing.

Sharing Data between Applications: Transmitting an ADO.NET dataset between applications is much easier than transmitting ADO disconnected RecordSet. To transmit ADO disconnected RecordSet from one component to another, you use COM marshalling. To transmit data in ADO.NET, you use a dataset, which can transmit an XML stream.

Richer data types: COM marshalling provides a limited set of data types — those defined by the COM standard. Because the transmission of datasets in ADO.NET is based on an XML format, there is no restriction on data types. Thus, the components sharing the dataset can use whatever rich set of data types they would ordinarily use.

Performance: Transmitting a large ADO RecordSet or a large ADO.NET dataset can consume network resources; as the amount of data grows, the stress placed on the network also rises. Both ADO and ADO.NET let you minimize which data is transmitted. But ADO.NET offers another performance advantage, in that ADO.NET does not require data-type conversions. ADO, which requires COM marshalling to transmit records sets among components, does require that ADO data types be converted to COM data types.

Penetrating Firewalls: A firewall can interfere with two components trying to transmit disconnected ADO RecordSets. Remember, firewalls are typically configured to allow HTML text to pass, but to prevent system-level requests (such as COM marshalling) from passing. Because components exchange ADO.NET datasets using XML, firewalls can allow datasets to pass.

ADO.NET Definition:

ADO.NET is a set of classes that you use to connect to and manipulate data sources. Unlike ADO, which relies on connections, uses OLE DB to access data, and is COM-based; ADO.NET is specifically designed for data-related connections in a disconnected environment, thereby making it the perfect choice for Internet-based Web applications. ADO.NET uses XML as the format for transmitting data to and from the database and your Web application, thereby ensuring greater compatibility and flexibility than ADO.

It provides:

- An evolutionary, more flexible successor to ADO
- A system designed for disconnected environments
- A programming model with advanced XML support
- A set of classes, interfaces, structures, and enumerations that manage data access from within the .NET Framework

Understanding ADO.NET

There are three layers to data access in ADO.NET:

- **The physical data store.** This can be an OLE database, a SQL database, or an XML file.
- **The data provider.** This consists of the Connection object and command objects that create the in-memory representation of the data.
- **The data set.** This is the in-memory representation of the tables and relationships that you work with in your application.

The data provider layer provides abstraction between the physical data store and the data set you work with in code. After you've created the data set, it doesn't matter where it comes from or where it is stored. This architecture is referred to as disconnected because the data set is independent of the data store.

ADO.NET provides the following advantages over other data access models and components:

Interoperability: ADO.NET uses XML as the format for transmitting data from a data source to a local in-memory copy of the data.

Maintainability: When an increasing number of users work with an application, the increased use can strain resources. By using n-tier applications, you can spread application logic across additional tiers. ADO.NET architecture uses local in-memory caches to hold copies of data, making it easy for additional tiers to trade information.

Programmability: The ADO.NET programming model uses strongly typed data. Strongly typed data makes code more concise and easier to write because Microsoft Visual Studio® .NET provides statement completion.

Performance: ADO.NET helps you to avoid costly data type conversions because of its use of strongly typed data.

Scalability: The ADO.NET programming model encourages programmers to conserve system resources for applications that run over the Web. Because data is held locally in in-memory caches, there is no need to retain database locks or maintain active database connections for extended periods.

ADO.Net Namespaces:

ADO.NET provides its objects, properties, and methods through the three name-spaces described in below table. The System.Data.SqlClient and System.Data.OleDb namespaces listed in the table provide equivalent features for SQL and OLE databases, respectively.

Namespace	Provides
System.Data	Classes, types, and services for creating and accessing data sets and their subordinate objects
System.Data.SqlClient	Classes and types for accessing Microsoft SQL Server databases
System.Data.OleDb	Classes and types for accessing OLE databases

New Features of ADO.NET 2.0

1. Bulk Copy Operation

Bulk copying of data from a data source to another data source is a new feature added to ADO.NET 2.0. Bulk copy classes provide the fastest way to transfer set of data from once source to the other. Each ADO.NET data provider provides bulk copy classes. For example, in SQL .NET data provider, the bulk copy operation is handled by **SqlBulkCopy** class, which can read a DataSet, DataTable, DataReader, or XML objects.

2. Batch Update

Batch update can provide a huge improvement in the performance by making just one round trip to the server for multiple batch updates, instead of several trips if the database server supports the batch update feature. The **UpdateBatchSize** property provides the number of rows to be updated in a batch. This value can be set up to the limit of decimal.

3. Data Paging

Now command object has a new execute method called **ExecutePageReader**. This method takes three parameters - **CommandBehavior**, **startIndex**, and **pageSize**. So if you want to get rows from 101 - 200, you can simply call this method with start index as 101 and page size as 100.

4. Asynchronous Data Access

In ADO.NET 1.x commands like ExecuteReader, ExecuteScalar and ExecuteNonQuery will synchronously execute and block the current thread. Even when you open connection to the database, current thread is blocked. But in ADO.NET 2.0, all of these methods come with Begin and End methods to support asynchronous execution. This asynchronous methodology is very similar to our .NET framework asynchronous methodology. Even you can have callback mechanism using this approach. This Asynchronous Data Access is currently only supported in SQLClient, but complete API support is available for other providers to implement this mechanism.

5. Connection Details

Now you can get more details about a connection by setting Connection's **StatisticsEnabled** property to True. The Connection object provides two new methods - **RetrieveStatistics** and **ResetStatistics**. The **RetrieveStatistics** method returns a HashTable object filled with the information about the connection such as data transferred, user details, curser details, buffer information and transactions.

6. DataSet.RemotingFormat Property

When **DataSet.RemotingFormat** is set to binary, the DataSet is serialized in binary format instead of XML tagged format, which improves the performance of serialization and **deserialization** operations significantly.

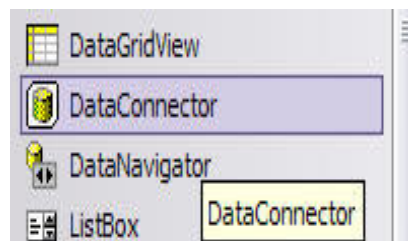
7. DataTable's Load and Save Methods

In previous version of ADO.NET, only DataSet had Load and Save methods. The Load method can load data from objects such as XML into a DataSet object and Save method saves the data to a persistent media. Now DataTable also supports these two methods.

You can also load a DataReader object into a DataTable by using the Load method.

8. New Data Controls

In Toolbox, you will see these new controls - **DataGridView**, **DataConnector**, and **DataNavigator**. Now using these controls, you can provide navigation (paging) support to the data in data bound controls.



9. DbProvidersFactories Class

This class provides a list of available data providers on a machine. You can use this class and its members to find out the best suited data provider for your database when writing a database independent application.

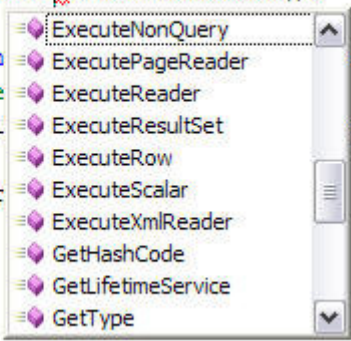
10. Customized Data Provider

By providing the factory classes now ADO.NET extends its support to custom data provider. Now you don't have to write a data provider dependent code. You use the base classes of data provider and let the connection string does the trick for you.

11. DataReader's New Execute Methods

Now command object supports more execute methods. Besides old **ExecuteNonQuery**, **ExecuteReader**, **ExecuteScalar**, and **ExecuteXmlReader**, the new execute methods are **ExecutePageReader**, **ExecuteResultSet**, and **ExecuteRow**.

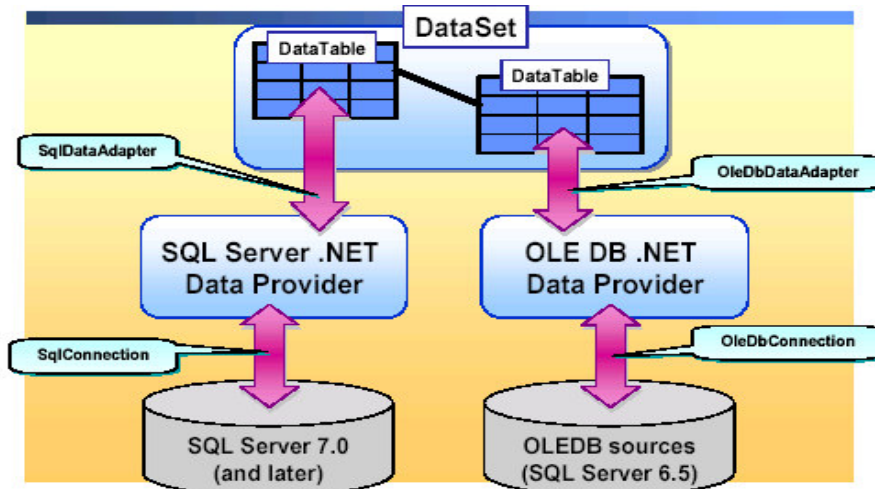
```
// Execute reader
SqlDataReader reader = cmd.ExecuteReader();
// Create SqlBulkCopy
SqlBulkCopy bulkData = new SqlBulkCopy(cmd.Connection);
// Set destination table
bulkData.DestinationTableName = "table";
// Write data
bulkData.WriteToServer(reader);
// Close objects
bulkData.Close();
destination.Close();
source.Close();
```



12. Multiple Active ResultSets

Using this feature we can have more than one simultaneous pending request per connection i.e. multiple active DataReader is possible. Previously when a DataReader is open and if we use that connection in another DataReader, we used to get the following error "System.InvalidOperationException: There is already an open DataReader associated with this connection which must be closed first". This error won't come now, as this is possible now because of MAR's. This feature is supported only in Yukon.

The ADO.NET Object Model

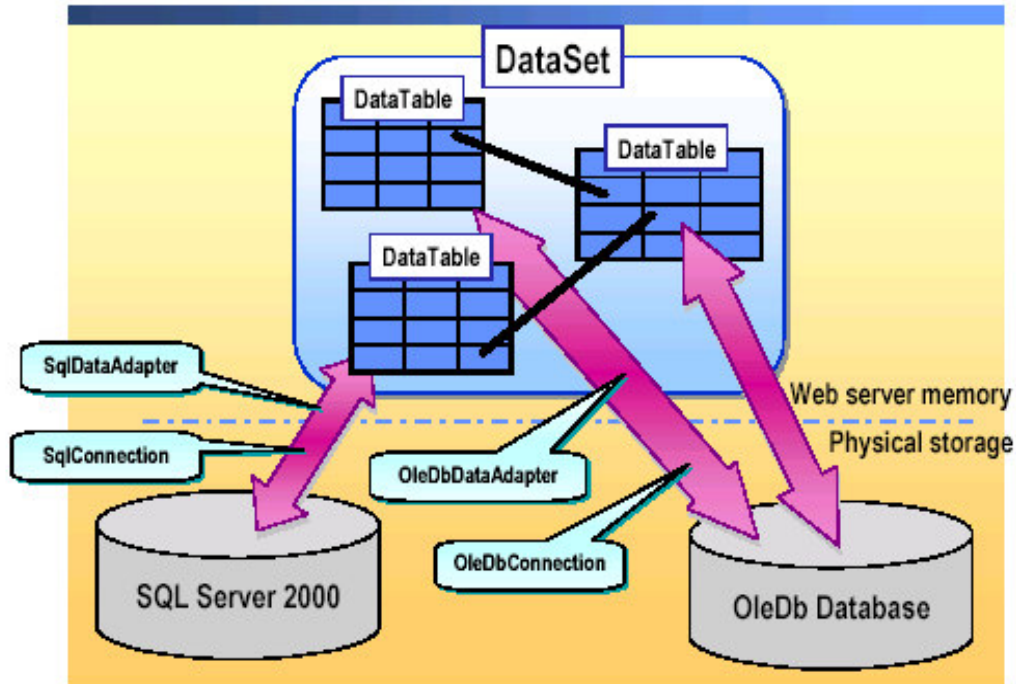


The ADO.NET object model provides the structure for accessing data from different data sources. There are two main components of the ADO.NET object model: the **DataSet** and the .NET data provider.

The .NET data provider provides the link between the data source and the **DataSet**. Examples of objects that are provided by the .NET data providers are listed in the following table.

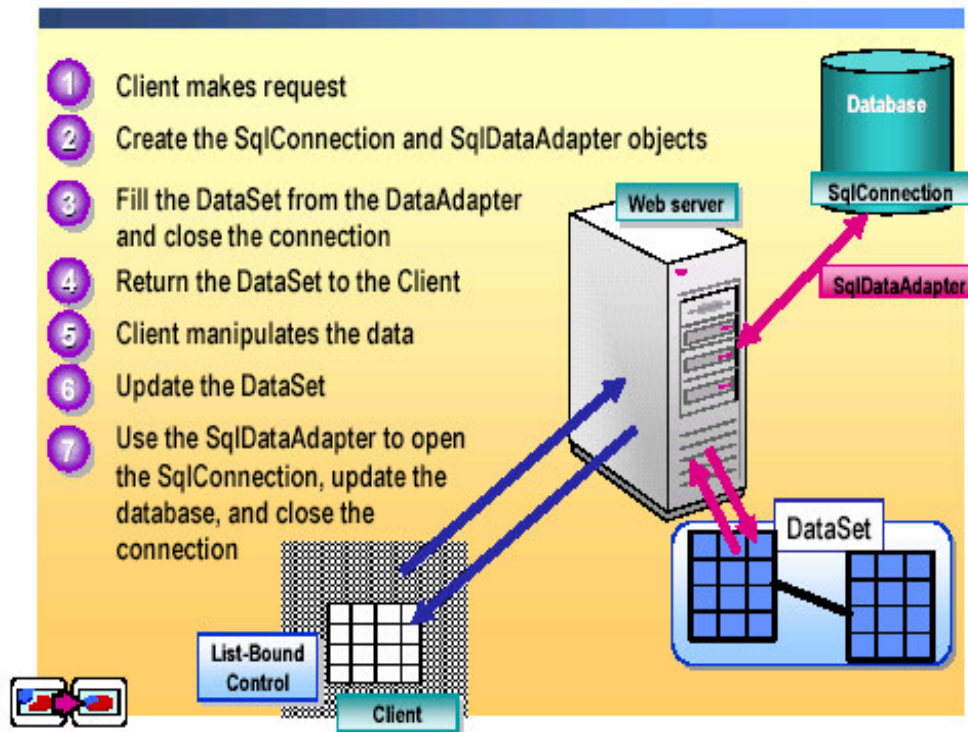
.NET data provider objects	Purpose
Connection	Provides connectivity to the data source.
Command	Provides access to database commands.
DataReader	Provides data streaming from the source.
DataAdapter	Uses the Connection object to provide a link between the DataSet and the data provider. The DataAdapter object also reconciles changes that are made to the data in the DataSet .

What is a DataSet?

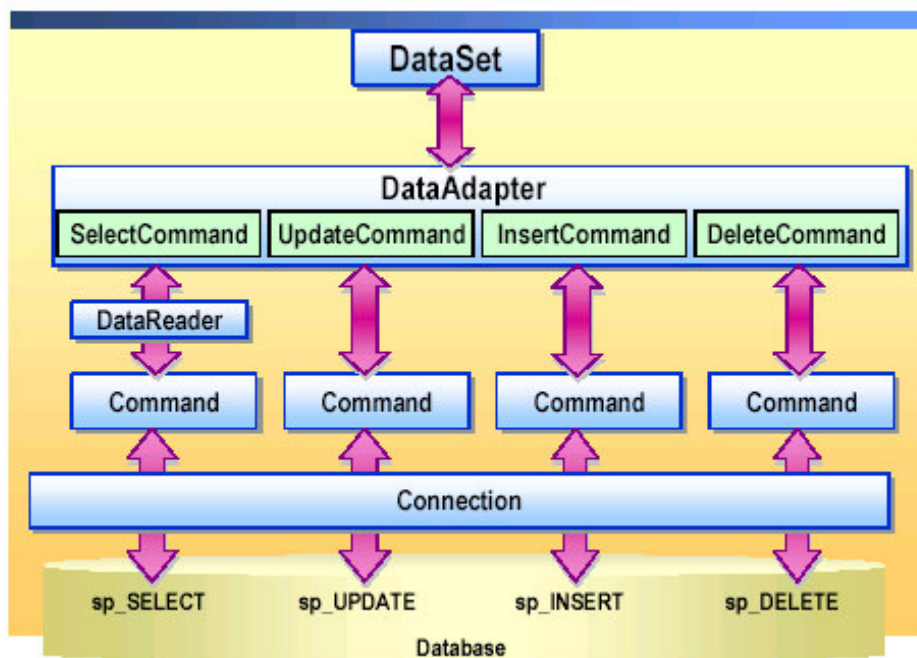


A **DataSet** stores information in a disconnected environment. After you establish a connection to a database, you can then access its data. The **DataSet** object is the primary way to store that data when using ADO.NET. The **DataSet** object allows you to store data, which has been collected from a data source, in your Web application. The data stored in a **DataSet** can be manipulated without the ASP.NET Web Form maintaining a connection to the data source. A connection is re-established only when the data source is updated with changes. The **DataSet** object stores the data in one or more **DataTables**. Each **DataTable** may be populated with data from a unique data source. You can also establish relationships between two **DataTables** by using a **DataRelation** object.

Accessing Data with ADO.NET



The DataAdapter Object Model



When a **DataAdapter** connects to a database, it uses several other objects to communicate with the database. The **DataAdapter** object uses the **Connection** object to connect to a database, and it then uses **Command** objects to retrieve data and resolve changes to the database.

The **DataAdapter** object has properties. These properties are **SqlCommand** or **OleDbCommand** objects that contain SQL statements. The **DataAdapter** object has the following four **Command-type** properties:

- SelectCommand:** This property issues a **SQL SELECT** statement.
- UpdateCommand:** This property issues a **SQL UPDATE** statement.
- InsertCommand:** This property issues a **SQL INSERT** statement.
- DeleteCommand:** This property issues a **SQL DELETE** statement.

Naresh I Technologies