ADO.NET Architecture

Data processing has traditionally relied primarily on a connection-based, two-tier model. As data processing increasingly uses multi-tier architectures, programmers are switching to a disconnected approach to provide better scalability for their applications.

ADO.NET is a set of classes that comes with the Microsoft .NET framework to facilitate data access from managed languages. ADO.NET has been in existence for a long time and it provides a comprehensive and complete set of libraries for data access. The strength of ADO.NET is firstly that it lets applications access various types of data using the same methodology. If you know how to use ADO.NET to access a SQL Server database then the same methodology can be used to access any other type of database (like Oracle or MS Access) by just using a different set of classes. Secondly, ADO.NET provides two models for data access: a connected model where you can keep the connection with the database and perform data access, and another way is to get all the data in ADO.NET objects that let us perform data access on disconnected objects.

Note: Many developers and development houses are now using ORMs to perform data access instead of using ADO.NET. ORMs provide a lot of data access functionality out of the box and relieves users from writing mundane data access code again and again. Still, I think that knowing and understanding ADO.NET is crucial as a .NET developer as it gives a better understanding of the data access methodologies. Also, there are many development houses that are still using ADO.NET.

ADO.NET Components

The two main components of ADO.NET 3.0 for accessing and manipulating data are the .NET Framework data providers and the DataSet.

.NET Framework Data Providers

The .NET Framework Data Providers are components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data. The Connection object provides connectivity to a data source. The Command object enables access to database commands to return data, modify data, run stored procedures, and send or retrieve parameter information. The DataReader provides a high-performance stream of data from the data source. Finally, the DataAdapter provides the bridge between the DataSet object and the data source. The DataAdapter uses Command objects to execute SQL commands at the data source to both load the DataSet with data and reconcile changes that were made to the data in the DataSet back to the data source. For more information, see .NET Framework Data Providers and Retrieving and Modifying Data in ADO.NET.

The DataSet

The ADO.NET DataSet is explicitly designed for data access independent of any data source. As a result, it can be used with multiple and differing data sources, used with XML data, or used to manage data local to the application. The DataSet contains a collection of one or more DataTable objects consisting of rows and columns of data, and also primary key, foreign key, constraint, and relation information about the data in the DataTable objects. For more information, see DataSets, DataTables, and DataViews.

Choosing a DataReader or a DataSet

When you decide whether your application should use a DataReader (see Retrieving Data Using a DataReader) or a DataSet (see DataSets, DataTables, and DataViews), consider the type of functionality that your application requires. Use a DataSet to do the following:

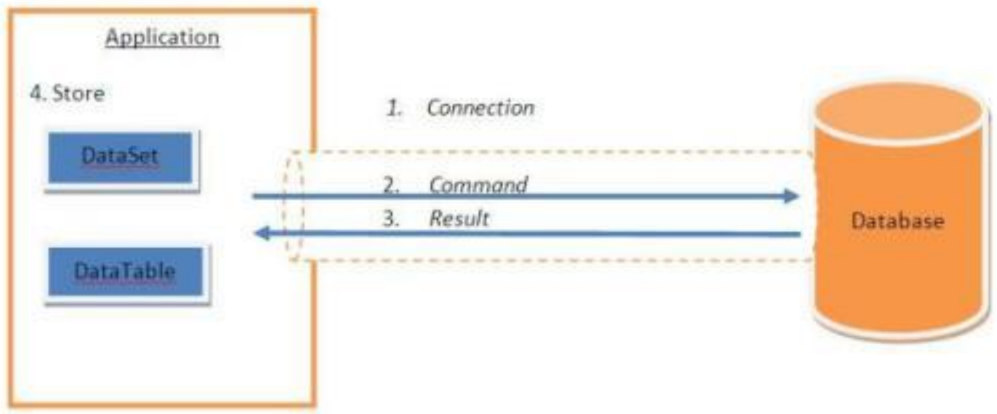
- Cache data locally in your application so that you can manipulate it. If you only need to read the results of a query, the DataReader is the better choice.
- Remote data between tiers or from an XML Web service.
- Interact with data dynamically such as binding to a Windows Forms control or combining and relating data from multiple sources.
- Perform extensive processing on data without requiring an open connection to the data source, which frees the connection to be used by other clients.

If you do not require the functionality provided by the DataSet, you can improve the performance of your application by using the DataReader to return your data in a forward-only, read-only manner. Although the DataAdapter uses the DataReader to fill the contents of a DataSet (see Populating a DataSet from a DataAdapter), by using the DataReader, you can boost performance because you will save memory that would be consumed by the DataSet, and avoid the processing that is required to create and fill the contents of the DataSet.

In a typical scenario requiring data access, we need to perform four major tasks:

1. Connecting to the database
2. Passing the request to the database, i.e., a command like select, insert, or update.
3. Getting back the results, i.e., rows and/or the number of rows effected.
4. Storing the result and displaying it to the user.

This can be visualized as:



ADO.NET A set of classes in the .NET Framework that enables you to connect to a database, retrieve data, execute stored procedures, add, update, or delete records in a table.

connection object An object in ADO.NET that allows you to open and execute commands against a database.

The following code creates an instance of the SqlConnection object, sets the connectionString property, and opens a connection to the database:

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();
```

A **Command** object is used to call a stored procedure or execute a dynamic SQL statements (insert, update, delete, and select) .

The **Command's ExecuteNonQuery** method is used to execute nonresult-returning queries such as an INSERT or UPDATE command.

The following code demonstrates how to execute an insert statement against the database:

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();

            SqlCommand cmd = new SqlCommand();
            cmd.Connection = cn;
            cmd.CommandType = CommandType.Text;
            cmd.CommandText = "INSERT INTO Person (FirstName, LastName) VALUES ('Joe',
'Smith')";
```

```
            cmd.ExecuteNonQuery();
            cn.Close();
```

The following code executes the stored procedure and passes in the @FirstName and @LastName parameters:

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();

            SqlCommand cmd = new SqlCommand();
            cmd.Connection = cn;
            cmd.CommandType = CommandType.StoredProcedure;
            cmd.CommandText = "PersonInsert";
            cmd.Parameters.Add(new SqlParameter("@FirstName", "Joe"));
            cmd.Parameters.Add(new SqlParameter("@LastName", "Smith"));
            cmd.ExecuteNonQuery();

Assume you have the following stored procedure that inserts a recored into the Person
table.

CREATE PROCEDURE PersonInsert
       @FirstName varchar(50)
       @LastName varchar(50)
AS
BEGIN
       INSERT INTO PERSON (FirstName, LastName) VALUES (@FirstName, @LastName)
END
```

ExecuteReader method Executes the CommandText against the Connection and returns a DbDataReader. A **DBDataReader** object is a read-only, forward-only cursor connected to the database.

The following code prints all the records in the Person table to the output window using a DBDataReader object.

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();

            SqlCommand cmd = new SqlCommand();
            cmd.Connection = cn;
            cmd.CommandType = CommandType.Text;
            cmd.CommandText = "SELECT * FROM Person";
            SqlDataReader dr = cmd.ExecuteReader();

            if (dr.HasRows)
            {
                while (dr.Read())
                {
                    Debug.WriteLine(string.Format("First Name: {0} , Last Name: {1}",
dr["FirstName"], dr["LastName"]));
                }
            }
            dr.Close();
            cn.Close();
            SqlCommand cmd = new SqlCommand();
            cmd.Connection = cn;
            cmd.CommandType = CommandType.Text;
            cmd.CommandText = "SELECT * FROM Person";
            SqlDataReader dr = cmd.ExecuteReader();

            if (dr.HasRows)
            {
                while (dr.Read())
                {
                    Debug.WriteLine(string.Format("First Name: {0} , Last Name: {1}",
dr["FirstName"], dr["LastName"]));
                }
            }
            dr.Close();

            cn.Close();
```

The **Command's ExecuteScalar** method is used to return a single value from a database such as when a query returns a SUM or COUNT.

The following code calls the ExecuteScalar method and returns the Count of records in the Person table:

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();

            SqlCommand cmd = new SqlCommand();
            cmd.Connection = cn;
            cmd.CommandType = CommandType.Text;
            cmd.CommandText = "SELECT COUNT(*) FROM Person";
            object obj = cmd.ExecuteScalar();

            Debug.WriteLine(string.Format("Count: {0}", obj.ToString()));

            cn.Close();
```

A **DataSet** is a disconnected resultset and can contain one or more DataTables. A DataAdapter is used to fill a DataSet.

The following code uses a DataSet to retrieve the data from the Person table and write all the records to the Output window:

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Person", cn);
```

```
            DataSet ds = new DataSet();
            da.Fill(ds, "Person");

            cn.Close();

            foreach (DataRow row in ds.Tables[0].Rows)
            {
                Debug.WriteLine(string.Format("First Name: {0} , Last Name: {1}", 
row["FirstName"], row["LastName"]));
            }
```

A **DataAdapte**r can be used with a DataSet to add, update, or delete records in a database.

The following code demonstrates how to use a DataAdapter to add a new record to a table.

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User 
Id=myUsername;Password=myPassword;";
cn.Open();
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Person", cn);

            //Create the insert command
            SqlCommand insert = new SqlCommand();
            insert.Connection = cn;
            insert.CommandType = CommandType.Text;
            insert.CommandText = "INSERT INTO Person (FirstName, LastName) VALUES 
(@FirstName, @LastName)";

            //Create the parameters
            insert.Parameters.Add(new SqlParameter("@FirstName", SqlDbType.VarChar, 50, 
"FirstName"));
            insert.Parameters.Add(new SqlParameter("@LastName", SqlDbType.VarChar, 50, 
"LastName"));

            //Associate the insert command with the DataAdapter.
            da.InsertCommand = insert;

            //Get the data.
            DataSet ds = new DataSet();
            da.Fill(ds, "Person");

            //Add a new row.
            DataRow newRow = ds.Tables[0].NewRow();
            newRow["FirstName"] = "Jane";
            newRow["LastName"] = "Doe";
            ds.Tables[0].Rows.Add(newRow);

            //Updat the database.
            da.Update(ds.Tables[0]);

            cn.Close();
```

The following code demonstrates how to use the UpdateCommand property of a DbataAdapter object and update a record, and also how to use DeleteCommand property of a DbDataAdapter to delete a record.

```
SqlConnection cn = new SqlConnection();
            cn.ConnectionString = "Server=myServerAddress;Database=myDataBase;User
Id=myUsername;Password=myPassword;";
cn.Open();
SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Person", cn);

            //Create the update command
            SqlCommand update = new SqlCommand();
            update.Connection = cn;
            update.CommandType = CommandType.Text;
            update.CommandText = "UPDATE Person SET FirstName = @FirstName, LastName =
@LastName WHERE PersonId = @PersonId";

            //Create the parameters
            update.Parameters.Add(new SqlParameter("@FirstName", SqlDbType.VarChar, 50,
"FirstName"));
            update.Parameters.Add(new SqlParameter("@LastName", SqlDbType.VarChar, 50,
"LastName"));
            update.Parameters.Add(new SqlParameter("@PersonId", SqlDbType.Int, 0,
"PersonId"));

            //Create the delete command
            SqlCommand delete = new SqlCommand();
            delete.Connection = cn;
            delete.CommandType = CommandType.Text;
            delete.CommandText = "DELETE FROM Person WHERE PersonId = @PersonId";

            //Create the parameters
            SqlParameter deleteParameter = new SqlParameter("@PersonId", SqlDbType.Int,
0, "PersonId");
            deleteParameter.SourceVersion = DataRowVersion.Original;
            delete.Parameters.Add(deleteParameter);

            //Associate the update and delete commands with the DataAdapter.
            da.UpdateCommand = update;
            da.DeleteCommand = delete;

            //Get the data.
            DataSet ds = new DataSet();
            da.Fill(ds, "Person");

            //Update the first row
            ds.Tables[0].Rows[0]["FirstName"] = "Jack";
            ds.Tables[0].Rows[0]["LastName"] = "Johnson";

            //Delete the second row.
            ds.Tables[0].Rows[1].Delete();

            //Updat the database.
            da.Update(ds.Tables[0]);


            cn.Close();
```