Microsoft ActiveX Data Objects.

ADO.NET is a set of computer software components that programmers can use to access data and data services from the database.

It is a part of the base class library that is included with the Microsoft .NET Framework.

ADO.NET provides a bridge between the front end controls and the back end database

ADO.NET is an object-oriented set of libraries that allows you to interact with data sources.

Commonly, the data source is a database, but it could also be a text file, an Excel spreadsheet, or an XML

The following are, a few of the different types of .NET applications that use ADO.NET to connect to a database, execute commands, and retrieve data.

ASP.NET Web Applications

Windows Applications

Console Applications

Different .NET Data Providers

Data Provider for SQL Server - System.Data.SqlClient

Data Provider for Oracle - System.Data.OracleClient

Data Provider for OLEDB - System.Data.OleDb

Data Provider for ODBC - System.Data.Odbc

The SqlCommand Object

The process of interacting with a database means that you must specify the actions you want to occur.

This is done with a command object. You use a command object to send SQL statements to the database.

A command object uses a connection object to figure out which database to communicate with.

You can use a command object alone, to execute a command directly, or assign a reference to a command object to an SqlDataAdapter,

which holds a set of commands that work on a group of data as described below.

It allow you to talk to a data source and send commands to it.It also allows you to specify what type of interaction you want to perform with a database.

For example, you can do select, insert, modify, and delete commands on rows of data in a database table.

A SqlCommand object allows you to query and send commands to a database.

You should use parameters to filter queries in a secure manner.

The process of using parameter contains three steps: define the parameter in the SqlCommand command string,

declare the SqlParameter object with applicable properties,

and assign the SqlParameter object to the SqlCommand object.

When the SqlCommand executes, parameters will be replaced with values specified by the SqlParameter object.

The SqlDataReader Object

Many data operations require that you only get a stream of data for reading.

The data reader object allows you to obtain the results of a SELECT statement from a command object.

For performance reasons, the data returned from a data reader is a fast forward-only stream of data.

This means that you can only pull the data from the stream in a sequential manner T

This is good for speed, but if you need to manipulate data, then a DataSet is a better object to work with.

The ExecuteReader method returns a SqlDataReader object for viewing the results of a select query.

SqlDataReader is connection oriented and the connection needs to be opened explicitly, by calling the Open() method on the connection object,

before calling the ExecuteReader() method of the command object.

The DataSet Object

DataSet objects are in-memory representations of data.

They contain multiple Datatable objects, which contain columns and rows, just like normal database tables.

You can even define relations between tables to create parent-child relationships.

The DataSet is specifically designed to help manage data in memory and to support disconnected operations on data, when such a scenario make sense.

The DataSet is an object that is used by all of the Data Providers, which is why it does not have a Data Provider specific prefix.

DataSet can fetch multiple TableRows at a time

DataTable-

A DataTable is an in-memory representation of a single database table which has collection of rows and columns.

DataTable fetches only one TableRow at a time

It is used as a temporary storage.

SqlDataAdapter Object

the data adapter contains command object references for SELECT, INSERT, UPDATE, and DELETE operations on the data.

You will have a data adapter defined for each table in a DataSet and it will take care of all communication with the database for you.

All you need to do is tell the data adapter when to load from or write to the database.

SqlDataAdapter and DataSet provides us with disconnected data access model.

Ex:-

SqlConnection conn = new SqlConnection();

conn.ConnectionString = "Data source=SUNNY;Initial catalog=demo;Integrated security=true";

string query = "select \* from student";

SqlCommand cmd = new SqlCommand(query, conn);

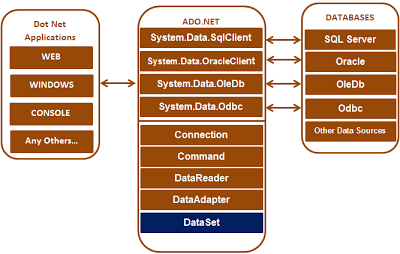
SqlDataAdapter sda = new SqlDataAdapter(cmd);

DataSet ds = new DataSet();

sda.Fill(ds);

GridView1.DataSource = ds;

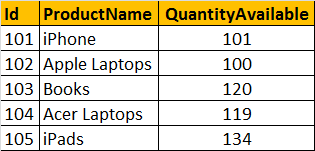
GridView1.DataBind();

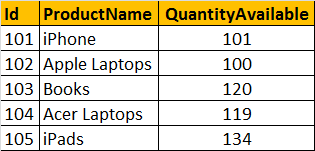


**The DataSet object is not provider specific**. Once we connect to a Database, execute command, and retrieve data into .NET application. The data can then be stored in a DataSet and work independently of the database.

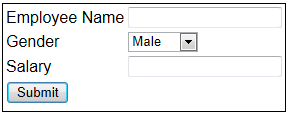
**We can also use "using" statement to properly close the connection as shown below.** We don't have to explicitly call **Close()** method, when **using**is used. The connection will be automatically closed for us.  
protected void Page\_Load(object sender, EventArgs e)  
{  
    using (SqlConnection connection = new SqlConnection("data source=.; database=Sample\_Test\_DB; integrated security=SSPI"))  
    {  
        SqlCommand cmd = new SqlCommand("Select \* from tblProductInventory", connection);  
        connection.Open();  
        GridView1.DataSource = cmd.ExecuteReader();  
        GridView1.DataBind();  
    }  
}

**What are the 2 uses of an using statement in C#?**  
**1.** To import a namespace. Example: using System;  
**2.** To close connections properly as shown in the example above

**The best practice is to store the connection in the configuration file**, from which all the pages can read and use it. This way we have only one place to change, and we don't have to re-build and re-deploy our application. This saves a lot of time.  
  
**In an asp.net web application, the configuration strings can be stored in web.config file**.  
**It is also possible, to create an instance of SqlCommand class using the parameter less constructor**, and then later specify the command text and connection, using the **CommandText** and **Connection** properties of the **SqlCommand** object as shown below.  
protected void Page\_Load(object sender, EventArgs e)  
{  
    string ConnectionString = ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;  
    using (SqlConnection connection = new SqlConnection("data source=.; database=Sample\_Test\_DB; integrated security=SSPI"))  
    {  
        //Create an instance of SqlCommand class using the parameter less constructor  
        SqlCommand cmd = new SqlCommand();  
        //Specify the command, we want to execute using the CommandText property  
        cmd.CommandText = "Select Id,ProductName,QuantityAvailable from tblProductInventory";  
        //Specify the connection, on which we want to execute the command   
        //using the Connection property  
        cmd.Connection = connection;  
        connection.Open();  
        //As the T-SQL statement that we want to execute return multiple rows of data,   
        //use ExecuteReader() method of the command object.  
        GridView1.DataSource = cmd.ExecuteReader();  
        GridView1.DataBind();  
    }  
}   
  
SQL Injection-  
  
   
  
**<asp:TextBox ID="ProductNameTextBox" runat="server"></asp:TextBox>**  
**<asp:Button ID="GetProductsButton" runat="server" Text="Get Products" />**  
**<br /><br />**  
**<asp:GridView ID="ProductsGridView" runat="server">**  
**</asp:GridView>**  
  
**Now double click the Button control to generate the Click event handler**in the code behind file, and then copy and paste the following code. In this example, we are building the query dynamically by concatenating the strings that the user has typed into the textbox. This is extremely dangerous, as it is vulnerable to SQL injection attacks.  
protected void GetProductsButton\_Click(object sender, EventArgs e)  
{  
    string ConnectionString = ConfigurationManager.ConnectionStrings["DatabaseConnectionString"].ConnectionString;  
    using (SqlConnection connection = new SqlConnection("DatabaseConnectionString"))  
    {  
        //Build the query dynamically, by concatenating the text, that the user has   
        //typed into the ProductNameTextBox. This is a bad way of constructing  
        //queries. This line of code will open doors for sql injection attack  
        SqlCommand cmd = new SqlCommand("Select \* from tblProductInventory where ProductName like '" + ProductNameTextBox.Text + "%'", connection);  
        connection.Open();  
        ProductsGridView.DataSource = cmd.ExecuteReader();  
        ProductsGridView.DataBind();  
    }  
}   
  
**Now, run the project. Enter letter "i" into the textbox and click Get Products button**. The **iPhone** and **ipad** products will be listed in the **gridview** as expected. But remember, user can type some dangerous sql queries into the textbox, which in turn will be executed by the application on the database. To give you a flavour of that, just imagine what could happen if the user types the following into the TextBox, and clicks Get Products button.  
**i'; Delete from tblProductInventory --**  
  
**Now execute the following select query on the database**  
Select \* from tblProductInventory  
  
**The entire data from tblProductInventory table is deleted**. This is called **SQL injection** attack. I have seen a lot of new developers building queries dynamically by concatenating the strings, that end users enter into user interface controls like textboxes. Just imagine the extent of damage that can happen as a result of sql injection.

1. Preventing sql injection using parametrized queries and stored procedures.  
2. How to execute stored procedures and parameterized queries using ADO.NET command object  
  
**The table, this demo is based on, is shown below.**   
   
  
**The following ADO.NET code is from Part 5**. This is the code, that let's sql injection happen.   
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
using (SqlConnection con = new SqlConnection(CS))  
{  
    string Command = "Select \* from tblProductInventory where ProductName like '" + TextBox1.Text + "%'";  
    SqlCommand cmd = new SqlCommand(Command, con);  
    con.Open();  
    GridView1.DataSource = cmd.ExecuteReader();  
    GridView1.DataBind();  
}  
  
**The above code can be easily re-written using parameterized queries** to prevent sql injection attack. The re-written code is shown below. Notice, that the query now uses parameter - **@ProductName**. The value for this parameter is then provided using the **AddWithValue**() method. The parameter is associated with the command object using **Prameters** collection property of the **command** object.  
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
using (SqlConnection con = new SqlConnection(CS))  
{  
    // Parameterized query. @ProductName is the parameter  
    string Command = "Select \* from tblProductInventory where ProductName like @ProductName" ;  
    SqlCommand cmd = new SqlCommand(Command, con);  
    // Provide the value for the parameter  
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");  
    con.Open();  
    GridView1.DataSource = cmd.ExecuteReader();  
    GridView1.DataBind();  
}   
  
**Sql injection can also be prevented using stored procedures.** So, first let's write a stored procedure, that returns the list of products. This stored procedure takes an input parameter **@ProductName.**  
Create Procedure spGetProductsByName  
@ProductName nvarchar(50)  
as  
Begin  
 Select \* from tblProductInventory   
 where ProductName like @ProductName + '%'  
End  
  
**To test this procedure execute the follwing command in sql server management studio.**  
Execute spGetProductsByName 'ip'  
  
**Now, let's re-write the code, to use stored procedure spGetProductsByName.**  
string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
using (SqlConnection con = new SqlConnection(CS))  
{  
    // The command, that we want to execute is a stored procedure,  
    // so specify the name of the procedure as cmdText  
    SqlCommand cmd = new SqlCommand("spGetProductsByName", con);  
    // Specify that the T-SQL command is a stored procedure  
    cmd.CommandType = System.Data.CommandType.StoredProcedure;  
    // Associate the parameter and it's value with the command object  
    cmd.Parameters.AddWithValue("@ProductName", TextBox1.Text + "%");  
    con.Open();  
    GridView1.DataSource = cmd.ExecuteReader();  
    GridView1.DataBind();  
}  
  
**If you type the following input into the TextBox**, the entire content of the TextBox is now treated as a value for the parameter - @ProductName not as a seperate sql statement.  
**i'; Delete from tblProductInventory --**  
  
**So the conclusion is that, always used parameterized queries or stored procedures, to avoid sql injection attacks.**

Pasrt7-

In [Part 6](http://csharp-video-tutorials.blogspot.com/2012/10/sql-injection-prevention-part-6.html), we have discussed that sql injection can be prevented by using parameterized queries or stored procedures. We have also seen how to call a stored procedure with input parameters. In this part, we will learn about calling a stored procedure with output parameters.  
  
Our example, will be based on **tblEmployees**. The script to create this table is shown below. The important point to note here is that, **EmployeeId**is marked as an identity column. When inserting a row into the table, we need not provide a value for the **EmployeeId** column. If you want to learn more about identity column, please watch - [Part 7 of SQL Server : Identity Column in SQL Server.](http://csharp-video-tutorials.blogspot.com/2012/08/identity-column-in-sql-server-part-7.html)  
Create Table tblEmployees  
(  
 EmployeeId int identity primary key,  
 Name nvarchar(50),  
 Gender nvarchar(10),  
 Salary int  
)   
  
  
  
**Script to insert sample data. Notice, that in the insert statement we are not providing a value for EmployeeId Column.**  
Insert into tblEmployees values('Mike','Male',5000)  
Insert into tblEmployees values('Pam','Female',3500)  
Insert into tblEmployees values('John','Male',2350)  
Insert into tblEmployees values('Sara','Female',5700)  
Insert into tblEmployees values('Steve','Male',4890)  
Insert into tblEmployees values('Sana','Female',4500)  
  
**1.** **spAddEmployee** stored procedure inserts a row into tblEmployees tables.   
**2.** @Name, @Gender and @Salary are **input** parameters.  
**3.** @EmployeeId is an **output** parameter  
**4.** The stored procedure has got only 2 lines of code with in the body. The first line inserts a row into the **tblEmployees**table. The second line, gets the **auto generated identity value** of the  **EmployeeId**column.  
**5.** This procedure, will later be called by a dot net application.  
Create Procedure spAddEmployee    
@Name nvarchar(50),    
@Gender nvarchar(20),    
@Salary int,    
@EmployeeId int Out    
as    
Begin    
 Insert into tblEmployees values(@Name, @Gender, @Salary)    
 Select @EmployeeId = SCOPE\_IDENTITY()    
End   
  
  
  
**At this point, we have done everything that is required for our demo, from a database perspective. Now let's flip to visual studio. Create an asp.net web application. Copy and Paste the following HTML onto a webform.**  
<table style="border: 1px solid black; font-family:Arial">  
    <tr>  
        <td>  
            Employee Name  
        </td>  
        <td>  
            <asp:TextBox ID="txtEmployeeName" runat="server"></asp:TextBox>  
        </td>  
    </tr>          
    <tr>  
        <td>  
            Gender  
        </td>  
        <td>  
            <asp:DropDownList ID="ddlGender" runat="server">  
                <asp:ListItem>Male</asp:ListItem>  
                <asp:ListItem>Female</asp:ListItem>  
            </asp:DropDownList>  
        </td>  
    </tr>   
    <tr>  
        <td>  
            Salary  
        </td>  
        <td>  
            <asp:TextBox ID="txtSalary" runat="server"></asp:TextBox>  
        </td>  
    </tr>         
    <tr>  
        <td colspan="2">  
            <asp:Button ID="btnSubmit" runat="server" Text="Submit"   
                onclick="btnSubmit\_Click" />  
        </td>  
    </tr>            
    <tr>  
        <td colspan="2">  
            <asp:Label ID="lblMessage" runat="server"></asp:Label>  
        </td>  
    </tr>    
</table>  
  
**The design of the webform, should be as shown below.**  
   
  
**Copy and paste the following code in the code behind page.**  
protected void btnSubmit\_Click(object sender, EventArgs e)  
{  
    //Read the connection string from Web.Config file  
    string ConnectionString = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
    using (SqlConnection con = new SqlConnection(ConnectionString))  
    {  
        //Create the SqlCommand object  
        SqlCommand cmd = new SqlCommand("spAddEmployee", con);  
        //Specify that the SqlCommand is a stored procedure  
        cmd.CommandType = System.Data.CommandType.StoredProcedure;  
  
        //Add the input parameters to the command object  
        cmd.Parameters.AddWithValue("@Name", txtEmployeeName.Text);  
        cmd.Parameters.AddWithValue("@Gender", ddlGender.SelectedValue);  
        cmd.Parameters.AddWithValue("@Salary", txtSalary.Text);  
  
        //Add the output parameter to the command object  
        SqlParameter outPutParameter = new SqlParameter();  
        outPutParameter.ParameterName = "@EmployeeId";  
        outPutParameter.SqlDbType = System.Data.SqlDbType.Int;  
        outPutParameter.Direction = System.Data.ParameterDirection.Output;  
        cmd.Parameters.Add(outPutParameter);  
  
        //Open the connection and execute the query  
        con.Open();  
        cmd.ExecuteNonQuery();  
                  
        //Retrieve the value of the output parameter  
        string EmployeeId = outPutParameter.Value.ToString();  
        lblMessage.Text = "Employee Id = " + EmployeeId;  
    }  
}  
**If for some reason, you want to loop thru each row in the SqlDataReader object**, then use the **Read**() method, which returns **true** as long as there are rows to read. If there are no more rows to read, then this method will return false. In the following example, we loop thru each row in the **SqlDataReader** and then compute the **10% discounted price.**  
string ConnectionString = ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;  
using (SqlConnection connection = new SqlConnection(ConnectionString))  
{  
    connection.Open();  
    SqlCommand command = new SqlCommand("Select \* from tblProductInventory", connection);  
    using (SqlDataReader reader = command.ExecuteReader())  
    {  
        // Create the DataTable and columns. This will   
        // be used as the datasource for the GridView  
        DataTable sourceTable = new DataTable();  
        sourceTable.Columns.Add("ID");  
        sourceTable.Columns.Add("Name");  
        sourceTable.Columns.Add("Price");  
        sourceTable.Columns.Add("DiscountedPrice");  
  
        while (reader.Read())  
        {  
            //Calculate the 10% discounted price  
            int OriginalPrice = Convert.ToInt32(reader["UnitPrice"]);  
            double DiscountedPrice = OriginalPrice \* 0.9;  
  
            // Populate datatable column values from the SqlDataReader  
            DataRow datarow = sourceTable.NewRow();  
            datarow["ID"] = reader["ProductId"];  
            datarow["Name"] = reader["ProductName"];  
            datarow["Price"] = OriginalPrice;  
            datarow["DiscountedPrice"] = DiscountedPrice;  
  
            //Add the DataRow to the DataTable  
            sourceTable.Rows.Add(datarow);  
        }  
  
        // Set sourceTable as the DataSource for the GridView  
        ProductsGridView.DataSource = sourceTable;  
        ProductsGridView.DataBind();  
    }  
}  
  
**SQL script to create the table we used in the Demo.**  
Create table tblProductInventory   
(  
 ProductId int identity primary key,  
 ProductName nvarchar(50),  
 UnitPrice int  
)  
  
**Script to populate data**  
Insert into tblProductInventory values('iPhone',350)  
Insert into tblProductInventory values('Apple Laptops',1250)  
Insert into tblProductInventory values('Books',110)  
Insert into tblProductInventory values('Acer Laptops',1150)  
Insert into tblProductInventory values('iPads',450)

Part9-

**In this video we will learn about retrieving two or more result sets using the SqlDataReader** object's **NextResult**() method. The follwoing SqlCommand object returns two result-sets, **one from - tblProductInventory**and the **other from tblProductCategories**.  
SqlCommand command = new SqlCommand("select \* from tblProductInventory; select \* from tblProductCategories", connection);  
  
**When you run this code only the result set from tblProductInventory is displayed.** The result-set from **tblProductCategories** is not shown.  
string ConnectionString = ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;  
using (SqlConnection connection = new SqlConnection(ConnectionString))  
{  
    connection.Open();  
    SqlCommand command = new SqlCommand("select \* from tblProductInventory; select \* from tblProductCategories", connection);  
    using (SqlDataReader reader = command.ExecuteReader())  
    {  
        ProductsGridView.DataSource = reader;  
        ProductsGridView.DataBind();  
  
        CategoriesGridView.DataSource = reader;  
        CategoriesGridView.DataBind();  
    }  
}   
**To retrieve the second result-set from SqlDataReader object**, use the **NextResult**() as shown in the code snippet below. The **NextResult**() method returns true and advances to the next result-set.   
string ConnectionString = ConfigurationManager.ConnectionStrings["DBConnectionString"].ConnectionString;  
using (SqlConnection connection = new SqlConnection(ConnectionString))  
{  
    connection.Open();  
    SqlCommand command = new SqlCommand("select \* from tblProductInventory; select \* from tblProductCategories", connection);  
    using (SqlDataReader reader = command.ExecuteReader())  
    {  
        ProductsGridView.DataSource = reader;  
        ProductsGridView.DataBind();  
  
        while (reader.NextResult())  
        {  
            CategoriesGridView.DataSource = reader;  
            CategoriesGridView.DataBind();  
        }  
    }  
}  
  
**The SqlDataReader object's Read() method is used to loop thru the rows in a given result set, where as the NextResult() method is used to loop thru multiple  result sets.**  
**In this video we will learn about**  
**1.** Caching a DataSet  
**2.** Check to see if the DataSet exists in the cache, and load data from the cache.  
**3.** Clearing the cache  
  
**We will be using tblProductInventory** table for this demo. Sql script to the create the table populate it with some sample data.  
Create table tblProductInventory   
(  
 ProductId int identity primary key,  
 ProductName nvarchar(50),  
 UnitPrice int  
)  
  
Insert into tblProductInventory values('iPhone',350)  
Insert into tblProductInventory values('Apple Laptops',1250)  
Insert into tblProductInventory values('Books',110)  
Insert into tblProductInventory values('Acer Laptops',1150)  
Insert into tblProductInventory values('iPads',450)   
  
  
  
**Create an asp.net web application project**, and add the following database connectionstring to the web.config file.  
<connectionStrings>  
  <add name="DBCS"  
  connectionString="data source=.; database=Sample\_Test\_DB; Integrated Security=SSPI"  
  providerName="System.Data.SqlClient" />  
</connectionStrings>  
  
**Drag and drop 2 button controls, a label and a gridview control onto the webform**.  
**1.** Set the ID of the first button control to btnLoadData and Text to Load Data  
**2.** Set the ID of the second button control to btnClearnCache and Text to Clear Cache  
**3.** Set the ID of the label control to lblMessage and remove the Text property  
**4.** Set the ID of the GridView to gvProducts  
  
**At this stage the HTML of your webform, should be as shown below.**  
<asp:Button ID="btnLoadData" runat="server" Text="Load Data"   
    onclick="btnLoadData\_Click" />  
<asp:Button ID="btnClearnCache" runat="server" Text="Clear Cache"   
    onclick="btnClearnCache\_Click" />  
<br />  
<br />  
<asp:Label ID="lblMessage" runat="server"></asp:Label>  
<br />  
<br />  
<asp:GridView ID="gvProducts" runat="server">  
</asp:GridView>   
  
  
  
**Now, copy and paste the following code in the code behind page**  
protected void btnLoadData\_Click(object sender, EventArgs e)  
{  
    // Check if the DataSet is present in the cache  
    if (Cache["Data"] == null)  
    {  
        // If the dataset is not in the cache load data from the database into the DataSet  
        string CS = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
        using (SqlConnection connection = new SqlConnection(CS))  
        {  
            SqlDataAdapter dataAdapter = new SqlDataAdapter("Select \* from tblProductInventory", connection);  
            DataSet dataset = new DataSet();  
            dataAdapter.Fill(dataset);  
  
            gvProducts.DataSource = dataset;  
            gvProducts.DataBind();  
  
            // Store the DataSet in the Cache  
            Cache["Data"] = dataset;  
            lblMessage.Text = "Data loaded from the Database";  
        }  
    }  
    // If the DataSet is in the Cache  
    else   
    {  
        // Retrieve the DataSet from the Cache and type cast to DataSet  
        gvProducts.DataSource = (DataSet)Cache["Data"];  
        gvProducts.DataBind();  
        lblMessage.Text = "Data loaded from the Cache";  
    }  
}  
  
protected void btnClearnCache\_Click(object sender, EventArgs e)  
{  
    // Check if the DataSet is present in the cache  
    if (Cache["Data"] != null)  
    {  
        // Remove the DataSet from the Cache  
        Cache.Remove("Data");  
        lblMessage.Text = "DataSet removed from the cache";  
    }  
    // If the DataSet is not in the Cache  
    else  
    {  
        lblMessage.Text = "There is nothing in the cache to remove";  
    }  
}  
  
**Now, run the application. The first time you click Load Data button**, the data will be loaded from the database, as we don't have the DataSet in the Cache yet. Once the Data is loaded into the DataSet. The DataSet is then cached. If you click the Load Data button now, then the Data will be loaded from the cache. At this point we don't need to have any connection to the Database.   
  
**To prove this stop the sql server service on your machine.**  
**1.** In the run window, type services.msc and press enter key  
**2.** In the services window, find SQL Server service  
**3.** Right click on the SQL Server service and stop it.  
  
**Since we have stopped the service**, sql server is no longer running on our machine. Now click the Load Data button. The data will be loaded from the cache. Now, clear the cache, by click on Clear Cache button. This will remove the DataSet from the cache. Now, try to load the data by clicking on Load Data button. Since, the DataSet is no longer present in the cache, and the sql server service is not running you will receive an error stating - A network-related or instance-specific error occurred while establishing a connection to SQL Server. The server was not found or was not accessible. Verify that the instance name is correct and that SQL Server is configured to allow remote connections. (provider: Named Pipes Provider, error: 40 - Could not open a connection to SQL Server)   
  
Start the service, and click the Load Data button. The data should now be loaded and cached.

Part13-

**SqlCommandBuilder automatically generates INSERT, UPDATE and DELETE sql statements based on the SELECT statement for a single table.**  
  
**For the Transact-SQL statements to be generated using SqlCommandBuilder, there are 2 steps**  
**Step 1.**Set the **"SelectCommand"**property of the **SqlDataAdapter**object  
SqlDataAdapter dataAdapter = new SqlDataAdapter();  
dataAdapter.SelectCommand = new SqlCommand("SELECT\_Query", con);  
  
**Step 2.**Create an instance of SqlCommandBuilder class and associate the SqlDataAdapter object created above using DataAdapter property of the SqlCommandBuilder object  
SqlCommandBuilder builder = new SqlCommandBuilder();  
builder.DataAdapter = dataAdapter;  
  
**Please Note:** Step 2, can also be done in single line as shown below. Here, we are passing the SqlDataAdapter instance as an argument to SqlCommandBuilder class constructor  
SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapter);  
  
**Sql script for the table used in this demo.**  
Create Table tblStudents  
(  
 ID int identity primary key,  
 Name nvarchar(50),  
 Gender nvarchar(20),  
 TotalMarks int  
)  
  
Insert into tblStudents values('Mark Hastings','Male',900)  
Insert into tblStudents values('Pam Nicholas','Female',760)  
Insert into tblStudents values('John Stenson','Male',980)  
Insert into tblStudents values('Ram Gerald','Male',990)  
Insert into tblStudents values('Ron Simpson','Male',440)  
Insert into tblStudents values('Able Wicht','Male',320)  
Insert into tblStudents values('Steve Thompson','Male',983)  
Insert into tblStudents values('James Bynes','Male',720)  
Insert into tblStudents values('Mary Ward','Female',870)  
Insert into tblStudents values('Nick Niron','Male',680)  
  
**ASPX Code:**  
<div style="font-family: Arial">  
<table border="1">  
    <tr>  
        <td>  
            Student ID  
        </td>  
        <td>  
            <asp:TextBox ID="txtStudentID" runat="server"></asp:TextBox>  
            <asp:Button ID="btnGetStudent" runat="server" Text="Load"   
                OnClick="btnGetStudent\_Click" />  
        </td>  
    </tr>  
    <tr>  
        <td>  
            Name  
        </td>  
        <td>  
            <asp:TextBox ID="txtStudentName" runat="server"></asp:TextBox>  
        </td>  
    </tr>  
    <tr>  
        <td>  
            Gender  
        </td>  
        <td>  
            <asp:DropDownList ID="ddlGender" runat="server">  
                <asp:ListItem Text="Select Gender" Value="-1"></asp:ListItem>  
                <asp:ListItem Text="Male" Value="Male"></asp:ListItem>  
                <asp:ListItem Text="Female" Value="Female"></asp:ListItem>  
            </asp:DropDownList>  
        </td>  
    </tr>  
    <tr>  
        <td>  
            Total Marks  
        </td>  
        <td>  
            <asp:TextBox ID="txtTotalMarks" runat="server"></asp:TextBox>  
        </td>  
    </tr>  
    <tr>  
        <td colspan="2">  
            <asp:Button ID="btnUpdate" runat="server" Text="Update"   
                OnClick="btnUpdate\_Click" />  
            <asp:Label ID="lblStatus" runat="server" Font-Bold="true">  
            </asp:Label>  
        </td>  
    </tr>  
</table>  
</div>  
  
**ASPX.CS Code:**  
public partial class WebForm1 : System.Web.UI.Page  
{  
    protected void Page\_Load(object sender, EventArgs e)  
    {  
    }  
  
    protected void btnGetStudent\_Click(object sender, EventArgs e)  
    {  
        string connectionString =  
            ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
        SqlConnection connection = new SqlConnection(connectionString);  
        string selectQuery = "Select \* from tblStudents where ID = " +  
            txtStudentID.Text;  
        SqlDataAdapter dataAdapter = new SqlDataAdapter(selectQuery, connection);  
  
        DataSet dataSet = new DataSet();  
        dataAdapter.Fill(dataSet, "Students");  
  
        // Store DataSet and the select query in ViewState, so they can be used  
        // later to generate the T-SQL commands using SqlCommandBuilder class  
        ViewState["DATASET"] = dataSet;  
        ViewState["SELECT\_QUERY"] = selectQuery;  
  
        if (dataSet.Tables["Students"].Rows.Count > 0)  
        {  
            DataRow dataRow = dataSet.Tables["Students"].Rows[0];  
            txtStudentName.Text = dataRow["Name"].ToString();  
            txtTotalMarks.Text = dataRow["TotalMarks"].ToString();  
            ddlGender.SelectedValue = dataRow["Gender"].ToString();  
            lblStatus.Text = "";  
        }  
        else  
        {  
            lblStatus.ForeColor = System.Drawing.Color.Red;  
            lblStatus.Text = "No record with ID = " + txtStudentID.Text;  
        }  
    }  
  
    protected void btnUpdate\_Click(object sender, EventArgs e)  
    {  
        string connectionString =  
            ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;  
        SqlConnection con = new SqlConnection(connectionString);  
  
        SqlDataAdapter dataAdapter = new SqlDataAdapter();  
        // Retrieve the Select query from ViewState and use it to build  
        // SqlCommand command object, which will then be set as the   
        // SelectCommand of the SqlDataAdapter object  
        dataAdapter.SelectCommand =   
            new SqlCommand((string)ViewState["SELECT\_QUERY"], con);  
  
        // Associate SqlDataAdapter object with SqlCommandBuilder. At this point  
        // SqlCommandBuilder should generate T-SQL statements automatically  
        SqlCommandBuilder builder = new SqlCommandBuilder(dataAdapter);  
  
        DataSet ds = (DataSet)ViewState["DATASET"];  
        DataRow dr = ds.Tables["Students"].Rows[0];  
        dr["Name"] = txtStudentName.Text;  
        dr["Gender"] = ddlGender.SelectedValue;  
        dr["TotalMarks"] = txtTotalMarks.Text;  
        dr["Id"] = txtStudentID.Text;  
  
        int rowsUpdated = dataAdapter.Update(ds, "Students");  
        if (rowsUpdated == 0)  
        {  
            lblStatus.ForeColor = System.Drawing.Color.Red;  
            lblStatus.Text = "No rows updated";  
        }  
        else  
        {  
            lblStatus.ForeColor = System.Drawing.Color.Green;  
            lblStatus.Text = rowsUpdated.ToString() + " row(s) updated";  
        }  
    }  
}