**Introduction to ADO.NET**

**Introduction to ADO.NET**

1. **What is ADO.NET?**
2. **What types of Applications use ADO.NET?**
3. **Components of ADO.NET**
4. **What is .NET Data Providers?**

**What is ADO.NET?**

ADO stands for Microsoft ActiveX Data Objects. ADO.NET is one of Microsoft’s Data Access technology. It is a part of the .Net Framework which is used to establish a connection between the .NET Application and data sources. The Data sources can be SQL Server, Oracle, MySQL, and XML, etc. ADO.NET consists of a set of classes that can be used to connect, retrieve, insert and delete data from data sources. ADO.NET mainly uses **System.Data.dll** and **System.Xml.dll.**

**What types of Applications use ADO.NET?**

ADO.NET can be used to develop any type of .NET application. The following are some of the .NET applications which you can ADO.NET Data Access Technology to interact with a data source.

1. ASP.NET Web Form Applications
2. Windows Applications
3. ASP.NET MVC Application
4. Console Applications
5. ASP.NET Web API Applications

**Components of ADO.NET**

Components are designed for data manipulation and fast access to data. **Connection, Command, DataReader, DataAdapter, DataSet, and DataView** are the components of ADO.NET that are used to perform database operations. ADO.NET has two main components that are used for accessing and manipulating data. They are as follows:

1. **Data provider and**
2. **DataSet.**

**What is .NET Data Providers?**

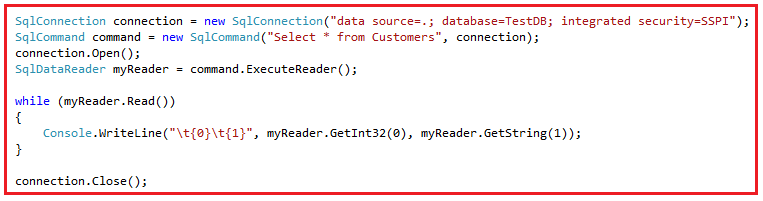
The Database can not directly execute our C# code, it only understands SQL. So, if a .NET application needs to retrieve data or to do some insert, update, and delete operations from or to a database, then the .NET application needs to

1. Connect to the Database
2. Prepare an SQL Command
3. Execute the Command
4. Retrieve the results and display them in the application

And this is possible with the help of .NET Data Providers.

**ADO.NET code to connect to SQL Server Database**

The following image shows the sample ADO.NET code which is connecting to SQL Server Database and retrieves data. If you notice in the below image, here, we are using some classes such as **SQLConnection**, **SQLCommand,** and **SQLDataReader.**These classes are called Provider classes and these classes are responsible for interacting with the database. If you further notice all the classes are prefixed with the word SQL, it means these classes are going to interact with only the SQL Server database.



All these classes are present in **System.Data.SqlClient**namespace. We can also say that the .NET data provider for the SQL Server database is **System.Data.SqlClient**.

**ADO.NET code to connect to Oracle Database**

The following code is for connecting to Oracle Database and retrieve data. If you notice, here we are using OracleConnection, OracleCommand, and OracleDataReader classes. That means all these classes have prefixed the word Oracle and these classes are used to communicate with the Oracle database.

OracleConnection connection = new OracleConnection**(**"data source=.; database=TestDB; integrated security=SSPI"**)**;

OracleCommand command = new OracleCommand**(**"Select \* from Customers", connection**)**;

connection.Open**()**;

OracleDataReader myReader = command.ExecuteReader**()**;

**while** **(**myReader.Read**())**

**{**

Console.WriteLine**(**"\t{0}\t{1}", myReader.GetInt32**(**0**)**, myReader.GetString**(**1**))**;

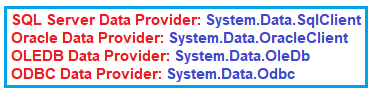
**}**

connection.Close**()**;

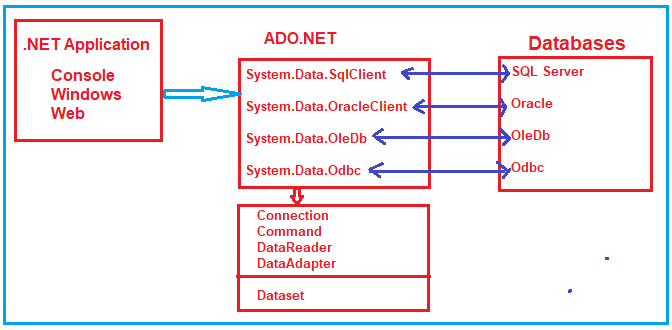
All the above classes are present in **System.Data.OracleClient** namespace. So, we can say that the .NET data provider for Oracle is **System.Data.OracleClient**.

**Note:** Similarly, if you want to communicate with OLEDB data sources such as Excel, Access, etc. then you need to use **OleDbConnection**, **OleDbCommand,** and **OleDbDataReader** classes. So, the .NET data provider for **OLEDB** data sources is **System.Data.OleDb**.

**Different .NET Data Providers**



**Please have a look at the following image to understand the data providers in a better manner.**



Note: The point that you need to remember is depending on the provider, the ADO.NET objects (**Connection, Command, DataReader, and DataAdapter**) have a different prefix as shown below.

1. **Connection**– SQLConnection, OracleConnection, OleDbConnection, OdbcConnection, etc.
2. **Command**– SQLCommand, OracleCommand, OleDbCommand, OdbcCommand, etc.
3. **DataReader**– SQLDataReader, OracleDataReader, OleDbDataReader, OdbcDataReader, etc.
4. **DataAdapter**– SQLDataAdapter, OracleDataAdapter, OleDbDataAdapter, OdbcDataAdapter, etc.

**DataSet:**

The DataSet object is not specific to provider-specific. Once you connect to a database, execute the command, and retrieve data into the .NET application. The data can then be stored in a DataSet and work independently of the database. So, it is used to access data independently from any data source. The DataSet contains a collection of one or more DataTable objects.

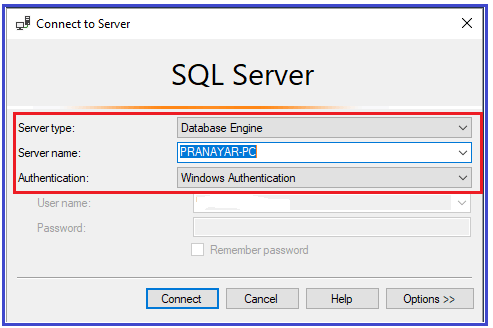
**ADO.NET using SQL Server**

**ADO.NET using SQL Server**

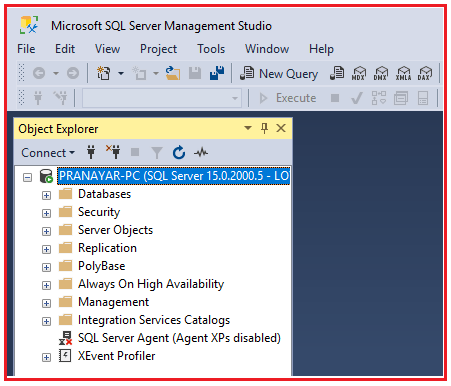
In this article, I am going to discuss **ADO.NET using SQL Server**. Please read our previous article where we discussed the [**Architecture of ADO.NET**](https://dotnettutorials.net/lesson/what-is-ado-net/). At the end of this article, you will understand how to connect with SQL Server database using ADO.NET. I hope you have SQL Server installed on your machine. We are using SQL Server Management Studio (SSMS) Tool to interact with SQL Server.

**Open SQL Server Management Studio Tool**

Once you open SSMS (SQL Server Management Studio), It will prompt you the connect to the server window. Here, you need to provide the server name and authentication details, select Database Engine as the server type and finally click on the **Connect** button as shown in the below image.

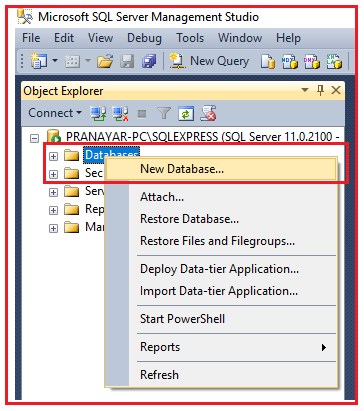


Once you click on the Connect, it will connect to the SQL Server Database and after a successful connection, it will display the following window.

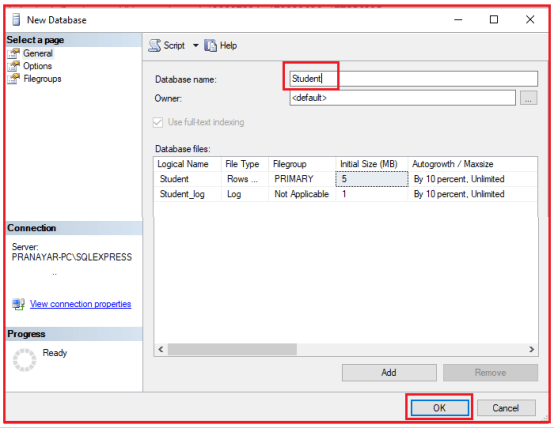


**Creating Database in SQL Server**

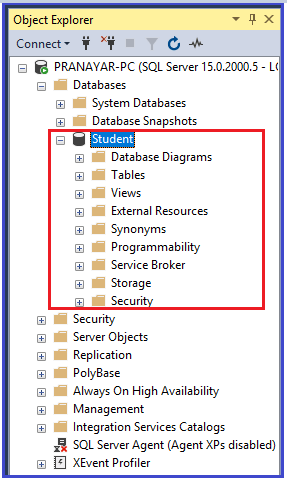
In order to create a database using GUI, you need to select the database option from object explorer and then right-click on it. It pops up an options menu and here, you need to click on the New Database option as shown in the below image.



Once you click on the **New Database** option, then it will open the following New Database window. Here, you just need to provide the database name and click on the **OK** button. Here, I created a database with the name Student. But it is up to you, you can provide any meaningful name as per your choice.



Once you click on the **OK** button, then it will create a Student database and you can see the Student database in the object explorer as shown in the below image.



That’s it. Our database part is over. Now let us move to the ado.net part.

**Establish a connection and create a table**

Once the Student Database is ready, now, let’s move and create a table (Student table) by using the ADO.NET Provider and C# code. In the below example, we are using created student database to connect.

Open visual studio 2017 (you can use any version of visual studio), then create a new .NET console application project. Once you create the project, then modify the Program.cs class file as shown below. In this article, I am not going to explain to you the code. Here in this article, I am just going to show you how to communicate with SQL Server database. From our next article onwards, I will explain each and everything in detail.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

new Program**()**.CreateTable**()**;

Console.ReadKey**()**;

**}**

**public** **void** CreateTable**()**

**{**

SqlConnection con = **null**;

**try**

**{**

// Creating Connection

con = new SqlConnection**(**"data source=.; database=student; integrated security=SSPI"**)**;

// writing sql query

SqlCommand cm = new SqlCommand**(**"create table student(id int not null, name varchar(100), email varchar(50), join\_date date)", con**)**;

// Opening Connection

con.Open**()**;

// Executing the SQL query

cm.ExecuteNonQuery**()**;

// Displaying a message

Console.WriteLine**(**"Table created Successfully"**)**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong." + e**)**;

**}**

// Closing the connection

**finally**

**{**

con.Close**()**;

**}**

**}**

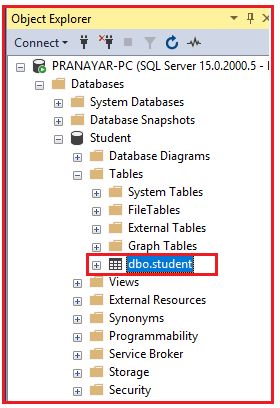
**}**

**}**

Now, execute the program and you should see the following message on the console.

Establish connection and create a table using ado.net

We can see the created table in Microsoft SQL Server Management Studio also. It shows the created table as shown below.



See, we have the Student table within the Student database. As of now, the Student table is empty. Let us insert one record into the Student table using ADO.NET and C#.

**Inserting Record using ADO.NET:**

Please modify the **Program.cs** class file as shown below. Here, we will insert a record into the student table.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

new Program**()**.InsertRecord**()**;

Console.ReadKey**()**;

**}**

**public** **void** InsertRecord**()**

**{**

SqlConnection con = **null**;

**try**

**{**

// Creating Connection

con = new SqlConnection**(**"data source=.; database=student; integrated security=SSPI"**)**;

// writing sql query

SqlCommand cm = new SqlCommand**(**"insert into student (id, name, email, join\_date) values ('101', 'Ronald Trump', 'ronald@example.com', '1/12/2017')", con**)**;

// Opening Connection

con.Open**()**;

// Executing the SQL query

cm.ExecuteNonQuery**()**;

// Displaying a message

Console.WriteLine**(**"Record Inserted Successfully"**)**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong." + e**)**;

**}**

// Closing the connection

**finally**

**{**

con.Close**()**;

**}**

**}**

**}**

**}**

Once you run the application, you will get the following output.

Inserting Record using ADO.NET

**Retrieve Record using ADO.NET**

Here, we will retrieve the inserted data from the Student table of the student database. Please modify the**Program.cs** class file as shown below.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

new Program**()**.DisplayData**()**;

Console.ReadKey**()**;

**}**

**public** **void** DisplayData**()**

**{**

SqlConnection con = **null**;

**try**

**{**

// Creating Connection

con = new SqlConnection**(**"data source=.; database=student; integrated security=SSPI"**)**;

// writing sql query

SqlCommand cm = new SqlCommand**(**"Select \* from student", con**)**;

// Opening Connection

con.Open**()**;

// Executing the SQL query

SqlDataReader sdr = cm.ExecuteReader**()**;

// Iterating Data

**while** **(**sdr.Read**())**

**{**

// Displaying Record

Console.WriteLine**(**sdr**[**"id"**]** + " " + sdr**[**"name"**]** + " " + sdr**[**"email"**])**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong." + e**)**;

**}**

// Closing the connection

**finally**

**{**

con.Close**()**;

**}**

**}**

**}**

**}**

You will get the following output when you run the above program.

Retrieve Record using ADO.NET

**Deleting Record from SQL Server database using ADO.NET**

As of now, the student table contains one record. Let us delete that record using ADO.NET and C#. Please modify the Program.cs class file code as shown below which will delete the record from the Student table.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

new Program**()**.DeleteData**()**;

Console.ReadKey**()**;

**}**

**public** **void** DeleteData**()**

**{**

SqlConnection con = **null**;

**try**

**{**

// Creating Connection

con = new SqlConnection**(**"data source=.; database=student; integrated security=SSPI"**)**;

// writing sql query

SqlCommand cm = new SqlCommand**(**"delete from student where id = '101'", con**)**;

// Opening Connection

con.Open**()**;

// Executing the SQL query

cm.ExecuteNonQuery**()**;

Console.WriteLine**(**"Record Deleted Successfully"**)**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong." + e**)**;

**}**

// Closing the connection

**finally**

**{**

con.Close**()**;

**}**

**}**

**}**

**}**

It will display the following output once you execute the program.

Deleting Record from SQL Server database using ADO.NET

Now, if you verify the student table, then you will see that the record is deleted. In this article, I didn’t explain a single line of code intentionally. I will explain each and everything in detail from our next article.

# ADO.NET SqlConnection

## ****ADO.NET SqlConnection Class in C# with Examples****

1. **What is the ADO.NET SqlConnection class?**
2. **How to instantiate SqlConnection object**
3. **Using the SqlConnection object**
4. **Why is it important to close a database connection**
5. **How to properly close a connection**
6. **What is the problem of hard-coding the connection string in the application?**
7. **How to store and retrieve the connection string from the configuration file?**

##### ****What we discussed in the Introduction Part of this article?****

Let us first recap what we discussed in our [**introduction to ADO.NET**](https://dotnettutorials.net/lesson/what-is-ado-net/) Article. We discussed the different .NET data providers. The key to understanding ADO.NET is to understand the following objects.

1. [**Connection**](https://dotnettutorials.net/lesson/ado-net-sqlconnection-class/)
2. [**Command**](https://dotnettutorials.net/lesson/ado-net-sqlcommand-class/)
3. [**DataReader**](https://dotnettutorials.net/lesson/ado-net-sqldatareader/)
4. [**DataAdapter**](https://dotnettutorials.net/lesson/ado-net-sqldataadapter/)
5. [**DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/)

In our introduction part, we discussed that **Connection, Command, DataAdapter,** and **DataReader** objects are providers specific where the **DataSet** is provider independent. That means if you are going to work with SQL Server database, then you need to use SQL-specific provider objects such as **SQLConnection**, **SqlCommand**, **SqlDataAdapter,** and **SqlDataReader** objects which belong to the **System.Data.SqlClient** namespace.

###### **Note:**

If you understand how to work with one database, then you can easily work with any other database. All you have to do is, change the provider-specific string (i.e. SQL, Oracle, Oledb, Odbc) on the Connection, Command, DataReader, and DataAdapter objects depending on the data source you are working with.

Here, in this article, I am going to discuss the **SqlConnection** object in detail. The concepts that we discuss here are applicable to all the .NET data providers.

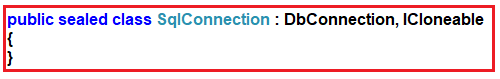
##### ****What is ADO.NET SqlConnection Class in C#?****

The **ADO.NET** **SqlConnection** class belongs to **System.Data.SqlClient** namespace and is used to establish an open connection to the SQL Server database. The most important point that you need to remember is the connection does not close implicitly even it goes out of scope. Therefore, it is always recommended and always a good programming practice to close the connection object explicitly by calling the Close() method of the connection object

**Note:** The connections should be opened as late as possible, and should be closed as early as possible as the connection is one of the most expensive resources. .

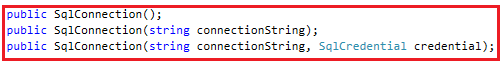
##### ****ADO.NET SqlConnection class Signature in C#:****

Following is the signature of **SqlConnection** class. As you can see, it is a sealed class and inherited from **DbConnection** class and implement the **ICloneable** interface.



##### ****SqlConnection Class Constructors:****

The ADO.NET SqlConnection class has three constructors which are shown in the below image.



Let us discuss each of these constructors in detail.

1. **SqlConnection():** It initializes a new instance of the System.Data.SqlClient.SqlConnection class
2. **SqlConnection(String connectionString):**This constructor is used to initializes a new instance of the System.Data.SqlClient.SqlConnection class when given a string that contains the connection string.
3. **SqlConnection(String connectionString, SqlCredential credential):**It is used to initializes a new instance of the System.Data.SqlClient.SqlConnection class given a connection string, that does not use Integrated Security = true and a System.Data.SqlClient.SqlCredential object that contains the user ID and password.

##### ****C# SqlConnection class Methods:****

Following are some of the important methods of the SqlConnection object.

1. **BeginTransaction():** It is used to start a database transaction and returns an object representing the new transaction.
2. **ChangeDatabase(string database):** It is used to change the current database for an open SqlConnection. Here, the parameter database is nothing but the name of the database to use instead of the current database.
3. **ChangePassword(string connectionString, string newPassword):** Changes the SQL Server password for the user indicated in the connection string to the supplied new password. Here, the parameter connectionString is the connection string that contains enough information to connect to the server that you want. The connection string must contain the user ID and the current password. The parameter newPassword is the new password to set. This password must comply with any password security policy set on the server, including minimum length, requirements for specific characters, and so on.
4. **Close():** It is used to closes the connection to the database. This is the preferred method of closing any open connection.
5. **CreateCommand():** It Creates and returns a System.Data.SqlClient.SqlCommand object associated with the System.Data.SqlClient.SqlConnection.
6. **GetSchema():**It returns schema information for the data source of this System.Data.SqlClient.SqlConnection.
7. **Open():** This method is used to open a database connection with the property settings specified by the System.Data.SqlClient.SqlConnection.ConnectionString.

##### ****How to create Connection Object in C#?****

You can create an instance of the SqlConnection class in three as there is three constructors in SqlConnection class. Here, I am going to show you the two most preferred ways of creating an instance of SqlConnection class. They are as follows:

##### ****Using the constructor which takes connection string as the parameter.****

The following image shows how to create an instance of SqlConnection class using the constructor which takes ConnectionString as the only parameter.

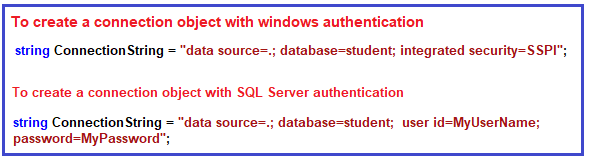
How to create Connection Object?

##### ****Using the parameterless constructor of C# SqlConnection class:****

The following image shows how to create an instance of SqlConnection class using the parameterless constructor. It is a two-step process. First, you need to create an instance of SqlConnection class using the parameterless constructor, and then using the ConnectionString property of the connection object you need to specify the connection string.

How to instantiate SqlConnection object

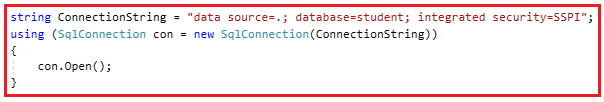
**Note:** The ConnectionString parameter is a string made up of Key/Value pairs that have the information required to create a connection object.



Here, the “**data source**” is the name or IP Address of the SQL Server that you want to connect to. If you are working with a local instance of SQL Server, then you can simply put a DOT(.). If the server is on a network, then you need to use either the Name or IP address of the server.

##### ****SqlConnection Example in C#****

Let us see an example to understand how to connect to an SQL Server database. We have created a Student database in our previous article and we will connect to that Student database. Please have a look at the following C# code which will create the connection object and then establish an open connection when the Open method is called on the connection object.



**Note:** Here, we are using the using block to close the connection automatically. If you are using the using block then you don’t require to call the close() method explicitly to close the connection. It is always recommended to close the database connection using the using block in C#.

##### ****The complete code is given below.****

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

new Program**()**.Connecting**()**;

Console.ReadKey**()**;

**}**

**public** **void** Connecting**()**

**{**

string ConnectionString = "data source=.; database=student; integrated security=SSPI";

**using** **(**SqlConnection con = new SqlConnection**(**ConnectionString**))**

**{**

con.Open**()**;

Console.WriteLine**(**"Connection Established Successfully"**)**;

**}**

**}**

**}**

**}**

**Output:**

Why is it important to close a database connection

##### ****What, if we don’t use using block?****

If you don’t use the using block to create the connection object, then you have to close the connection explicitly by calling the Close method on the connection object. In the following example, we are using try-block instead of using block and calling the Close method in finally block to close the database connection.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

new Program**()**.Connecting**()**;

Console.ReadKey**()**;

**}**

**public** **void** Connecting**()**

**{**

SqlConnection con = **null**;

**try**

**{**

// Creating Connection

string ConnectionString = "data source=.; database=student; integrated security=SSPI";

con = new SqlConnection**(**ConnectionString**)**;

con.Open**()**;

Console.WriteLine**(**"Connection Established Successfully"**)**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

**finally**

**{** // Closing the connection

con.Close**()**;

**}**

**}**

**}**

**}**

**Output:**

How to properly close a connection

Here, we hard-coded the connection strings in the application code. Let us first understand what is the problem when we hard-coded the connection string within the application code and then we will see how to overcome this problem.

##### ****The problem of hard-coding the connection string in application code:****

There are 3 problems when we hard coded the connection strings in the application code. They are as follows:

1. Let say, you move your database to a different server, then you need to change the database details in the application code itself. Once you change the application code, then you need to re-build the application as well as it also required a re-deployment which is time-consuming.
2. Again if you hard-coded the connection string in multiple places, then you need to change the connection in all the places which not only a maintenance overhead but also error-prone.
3. In real-time applications, while developing you may point to your Development database while moving to  UAT, you may have a different server for UAT and in a production environment, you need to point to the production database.

##### ****How to solve the above problems?****

We can solve the above problems, by storing the connection string in the application configuration file. The configuration file in windows or console application is **app.config** whereas for ASP.NET MVC or ASP.NET Web API application, the application configuration file is **web.config**.

##### ****How to store the connection string in the configuration file?****

As we are working with a console application, the configuration file is**app.config**. So, we need to store the connection string in the **app.config** file as shown below. Give a meaningful name to your connection string. As we are going to communicate with the SQL Server database, so, we need to provide the provider name as **System.Data.SqlClient**.

**<connectionStrings>**

**<add** name="ConnectionString"

connectionString="data source=.; database=student; integrated security=SSPI"

providerName="System.Data.SqlClient" **/>**

**</connectionStrings>**

**Note:** You need to put the above connection string inside the configuration section of the configuration file.

##### ****How to read the connection string from the app.config file?****

In order to read the connection string from the configuration file, you need to use the **ConnectionStrings** property of the **ConfigurationManager** class. The **ConfigurationManager**class is present in **System.Configuration** namespace.

##### ****Example to read the connection string from the configuration file:****

Please modify the **Program.cs** class file as shown below read the connection string from the configuration file.

**using** *System;*

**using** *System.Configuration;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = ConfigurationManager.ConnectionStrings**[**"ConnectionString"**]**.ConnectionString;

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

connection.Open**()**;

Console.WriteLine**(**"Connection Established Successfully"**)**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Problem of hard-coding the connection string in application code:

**Note:** Storing connection strings in web.config is similar to the app.config and in the same ConfigurationManager class is used to read connection string from the web.config file.

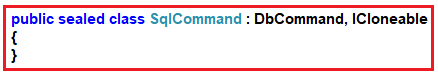
**ADO.NET SqlCommand**

**ADO.NET SqlCommand Class in C# with Examples**

1. **What is SqlCommand Class and its need?**
2. **How to create an instance of the SqlCommand class.**
3. **Understanding the constructors and methods of SqlCommand Class.**
4. **When to use ExecuteReader(), ExecuteScalar() and ExecuteNonQuery() methods of the SqlCommand object.**

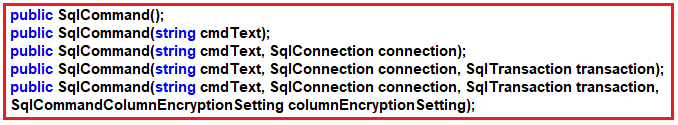
**What is ADO.NET SqlCommand Class in C#?**

The ADO.NET SqlCommand class in C# is used to store and execute the SQL statement against the SQL Server database. As you can see in the below image, the **SqlCommand** class is a sealed class and is inherited from the DbCommand class and implement the ICloneable interface. As a sealed class, it cannot be inherited.



**Constructors of ADO.NET SqlCommand Class in C#**

The SqlCommand class in C# provides the following five constructors.



Let us discuss each of these constructors in detail.

**SqlCommand():**

This constructor is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class..

**SqlCommand(string cmdText):**

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with the text of the query. Here, the cmdText is the text of the query that we want to execute.

**SqlCommand(string cmdText, SqlConnection connection):**

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with the text of the query and a System.Data.SqlClient.SqlConnection. Here, the cmdText is the text of the query that we want to execute and the parameter connection is the connection to an instance of SQL Server.

**SqlCommand(string cmdText, SqlConnection connection, SqlTransaction transaction):**

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with the text of the query, a SqlConnection instance, and the SqlTransaction instance. Here, the parameter cmdText is the text of the query. The parameter connection a SqlConnection that represents the connection to an instance of SQL Server and the parameter transaction is the SqlTransaction in which the SqlCommand executes.

**SqlCommand(string cmdText, SqlConnection connection, SqlTransaction transaction, SqlCommandColumnEncryptionSetting columnEncryptionSetting):**

It is used to initializes a new instance of the System.Data.SqlClient.SqlCommand class with specified command text, connection, transaction, and encryption setting. We already discussed the first three parameters which are the same as the previous. Here, the fourth parameter i.e. columnEncryptionSetting is the encryption setting.

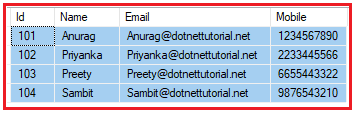
**Methods of SqlCommand Class in C#**

The SqlCommand class in C# provides the following methods.

1. **BeginExecuteNonQuery():** This method initiates the asynchronous execution of the Transact-SQL statement or stored procedure that is described by this System.Data.SqlClient.SqlCommand.
2. **Cancel():** This method tries to cancel the execution of a System.Data.SqlClient.SqlCommand.
3. **Clone():** This method creates a new System.Data.SqlClient.SqlCommand object is a copy of the current instance.
4. **CreateParameter():** This method creates a new instance of a System.Data.SqlClient.SqlParameter object.
5. **ExecuteReader():** This method Sends the System.Data.SqlClient.SqlCommand.CommandText to the System.Data.SqlClient.SqlCommand.Connection and builds a System.Data.SqlClient.SqlDataReader.
6. **ExecuteScalar():** This method Executes the query, and returns the first column of the first row in the result set returned by the query. Additional columns or rows are ignored.
7. **ExecuteNonQuery():** This method executes a Transact-SQL statement against the connection and returns the number of rows affected.
8. **Prepare():** This method creates a prepared version of the command on an instance of SQL Server.
9. **ResetCommandTimeout():** This method resets the CommandTimeout property to its default value.

**Example to understand the ADO.NET SqlCommand Object in C#:**

We are going to use the following student table to understand the SqlCommand object.



**Please use the below SQL script to create a database called StudentDB, a table called Student with the required test data.**

**CREATE** **DATABASE** StudentDB;

**GO**

**USE** StudentDB;

**GO**

**CREATE** **TABLE** Student(

Id **INT** **PRIMARY** **KEY**,

Name **VARCHAR**(100),

Email **VARCHAR**(50),

Mobile **VARCHAR**(50)

)

**GO**

**INSERT** **INTO** Student **VALUES** (101, 'Anurag', 'Anurag@dotnettutorial.net', '1234567890')

**INSERT** **INTO** Student **VALUES** (102, 'Priyanka', 'Priyanka@dotnettutorial.net', '2233445566')

**INSERT** **INTO** Student **VALUES** (103, 'Preety', 'Preety@dotnettutorial.net', '6655443322')

**INSERT** **INTO** Student **VALUES** (104, 'Sambit', 'Sambit@dotnettutorial.net', '9876543210')

**GO**

**Note:** **ExecuteReader**, **ExecuteNonQuery**, and **ExecuteScalar** are the methods that are commonly used. Let us see three examples to understand these methods.

**ExecuteReader method of SqlCommand Object in C#:**

As we already discussed this method is used to send the CommandText to the Connection and builds a SqlDataReader. When your T-SQL statement returns more than a single value (for example rows of data), then you need to use the ExecuteReader method.

Let us understand this with an example. The following example uses the **ExecuteReader** method of the SqlCommand object to executes the T-SQL statement which returns multiple rows of data.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

// Creating SqlCommand objcet

SqlCommand cm = new SqlCommand**(**"select \* from student", connection**)**;

// Opening Connection

connection.Open**()**;

// Executing the SQL query

SqlDataReader sdr = cm.ExecuteReader**()**;

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"Name"**]** + ", " + sdr**[**"Email"**]** + ", " + sdr**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

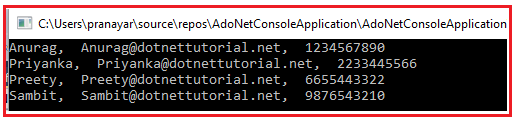
Console.ReadKey**()**;

**}**

**}**

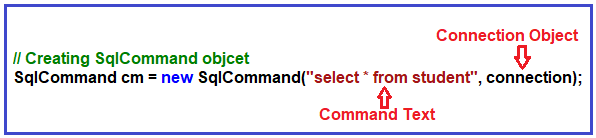
**}**

Once you execute the program, you will get the following output as expected.



**Understanding the ADO.NET SqlCommand Object in C#:**

In our example, we are creating an instance of the **SqlCommand** by using the constructor which takes two parameters as shown in the below image. The first parameter is the **command text** that we want to execute, and the second parameter is the connection object which provides the database details on which the command is going to execute.



You can also create the **SqlCommand** object using the parameterless constructor, and later you can specify the command text and connection using the **CommandText** and the **Connection** properties of the **SqlCommand** object as shown in the below example.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

// Creating SqlCommand objcet

SqlCommand cmd = new SqlCommand**()**;

cmd.CommandText = "select \* from student";

cmd.Connection = connection;

// Opening Connection

connection.Open**()**;

// Executing the SQL query

SqlDataReader sdr = cmd.ExecuteReader**()**;

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"Name"**]** + ", " + sdr**[**"Email"**]** + ", " + sdr**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**ExecuteScalar Method of SqlCommand Object in C#:**

When your T-SQL query or stored procedure returns a single(i.e. scalar) value then you need to use the **ExecuteScalar** method of the SqlCommand object in C#.

Let us understand this with an example. Now, we need to fetch the total number of records present in the Student table. As we know it is going to return a single value, so this is an ideal situation to use the **ExecuteScalar** method. The following example will retrieve the total number of records present in the **Student** table.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

// Creating SqlCommand objcet

SqlCommand cmd = new SqlCommand**(**"select count(id) from student", connection**)**;

// Opening Connection

connection.Open**()**;

// Executing the SQL query

// Since the return type of ExecuteScalar() is object, we are type casting to int datatype

**int** TotalRows = **(int)**cmd.ExecuteScalar**()**;

Console.WriteLine**(**"TotalRows in Student Table : " + TotalRows**)**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

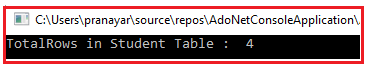
Console.ReadKey**()**;

**}**

**}**

**}**

The return type of the ExecuteScalar method is an object, so here we need to typecast it into integer type. Now, if you execute the above program, then you will get the following output.



**ExecuteNonQuery Method of ADO.NET SqlCommand Object in C#:**

When you want to perform Insert, Update or Delete operations and want to return the number of rows affected by your query then you need to use the **ExecuteNonQuery** method of the SqlCommand object in C#.

Let us understand this with an example. The following example performs an Insert, Update and Delete operations using the**ExecuteNonQuery()** method.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

SqlCommand cmd = new SqlCommand**(**"insert into Student values (105, 'Ramesh', 'Ramesh@dotnettutorial.net', '1122334455')", connection**)**;

connection.Open**()**;

**int** rowsAffected = cmd.ExecuteNonQuery**()**;

Console.WriteLine**(**"Inserted Rows = " + rowsAffected**)**;

//Set to CommandText to the update query. We are reusing the command object,

//instead of creating a new command object

cmd.CommandText = "update Student set Name = 'Ramesh Changed' where Id = 105";

rowsAffected = cmd.ExecuteNonQuery**()**;

Console.WriteLine**(**"Updated Rows = " + rowsAffected**)**;

//Set to CommandText to the delete query. We are reusing the command object,

//instead of creating a new command object

cmd.CommandText = "Delete from Student where Id = 105";

rowsAffected = cmd.ExecuteNonQuery**()**;

Console.WriteLine**(**"Deleted Rows = " + rowsAffected**)**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

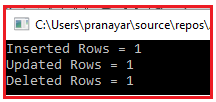
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**ADO.NET SqlDataReader**

**ADO.NET SqlDataReader in C# with Examples**

1. **What is SqlDataReader Class and its need?**
2. **How to create an instance of the SqlDataReader class.**
3. **How to read data from SqlDataReader object?**

**What is ADO.NET SqlDataReader Class in C#?**

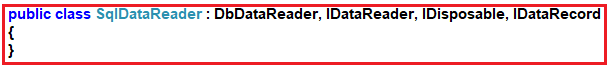
The ADO.NET SqlDataReader class in C# is used to read data from the SQL Server database in the most efficient manner. It **reads data in the forward-only** stream. It means, once it read a record, it will then read the next record, there is no way to go back and read the previous record.

The **SqlDataReader is connection-oriented**. It means it requires an open or active connection to the data source while reading the data. The data is available as long as the connection with the database exists

**SqlDataReader is read-only**. It means it is also not possible to change the data using SqlDataReader. You also need to open and close the connection explicitly.

**SqlDataReader Class Signature:**

If you look at the following image, you will see that this class is inherited from **DbDataReader** class and implements the **IDisposable** interface.



**ADO.NET SqlDataReader Class Properties in C#:**

The SqlDataReader class provides the following properties.

1. **Connection**: It gets the System.Data.SqlClient.SqlConnection associated with the System.Data.SqlClient.SqlDataReader.
2. **Depth**: It gets a value that indicates the depth of nesting for the current row.
3. **FieldCount**: It gets the number of columns in the current row.
4. **HasRows**: It gets a value that indicates whether the System.Data.SqlClient.SqlDataReader contains one or more rows.
5. **IsClosed**: It retrieves a Boolean value that indicates whether the specified System.Data.SqlClient.SqlDataReader instance has been closed.
6. **RecordsAffected**: It gets the number of rows changed, inserted, or deleted by the execution of the Transact-SQL statement.
7. **VisibleFieldCount**: It gets the number of fields in the System.Data.SqlClient.SqlDataReader that is not hidden.
8. **Item[String]**: It gets the value of the specified column in its native format given the column name.
9. **Item[Int32]**: It gets the value of the specified column in its native format given the column ordinal.

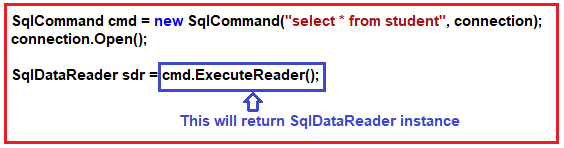
**ADO.NET SqlDataReader Class Methods in C#**:

The SqlDataReader class provides the following methods.

1. **Close():** It closes the SqlDataReader object.
2. **GetBoolean(int i):** It gets the value of the specified column as a Boolean. Here, parameter i is the zero-based column ordinal.
3. **GetByte(int i):** It gets the value of the specified column as a byte. Here, parameter i is the zero-based column ordinal.
4. **GetChar(int i):** It gets the value of the specified column as a single character. Here, parameter i is the zero-based column ordinal.
5. **GetDateTime(int i):** It gets the value of the specified column as a System.DateTime object. Here, parameter i is the zero-based column ordinal.
6. **GetDecimal(int i):** It gets the value of the specified column as a System.Decimal object. Here, parameter i is the zero-based column ordinal.
7. **GetDouble(int i):** It gets the value of the specified column as a double-precision floating-point number. Here, parameter i is the zero-based column ordinal.
8. **GetFloat(int i):** It gets the value of the specified column as a single-precision floating-point number. Here, parameter i is the zero-based column ordinal.
9. **GetName(int i):** It gets the name of the specified column. Here, parameter i is the zero-based column ordinal.
10. **GetSchemaTable():** It returns a System.Data.DataTable that describes the column metadata of the System.Data.SqlClient.SqlDataReader
11. **GetValue(int i):** It gets the value of the specified column in its native format. Here, parameter i is the zero-based column ordinal.
12. **GetValues(object[] values):** It Populates an array of objects with the column values of the current row. Here, the parameter values is an array of System.Object into which to copy the attribute columns.
13. **NextResult():** It advances the data reader to the next result when reading the results of batch Transact-SQL statements.
14. **Read():** It Advances the System.Data.SqlClient.SqlDataReader to the next record and returns true if there are more rows; otherwise false.

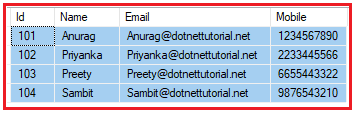
**How to create an instance of the ADO.NET SqlDataReader class in C#?**

You can not create the instance of SqlDataReader using the new keyword. Then the question is how we get or create the instance of SqlDataReader class. In order to create the instance of SqlDataReader class, what you need to do is, call the **ExecuteReader** method of the **SqlCommand** object which will return an instance of SqlDataReader class as shown in the below image.



**Example to understand the C# SqlDataReader Object in ADO.NET:**

We are going to use the following student table to understand the SqlDataReader object in C#.



**Please use the below SQL script to create a database called StudentDB, a table called Student with the required test data.**

**CREATE** **DATABASE** StudentDB;

**GO**

**USE** StudentDB;

**GO**

**CREATE** **TABLE** Student(

Id **INT** **PRIMARY** **KEY**,

Name **VARCHAR**(100),

Email **VARCHAR**(50),

Mobile **VARCHAR**(50)

)

**GO**

**INSERT** **INTO** Student **VALUES** (101, 'Anurag', 'Anurag@dotnettutorial.net', '1234567890')

**INSERT** **INTO** Student **VALUES** (102, 'Priyanka', 'Priyanka@dotnettutorial.net', '2233445566')

**INSERT** **INTO** Student **VALUES** (103, 'Preety', 'Preety@dotnettutorial.net', '6655443322')

**INSERT** **INTO** Student **VALUES** (104, 'Sambit', 'Sambit@dotnettutorial.net', '9876543210')

**GO**

**Example: Using SqlDataReader in C#**

We need to fetch all the data from the student table and need to display it in the console using SqlDataReader. The following code exactly does the same thing. In the below example, we use the **Read()** method of the SqlDataReader object to loop through the items of the **SqlDataReader** object. The Read method returns true as long as there are rows to read from the SqlDataReader object. If there are no more rows to read, then this method will simply return false.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

// Creating the command object

SqlCommand cmd = new SqlCommand**(**"select \* from student", connection**)**;

// Opening Connection

connection.Open**()**;

// Executing the SQL query

SqlDataReader sdr = cmd.ExecuteReader**()**;

//Looping through each record

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"Name"**]** + ", " + sdr**[**"Email"**]** + ", " + sdr**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

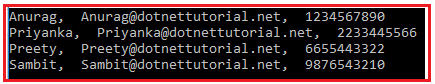
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Note:** The DataReader object increases the performance of the application as well as reduces the system overheads and the reason for this is, one row at a time is stored in memory.

ADO.NET SqlDataAdapter

**ADO.NET SqlDataAdapter in C# with Examples**

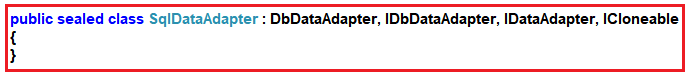
1. **What is ADO.NET SqlDataAdapter?**
2. **Understanding Constructors, Methods, and Properties of  SqlDataAdapter.**
3. **How to create an instance of SqlDataAdapter class in ADO.NET?**
4. **SqlDataAdapter using Stored Procedure.**
5. **How to call a stored procedure using SqlDataAdapter?**

**What is ADO.NET SqlDataAdapter in C#?**

The SqlDataAdapter in C# works as a bridge between a DataSet and a data source (SQL Server Database) to retrieve data. The SqlDataAdapter is a class that represents a set of SQL commands and a database connection. It can be used to fill the DataSet and update the data source.

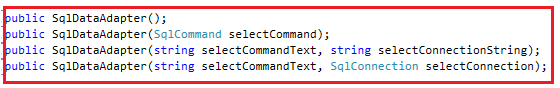
**Signature of SqlDataAdapter in C#**

As you can see in the below image, the **SqlDataAdapter**class is a sealed class so it cannot be inherited. Again is inherited from DbDataAdapter class and implement the IDbDataAdapter, IDataAdapter and ICloneable interface.



**Constructors of ADO.NET SqlDataAdapter class in C#:**

The SqlDataAdapter class provides the following constructors.



1. **SqlDataAdapter():** Initializes a new instance of the SqlDataAdapter class.
2. **SqlDataAdapter(SqlCommand selectCommand):**Initializes a new instance of the SqlDataAdapter class with the specified SqlCommand. Here, the selectCommand can be a Transact-SQL SELECT statement or a stored procedure.
3. **SqlDataAdapter(string selectCommandText, string selectConnectionString):** Initializes a new instance of the SqlDataAdapter class with a the command and a connection string. Here, the selectCommandText can be a Transact-SQL SELECT statement or a stored procedure.
4. **SqlDataAdapter(string selectCommandText, SqlConnection selectConnection)**: Initializes a new instance of the SqlDataAdapter class with a the command and a connection string. Here, the selectCommandText can be a Transact-SQL SELECT statement or a stored procedure. If your connection string does not use Integrated Security = true, you can use System.Data.SqlClient.SqlCredential to pass the user ID and password more securely than by specifying the user ID and password as text in the connection string.

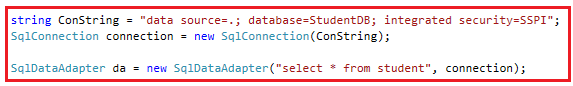
**Methods of ADO.NET SqlDataAdapter class in C#:**

The C# SqlDataAdapter class provides the following methods.

1. **CloneInternals():** It is used to create a copy of this instance of DataAdapter.
2. **Dispose(Boolean):** It is used to release the unmanaged resources used by the DataAdapter.
3. **Fill(DataSet):** It is used to add rows in the DataSet to match those in the data source.
4. **FillSchema(DataSet, SchemaType, String, IDataReader):** It is used to add a DataTable to the specified DataSet.
5. **GetFillParameters():** It is used to get the parameters set by the user when executing an SQL SELECT statement.
6. **ResetFillLoadOption():** It is used to reset FillLoadOption to its default state.
7. **ShouldSerializeAcceptChangesDuringFill()**: It determines whether the
8. **ShouldSerializeFillLoadOption():** It determines whether the FillLoadOption property should be persisted or not.
9. **ShouldSerializeTableMappings():** It determines whether one or more DataTableMapping objects exist or not.
10. **Update(DataSet):** It is used to call the respective INSERT, UPDATE, or DELETE statements.

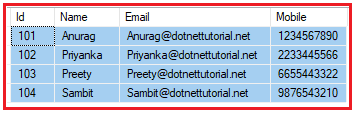
**How to create an instance of C# SqlDataAdapter class in ADO.NET?**

In order to create an instance of the SqlDataAdapter class, we need to specify two things. The SQL command that we want to execute and the connection on which we want to execute the command. Following is the syntax to create an instance of the SqlDataAdapter class.



**Example to understand the SqlDataAdapter in C#:**

We are going to use the following student table to understand the SqlDataAdapter object.



**Please use the below SQL script to create a database called StudentDB, a table called Student with the required test data.**

**CREATE** **DATABASE** StudentDB;

**GO**

**USE** StudentDB;

**GO**

**CREATE** **TABLE** Student(

Id **INT** **PRIMARY** **KEY**,

Name **VARCHAR**(100),

Email **VARCHAR**(50),

Mobile **VARCHAR**(50)

)

**GO**

**INSERT** **INTO** Student **VALUES** (101, 'Anurag', 'Anurag@dotnettutorial.net', '1234567890')

**INSERT** **INTO** Student **VALUES** (102, 'Priyanka', 'Priyanka@dotnettutorial.net', '2233445566')

**INSERT** **INTO** Student **VALUES** (103, 'Preety', 'Preety@dotnettutorial.net', '6655443322')

**INSERT** **INTO** Student **VALUES** (104, 'Sambit', 'Sambit@dotnettutorial.net', '9876543210')

**GO**

**Example: Using ADO.NET SqlDataAdapter in C#**

We need to fetch all the data from the student table and need to display it in the console using SqlDataAdapter. Let us first write the code and then we will understand the code. Following is the code.

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"select \* from student", connection**)**;

//Using Data Table

DataTable dt = new DataTable**()**;

da.Fill**(**dt**)**;

Console.WriteLine**(**"Using Data Table"**)**;

**foreach** **(**DataRow row in dt.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** +", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**(**"---------------"**)**;

//Using DataSet

DataSet ds = new DataSet**()**;

da.Fill**(**ds, "student"**)**;

Console.WriteLine**(**"Using Data Set"**)**;

**foreach** **(**DataRow row in ds.Tables**[**"student"**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

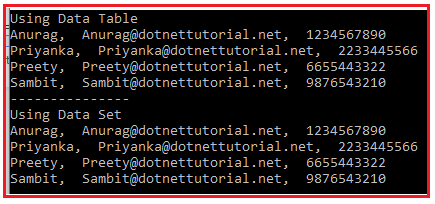
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Code Explanation:**

Here we create an instance of SqlDataAdapter class using the constructor which takes two parameters i.e. the SqlCommandText and the Connection object. Then we create an instance of DataSet and Datatable object. Both DataSet and DataTable are in-memory data stores, that can store tables, just like a database. We will discuss DataTable and DataSet in our next article.

Then we call the Fill() method of the DataAdapter class. This method does most of the work behind us. It opens the connection to the database, executes the SQL command, fills the dataset and datatable with the data, and closes the connection. This method handles the Opening and closing of the database connections automatically for us. The connection is kept open only as long as it is needed.

Finally, we are using DataRow to loop through each record and print the data on the console. We will also discuss DataRow in detail in our upcoming articles.

**C# SqlDataAdapter using SQL Server Stored Procedure:**

We will discuss how to call Stored Procedure in ADO.NET in detail in our upcoming articles. Here, in this article, I am going to show you a simple example to make you understand how to call stored procedure using SqlDataAdapter.

**Creating Stored Procedure:**

First, create the following stored procedure in the studentDB database.

**CREATE** **PROCEDURE** spGetStudents

**AS**

**BEGIN**

**SELECT** Id, Name, Email, Mobile

**FROM** Student

**END**

**How to call a stored procedure using SqlDataAdapter in C#?**

In order to execute a stored procedure using SqlDataAdapter, then you just need to specify the name of the procedure instead of the in-line SQL statement and then you have to specify the command type as StoredProcedure using the command type property of the SqlDataAdapter object as shown in the below image.

How to call a stored procedure using SqlDataAdapter

**The complete code is given below:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"spGetStudents", connection**)**;

da.SelectCommand.CommandType = CommandType.StoredProcedure;

DataTable dt = new DataTable**()**;

da.Fill**(**dt**)**;

**foreach** **(**DataRow row in dt.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** +", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

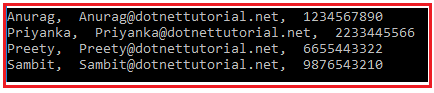
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**ADO.NET DataTable**

**ADO.NET DataTable**

1. **What is ADO.NET DataTable?**
2. **Understanding Properties, Methods, and Constructors of DataTable class in C#.**
3. **How to create a DataTable in C#?**

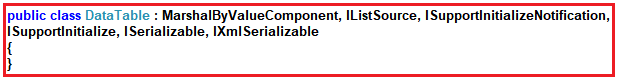
**What is ADO.NET DataTable in C#?**

Like a table in SQL, the DataTable is also going to represent the relational data in tabular form and this data is going to be store in memory. When you an create instance of DataTable, by default, it does not have table schema i.e. it does not have any columns or constraints by default. You can create a table schema by adding columns and constraints to the table. Once you define the schema (i.e. columns and constraints) for the DataTable, then only you can add rows to the data table. In order to use DataTable, you must have to include the **System.Data**namespace.

**Note:** The **ADO.NET DataTable** is a central object which can be used independently or can be used by other objects such as DataSet and the DataView.

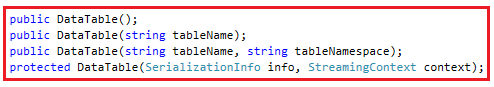
**Signature of DataTable in C#:**

The signature of the DataTable class is shown in the below image.



**Constructors of ADO.NET DataTable class in C#:**

The DataTable class provides the following four constructors.



**Let us discuss each of these constructors.**

1. **DataTable():** This constructor is used to initialize a new instance of the System.Data.DataTable class with no arguments.
2. **DataTable(string tableName):** It is used to initializes a new instance of the System.Data.DataTable class with the specified table name. Here, the Parameters tableName is the name to give the table. If tableName is null or an empty string, a default name is given when added to the System.Data.DataTableCollection.
3. **DataTable(SerializationInfo info, StreamingContext context):** This constructor is used to initializes a new instance of the System.Data.DataTable class with the System.Runtime.Serialization.SerializationInfo and the System.Runtime.Serialization.StreamingContext. Here, the parameter info specifies the data needed to serialize or deserialize an object and the parameter context specifies the source and destination of a given serialized stream.
4. **DataTable(string tableName, string tableNamespace):** It is used to initializes a new instance of the System.Data.DataTable class using the specified table name and namespace. Here, the parameter tableName specifies the name to give the table. If tableName is null or an empty string, a default name is given when added to the System.Data.DataTableCollection. The second parameter i.e. tableNamespace specifies the namespace for the XML representation of the data stored in the DataTable.

**Properties of ADO.NET DataTable in C#:**

The ADO.NET DataTable class in C# provides the following properties.

1. **Columns**: It is used to get the collection of columns that belong to this table.
2. **Constraints**: t is used to get the collection of constraints maintained by this table.
3. **DataSet**: It is used to get the DataSet to which this table belongs.
4. **DefaultView**: It is used to get a customized view of the table that may include a filtered view.
5. **HasErrors**: It is used to get a value indicating whether there are errors in any of the rows in the table of the DataSet.
6. **MinimumCapacity**: It is used to get or set the initial starting size for this table.
7. **PrimaryKey**: It is used to get or set an array of columns that function as primary keys for the data table.
8. **Rows**: It is used to get the collection of rows that belong to this table.
9. **TableName**: It is used to get or set the name of the DataTable.

**Methods of C# DataTable in ADO.NET:**

The C# DataTable class provides the following methods

1. **AcceptChanges()**: It is used to commit all the changes made to this table.
2. **Clear()**: It is used to clear the DataTable of all data.
3. **Clone()**: It is used to clone the structure of the DataTable.
4. **Copy():** It is used to copy both the structure and data of the DataTable.
5. **CreateDataReader():** It is used to returns a DataTableReader corresponding to the data within this DataTable.
6. **CreateInstance():** It is used to create a new instance of DataTable.
7. **GetRowType():** It is used to get the row type.
8. **GetSchema()**: It is used to get the schema of the table.
9. **ImportRow(DataRow):** It is used to copy a DataRow into a DataTable.
10. **Load(IDataReader):** It is used to fill a DataTable with values from a data source using the supplied IDataReader.
11. **Merge(DataTable, Boolean):** It is used to merge the specified DataTable with the current DataTable.
12. **NewRow():** It is used to create a new DataRow with the same schema as the table.
13. **Select()**: It is used to get an array of all DataRow objects.
14. **WriteXml(String):** It is used to write the current contents of the DataTable as XML using the specified file.

**How to create a DataTable in C#?**

In order to create a DataTable in C#, first, we need to create an instance of the DataTable class, and then we need to add DataColumn objects that define the type of data to be held and insert DataRow objects that contain the data. Let us discuss this step by step.

**Step1: Creating DataTable instance**

Please have look at the following image. Here, we are using the constructor which takes the table name as a parameter.

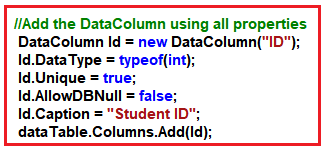
How to create a DataTable in C#?

The above code will create an empty data table for which the TableName property is set to Student. Later, you can use this property to access this data table from a DataTableCollection. Once the Data Table is created, the next important step is to add a data column and define the schema for the columns.

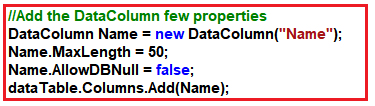
**Step2: Adding DataColumn and Defining Schema**

A DataTable is actually a collection of DataColumn objects which is referenced by the Columns property of the data table. A DataTable object is useless until it has a schema. You can create the schema by adding DataColumn objects and setting the constraints of columns. As we already know from SQL’s point of view, Constraints are basically used to maintain data integrity.

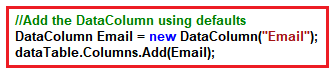
Let us see how to Create DataColumn and set the schema. The following image shows creating a data column using all the available properties.



The following image shows adding Data Column using few properties.



The following image shows creating a Data Column with the default properties.



**Creating Primary Key Column in Datatable:**

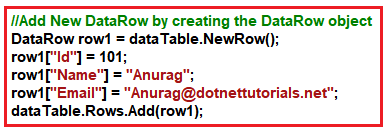
Like SQL, the primary key of a DataTable object also consists of a column or columns that make up a unique identity for each data row. The following image shows how to set the PrimaryKey property on the Id column of the Student DataTable object.

Creating Primary Key Column in Datatable

**Creating DataRow Objects in C#:**

Once you created the DataColumns for the DataTable object, then you can populate the DataTable object by adding DataRow objects. You need to use the DataRow object and its properties and methods to retrieve, insert, update and delete the values in the DataTable.

The DataRowCollection represents the actual DataRow objects in the DataTable and it has an Add method that accepts a DataRow object. The Add method is also overloaded to accept an array of objects instead of a DataRow object. The following image shows how to create and add data into the Student DataTable object.

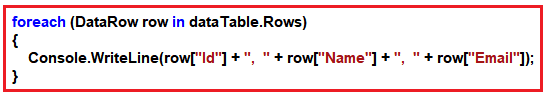


You can also add a new DataRow by simply adding the values as shown in the below image.

ADO.NET DataTable Example in C#

**Iterating the DataTable in C#:**

You can use a for each loop to loop through the rows and columns of a data table. The following image shows how to enumerate through the rows and columns of a data table.



**The complete code is given below:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

//Creating data table instance

DataTable dataTable = new DataTable**(**"Student"**)**;

//Add the DataColumn using all properties

DataColumn Id = new DataColumn**(**"ID"**)**;

Id.DataType = typeof**(int)**;

Id.Unique = **true**;

Id.AllowDBNull = **false**;

Id.Caption = "Student ID";

dataTable.Columns.Add**(**Id**)**;

//Add the DataColumn few properties

DataColumn Name = new DataColumn**(**"Name"**)**;

Name.MaxLength = 50;

Name.AllowDBNull = **false**;

dataTable.Columns.Add**(**Name**)**;

//Add the DataColumn using defaults

DataColumn Email = new DataColumn**(**"Email"**)**;

dataTable.Columns.Add**(**Email**)**;

//Setting the Primary Key

dataTable.PrimaryKey = new DataColumn**[]** **{** Id **}**;

//Add New DataRow by creating the DataRow object

DataRow row1 = dataTable.NewRow**()**;

row1**[**"Id"**]** = 101;

row1**[**"Name"**]** = "Anurag";

row1**[**"Email"**]** = "Anurag@dotnettutorials.net";

dataTable.Rows.Add**(**row1**)**;

//Adding new DataRow by simply adding the values

dataTable.Rows.Add**(**102, "Mohanty", "Mohanty@dotnettutorials.net"**)**;

**foreach** **(**DataRow row in dataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Email"**])**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

DataTable in C#

**DataColumn Properties**

In the above example, we have used some of the properties of DataColumn. Following is the list of all available of DataColumn classes.

1. **AllowDBNull**: This property is used to gets or sets a value that indicates whether the column will accept null values or not.
2. **Autoincrement**: This property is used when you want to increment the column values automatically.
3. **AutoincrementSeed**: This property is used to get or sets the starting value for the auto-incremented column.
4. **AutoincrementStep**: This property is used to get or sets the increment used by a column with its Autoincrement property set to true.
5. **Caption**: his property is used to get or sets the caption for the column.
6. **ColumnName**: This property is used to get or sets the name of the column.
7. **Expression**: This property is used to get or sets the expression used to filter rows, calculate the values in a column, or create an aggregate column.
8. **MaxLength**: This property is used to get or sets the maximum length of a text column.
9. **Unique**: This property is used to get or sets a value that indicates whether the values in each row of the column must be unique.

In our example, we discussed most of the above properties. Let us understand one important property i.e. **Autoincrement.**

**Example to understand Autoincrement Column:**

In the following example, we set the Id column of the data table as auto-increment.

**using** *System;*

**using** *System.Data;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

//Creating data table instance

DataTable dataTable = new DataTable**(**"Student"**)**;

DataColumn Id = new DataColumn

**{**

ColumnName = "Id",

DataType = System.Type.GetType**(**"System.Int32"**)**,

AutoIncrement = **true**,

AutoIncrementSeed = 1000,

AutoIncrementStep = 10

**}**;

dataTable.Columns.Add**(**Id**)**;

//Add the DataColumn few properties

DataColumn Name = new DataColumn**(**"Name"**)**;

Name.MaxLength = 50;

Name.AllowDBNull = **false**;

dataTable.Columns.Add**(**Name**)**;

//Add the DataColumn using defaults

DataColumn Email = new DataColumn**(**"Email"**)**;

dataTable.Columns.Add**(**Email**)**;

//Add New DataRow by creating the DataRow object

DataRow row1 = dataTable.NewRow**()**;

row1**[**"Name"**]** = "Anurag";

row1**[**"Email"**]** = "Anurag@dotnettutorials.net";

dataTable.Rows.Add**(**row1**)**;

//Adding new DataRow by simply adding the values

//Supply null for auto increment column

dataTable.Rows.Add**(null**, "Mohanty", "Mohanty@dotnettutorials.net"**)**;

**foreach** **(**DataRow row in dataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Email"**])**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

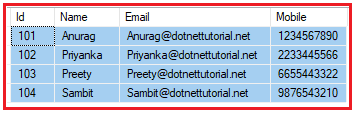
Example to understand Autoincrement Column

**DataTable Methods in C#**

**DataTable Methods in C# with Examples**

**Example to understand DataTable Methods in C# using SQL Server:**

We are going to use the following student table to understand the SqlDataAdapter object.



**Please use the below SQL script to create a database called StudentDB, a table called Student with the required test data.**

**CREATE** **DATABASE** StudentDB;

**GO**

**USE** StudentDB;

**GO**

**CREATE** **TABLE** Student(

Id **INT** **PRIMARY** **KEY**,

Name **VARCHAR**(100),

Email **VARCHAR**(50),

Mobile **VARCHAR**(50)

)

**GO**

**INSERT** **INTO** Student **VALUES** (101, 'Anurag', 'Anurag@dotnettutorial.net', '1234567890')

**INSERT** **INTO** Student **VALUES** (102, 'Priyanka', 'Priyanka@dotnettutorial.net', '2233445566')

**INSERT** **INTO** Student **VALUES** (103, 'Preety', 'Preety@dotnettutorial.net', '6655443322')

**INSERT** **INTO** Student **VALUES** (104, 'Sambit', 'Sambit@dotnettutorial.net', '9876543210')

**GO**

**Example: Using DataTable in C#**

We need to fetch all the data from the student table and then need to store the data in a data table and finally using a for each loop to display the data in the console. The following code exactly does the same thing. In the following example, we are creating a data table and filling the data table using the Fill method of the SqlDataAdapter object.

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"select \* from student", connection**)**;

DataTable dt = new DataTable**()**;

da.Fill**(**dt**)**;

**foreach** **(**DataRow row in dt.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

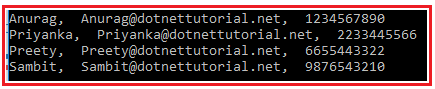
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Copying and Cloning the DataTable in C#:**

If you want to create a full copy of a data table, then you need to use the Copy method of the DataTable object which will copy not only the DataTable data but also its schema. But if you want to copy the data table schema without data, then you need to use the Clone method of the data table. The following example shows the use of both the clone and copy method.

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"select \* from student", connection**)**;

DataTable originalDataTable = new DataTable**()**;

da.Fill**(**originalDataTable**)**;

Console.WriteLine**(**"Original Data Table : originalDataTable"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Copy Data Table : copyDataTable"**)**;

DataTable copyDataTable = originalDataTable.Copy**()**;

**if** **(**copyDataTable != **null)**

**{**

**foreach** **(**DataRow row in copyDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"Clone Data Table : cloneDataTable"**)**;

DataTable cloneDataTable = originalDataTable.Clone**()**;

**if** **(**cloneDataTable.Rows.Count **>** 0**)**

**{**

**foreach** **(**DataRow row in cloneDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**else**

**{**

Console.WriteLine**(**"cloneDataTable is Empty"**)**;

Console.WriteLine**(**"Adding Data to cloneDataTable"**)**;

cloneDataTable.Rows.Add**(**101, "Test1", "Test1@dotnettutorial.net", "1234567890"**)**;

cloneDataTable.Rows.Add**(**101, "Test2", "Test1@dotnettutorial.net", "1234567890"**)**;

**foreach** **(**DataRow row in cloneDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

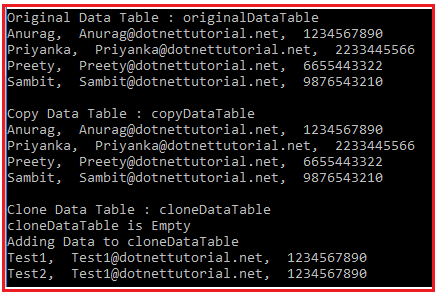
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Deleting Data Row from a DataTable in C#:**

You can delete a DataRow from the DataRowCollection by calling the **Remove**method of the **DataRowCollection**, or by calling the Delete method of the DataRow object.

The **Remove**method will remove the row from the collection whereas the **Delete**method marks the DataRow for removal. The actual removal will occur when you call the **AcceptChanges**method. If you want to rollback, then you need to use the **RejectChanges**method which will roll back to the previous state. The **RejectChanges**method will copy the Original data row version to the Current data row version.

**Delete Method Example:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"select \* from student", connection**)**;

DataTable originalDataTable = new DataTable**()**;

da.Fill**(**originalDataTable**)**;

Console.WriteLine**(**"Before Deletion"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

**if** **(**Convert.ToInt32**(**row**[**"Id"**])** == 101**)**

**{**

row.Delete**()**;

Console.WriteLine**(**"Row with Id 101 Deleted"**)**;

**}**

**}**

originalDataTable.AcceptChanges**()**;

Console.WriteLine**()**;

Console.WriteLine**(**"After Deletion"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

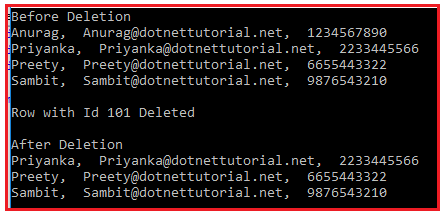
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**Remove method example:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"select \* from student", connection**)**;

DataTable originalDataTable = new DataTable**()**;

da.Fill**(**originalDataTable**)**;

Console.WriteLine**(**"Before Deletion"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

**foreach** **(**DataRow row in originalDataTable.Select**())**

**{**

**if** **(**Convert.ToInt32**(**row**[**"Id"**])** == 101**)**

**{**

originalDataTable.Rows.Remove**(**row**)**;

Console.WriteLine**(**"Row with Id 101 Deleted"**)**;

**}**

**}**

Console.WriteLine**()**;

Console.WriteLine**(**"After Deletion"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

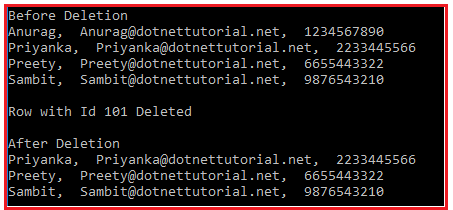
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**RejectChanges Method example:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlDataAdapter da = new SqlDataAdapter**(**"select \* from student", connection**)**;

DataTable originalDataTable = new DataTable**()**;

da.Fill**(**originalDataTable**)**;

Console.WriteLine**(**"Before Deletion"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

**if** **(**Convert.ToInt32**(**row**[**"Id"**])** == 101**)**

**{**

row.Delete**()**;

Console.WriteLine**(**"Row with Id 101 Deleted"**)**;

**}**

**}**

//Rollbacking the Data

originalDataTable.RejectChanges**()**;

Console.WriteLine**()**;

Console.WriteLine**(**"Rollbacking the Changes"**)**;

**foreach** **(**DataRow row in originalDataTable.Rows**)**

**{**

Console.WriteLine**(**row**[**"Name"**]** + ", " + row**[**"Email"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

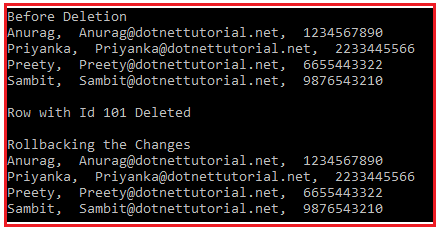
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



**ADO.NET DataSet**

**ADO.NET DataSet in C# with Examples**

**What is ADO.NET DataSet in C#?**

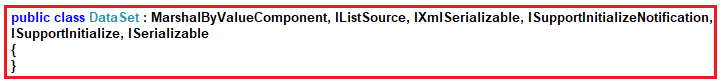
The DataSet represents a subset of the database in memory. That means the ADO.NET DataSet is a collection of data tables that contains the relational data in memory in tabular format.

It does not require a continuous open or active connection to the database. The DataSet is based on the disconnected architecture. This is the reason why it is used to fetch the data without interacting with any data source. We will discuss the disconnected architecture of the data set in our upcoming articles.

**Note:**The ADO.NET DataSet class is the core component for providing data access in a distributed and disconnected environment. The ADO.NET DataSet class belongs to the **System.Data** namespace.

**Signature of DataSet in C#:**

The signature of the DataSet class is shown in the below image.



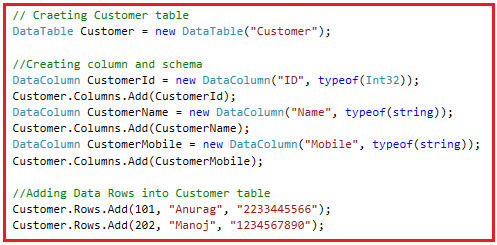
Let us first see an example to create and use a DataSet object and then we will discuss the different constructors, properties, and methods of the DataSet object.

**Example to understand DataSet in C#:**

Let us understand how to create and use DataSet with an example. Here, we want to create two data tables (Customers and Orders) and then we want to add both these data tables into the DataSet and then we want to log the data into the console.

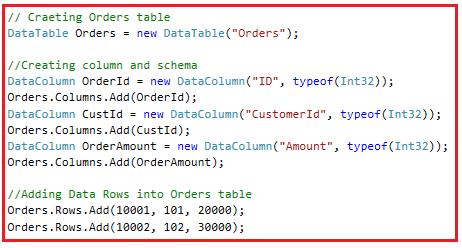
**Creating Customers Data Table:**

Please have a look at the below image. As you can see, here, we created one **DataTable**with the name **Customer**. Then we created three data columns and added these three columns into the Customer data table. Finally, we created two data rows and add these two data rows into the Customer data table.



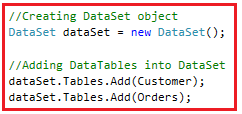
**Creating Orders Data Table:**

Please have a look at the following image. Here, you can see, we created the **DataTable**with the name **Orders**. Then we created three data columns (Id, CustomerId, and Amount) and add these three columns into the Orders table. Finally, we created two data rows and add these data rows into the Orders table.



**Creating DataSet with DataTable:**

As we already discussed the **DataSet**is a collection of DataTables. So, let’s create a DataSet object and then add the two data tables (Customers and Orders) into the DataSet. Please have a look at the following image. Here, first, we created an instance of the DataSet and then add the two data tables using the Tables property of the DataSet object.

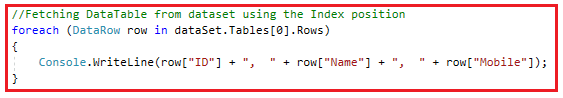


**Fetch DataTable from DataSet:**

Now, let us see how to fetch the data table from the dataset. You can fetch the data table from a dataset in two ways i.e. using the index position and using the table name (if provided).

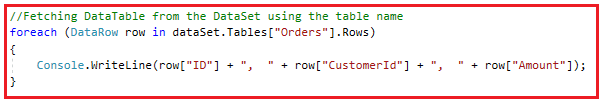
**Fetching DataTable from DataSet using index position:**

As we first add the Customers table to DataSet, so the Customer table Index position is 0. If you want to iterate through the Customers table data, then you could use a for-each loop to iterate as shown in the below image.



**Fetching DataTable From DataSet using Name:**

The second data table that we added to the dataset is Orders and it will be added at index position 1. Further, if you notice while creating the datatable we have provided a name for the data table i.e. Orders. Now, if you want to fetch the data table from the dataset, then you can use the name instead of the index position. The following image shows how to fetch the data table using the name and looping through the data using a for each loop.



**The complete code is given below.**

**using** *System;*

**using** *System.Data;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

// Craeting Customer table

DataTable Customer = new DataTable**(**"Customer"**)**;

//Creating column and schema

DataColumn CustomerId = new DataColumn**(**"ID", typeof**(**Int32**))**;

Customer.Columns.Add**(**CustomerId**)**;

DataColumn CustomerName = new DataColumn**(**"Name", typeof**(**string**))**;

Customer.Columns.Add**(**CustomerName**)**;

DataColumn CustomerMobile = new DataColumn**(**"Mobile", typeof**(**string**))**;

Customer.Columns.Add**(**CustomerMobile**)**;

//Adding Data Rows into Customer table

Customer.Rows.Add**(**101, "Anurag", "2233445566"**)**;

Customer.Rows.Add**(**202, "Manoj", "1234567890"**)**;

// Craeting Orders table

DataTable Orders = new DataTable**(**"Orders"**)**;

//Creating column and schema

DataColumn OrderId = new DataColumn**(**"ID", typeof**(**Int32**))**;

Orders.Columns.Add**(**OrderId**)**;

DataColumn CustId = new DataColumn**(**"CustomerId", typeof**(**Int32**))**;

Orders.Columns.Add**(**CustId**)**;

DataColumn OrderAmount = new DataColumn**(**"Amount", typeof**(**Int32**))**;

Orders.Columns.Add**(**OrderAmount**)**;

//Adding Data Rows into Orders table

Orders.Rows.Add**(**10001, 101, 20000**)**;

Orders.Rows.Add**(**10002, 102, 30000**)**;

//Creating DataSet object

DataSet dataSet = new DataSet**()**;

//Adding DataTables into DataSet

dataSet.Tables.Add**(**Customer**)**;

dataSet.Tables.Add**(**Orders**)**;

//Customer Table

Console.WriteLine**(**"Customer Table Data: "**)**;

//Fetching DataTable from dataset using the Index position

**foreach** **(**DataRow row in dataSet.Tables**[**0**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"ID"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

//Orders Table

Console.WriteLine**(**"Orders Table Data: "**)**;

//Fetching DataTable from the DataSet using the table name

**foreach** **(**DataRow row in dataSet.Tables**[**"Orders"**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"ID"**]** + ", " + row**[**"CustomerId"**]** + ", " + row**[**"Amount"**])**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

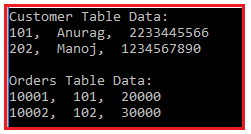
Console.ReadKey**()**;

**}**

**}**

**}**

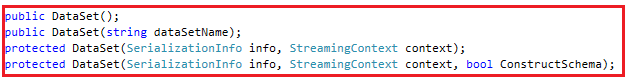
**Output:**



Now let us proceed and understand the different Constructors, Methods, and Properties of the DataSet class.

**Constructors of DataSet in C#:**

The DataSet in C# provides the following four constructors.



**Let us discuss each of these constructors.**

1. **DataSet():** It initializes a new instance of the System.Data.DataSet class..
2. **DataSet(string dataSetName):** It initializes a new instance of a System.Data.DataSet class with the given name. Here, the string parameter dataSetName specifies the name of the System.Data.DataSet.
3. **DataSet(SerializationInfo info, StreamingContext context):** It initializes a new instance of a System.Data.DataSet class that has the given serialization information and context. Here, the parameter info is the data needed to serialize or deserialize an object. The context specifies the source and destination of a given serialized stream.
4. **DataSet(SerializationInfo info, StreamingContext context, bool ConstructSchema):** It initializes a new instance of the System.Data.DataSet class.

**Properties of DataSet in C#:**

The DataSet class provides the following properties.

1. **CaseSensitive**: It is used to gets or sets a value indicating whether string comparisons within System.Data.DataTable objects are case-sensitive. It returns true if string comparisons are case-sensitive; otherwise false. The default is false.
2. **DefaultViewManager**: It is used to get a custom view of the data contained in the System.Data.DataSet to allow filtering, searching, and navigating using a custom System.Data.DataViewManager.
3. **DataSetName**: It is used to get or sets the name of the current System.Data.DataSet.
4. **EnforceConstraints**: It is used to gets or sets a value indicating whether constraint rules are followed when attempting any update operation.
5. **HasErrors**: It is used to get a value indicating whether there are errors in any of the System.Data.DataTable objects within this System.Data.DataSet.
6. **IsInitialized**: It is used to gets a value that indicates whether the System.Data.DataSet is initialized. It returns true to indicate the component has completed initialization; otherwise false.
7. **Prefix**: It is used to gets or sets an XML prefix that aliases the namespace of the System.Data.DataSet.
8. **Locale**: It is used to gets or sets the locale information used to compare strings within the table.
9. **Namespace**: It is used to Gets or sets the namespace of the System.Data.DataSet.
10. Site: It is used to gets or sets a System.ComponentModel.ISite for the System.Data.DataSet.
11. **Relations**: It is used to get the collection of relations that link tables and allow navigation from parent tables to child tables.
12. **Tables**: It is used to gets the collection of tables contained in the System.Data.DataSet.

**Methods of ADO.NET DataSet Class:**

Following are the methods provided by C# DataSet Class.

1. **BeginInit():** It Begins the initialization of a System.Data.DataSet that is used on a form or used by another component. The initialization occurs at run time.
2. **Clear():** It Clears the System.Data.DataSet of any data by removing all rows in all tables.
3. **Clone():** It Copies the structure of the System.Data.DataSet, including all System.Data.DataTable schemas, relations, and constraints. Do not copy any data.
4. **Copy():** It Copies both the structure and data for this System.Data.DataSet.
5. **CreateDataReader():** It Returns a System.Data.DataTableReader with one result set per System.Data.DataTable, in the same sequence as the tables appear in the System.Data.DataSet.Tables collection.
6. **CreateDataReader(params DataTable[] dataTables):** It returns a System.Data.DataTableReader with one result set per System.Data.DataTable. Here, the parameter dataTables specifies an array of DataTables providing the order of the result sets to be returned in the System.Data.DataTableReader
7. **EndInit():** It Ends the initialization of a System.Data.DataSet that is used on a form or used by another component. The initialization occurs at run time.
8. **GetXml():** It Returns the XML representation of the data stored in the System.Data.DataSet.
9. **GetXmlSchema():** It Returns the XML Schema for the XML representation of the data stored in the System.Data.DataSet.

Like these, there are so many methods available in the DataSet class. As we progress in this course, we will learn each and every method in detail.

The following question is being asked in the interviews.

**Which one to use between DataReader or DataSet?**

**DataSet to use:**

1. When you want to caches the data locally in your application so that you can manipulate the data.
2. When you want to interact with the data dynamically i.e. binding the data to windows form control.
3. When you want to work with disconnected architecture.

**DataReader to use:**

1. If you require some other functionality mentioned above, then you need to use DataReader which will improve the performance of your application.
2. DataReader works on connected-oriented architecture i.e. it requires an open connection to the database.

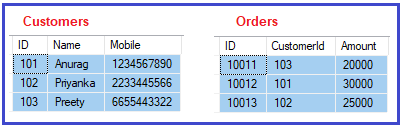
# DataSet using SQL Server

## ****ADO.NET DataSet using SQL Server****

In this article, I am going to discuss **ADO.NET DataSet using SQL Server**. Please read our previous article where we discussed the basics of [**ADO.NET DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/) and the different constructors, properties, and methods of DataSet Class. At the end of this article, you will understand how to use ADO.NET DataSet with the SQL Server database. We will see how to store one as well as multiple tables into the dataset. We will also see how to set explicitly the table names for the dataset tables.

##### ****Example to understand DataSet using SQL Server:****

We are going to use the following **Customers** and **Orders** tables to understand the ADO.NET DataSet object using the SQL Server data table.



**Please use the below SQL Script to create a database and tables and populate the Customers and Orders tables with the required test data.**

**CREATE** **DATABASE** ShoppingCartDB;

**GO**

**USE** ShoppingCartDB;

**GO**

**CREATE** **TABLE** Customers(

**ID** **INT** **PRIMARY** **KEY**,

Name **VARCHAR**(100),

Mobile **VARCHAR**(50)

)

**GO**

**INSERT** **INTO** Customers **VALUES** (101, 'Anurag', '1234567890')

**INSERT** **INTO** Customers **VALUES** (102, 'Priyanka', '2233445566')

**INSERT** **INTO** Customers **VALUES** (103, 'Preety', '6655443322')

**GO**

**CREATE** **TABLE** Orders(

**ID** **INT** **PRIMARY** **KEY**,

CustomerId **INT**,

Amount **INT**

)

**GO**

**INSERT** **INTO** Orders **VALUES** (10011, 103, 20000)

**INSERT** **INTO** Orders **VALUES** (10012, 101, 30000)

**INSERT** **INTO** Orders **VALUES** (10013, 102, 25000)

**GO**

##### ****Example:****

Our business requirement is to fetch all the data from the Customers table and then need to display it on the console. The following example exactly does the same using DataSet. In the below example, we created an instance of the DataSet and then fill the dataset using the Fill method data adapter object.

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=ShoppingCartDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlDataAdapter dataAdapter = new SqlDataAdapter**(**"select \* from customers", connection**)**;

//Creating DataSet Object

DataSet dataSet = new DataSet**()**;

//Filling the DataSet

dataAdapter.Fill**(**dataSet**)**;

//Iterating through the DataSet

**foreach** **(**DataRow row in dataSet.Tables**[**0**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Mobile"**])**;

**}**

**}**

Console.ReadKey**()**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

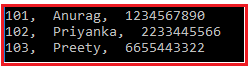
Console.ReadKey**()**;

**}**

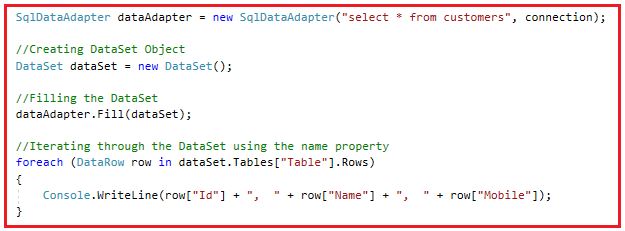
**}**

**}**

**Output:**



By default, the dataset assigns a name to the table as Table, Table1, Table2. So, the above example can be rewritten as shown below and it should give the same output as the previous example. As you can see, here, we are fetching the table using the name (Table).



##### ****DataSet with Multiple Database Tables using SQL Server:****

It is also possible that your SQL Query may return multiple tables. Let us understand this with an example. Now our business requirement is to fetch the Customers as well as Orders table data and needs to display on the Console. Here, you can access the first table from the dataset using an integral index 0 or string Table name. On the other hand, you can access the second table using the integral index 1 or the string name Table1.

###### **Using Integral Index Position:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=ShoppingCartDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

//Sql Command return data from customers and orders table

SqlDataAdapter dataAdapter = new SqlDataAdapter**(**"select \* from customers; select \* from orders", connection**)**;

DataSet dataSet = new DataSet**()**;

dataAdapter.Fill**(**dataSet**)**;

// First Table

Console.WriteLine**(**"Table 1 Data"**)**;

**foreach** **(**DataRow row in dataSet.Tables**[**0**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

// Second Table

Console.WriteLine**(**"Table 2 Data"**)**;

**foreach** **(**DataRow row in dataSet.Tables**[**1**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"CustomerId"**]** + ", " + row**[**"Amount"**])**;

**}**

**}**

Console.ReadKey**()**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

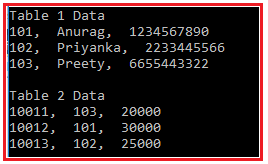
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



###### **Using Table Name:**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=ShoppingCartDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

//Sql Command return data from customers and orders table

SqlDataAdapter dataAdapter = new SqlDataAdapter**(**"select \* from customers; select \* from orders", connection**)**;

DataSet dataSet = new DataSet**()**;

dataAdapter.Fill**(**dataSet**)**;

// First Table

Console.WriteLine**(**"Table 1 Data"**)**;

**foreach** **(**DataRow row in dataSet.Tables**[**"Table"**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

// Second Table

Console.WriteLine**(**"Table 2 Data"**)**;

**foreach** **(**DataRow row in dataSet.Tables**[**"Table1"**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"CustomerId"**]** + ", " + row**[**"Amount"**])**;

**}**

**}**

Console.ReadKey**()**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

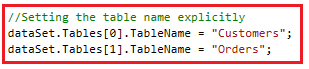
**}**

It will also give the same output as the previous example.

##### ****Setting the table name explicitly in DataSet:****

If your dataset going to contain multiple tables data, then it is very difficult for you to identify using the integral index position or using the default name. In such a scenario, it is always recommended to provide an explicit name for the table.

Let us understand this with an example. Now, we need to set the first table as Customers and the second table as Orders and then we will see how to use these custom table names to fetch the actual table data. You can set the table name using the TableName property as shown below.



The following is the complete example that uses the tableName property of the dataset object to set and get the table name.

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *AdoNetConsoleApplication*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=ShoppingCartDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

//Sql Command return data from customers and orders table

SqlDataAdapter dataAdapter = new SqlDataAdapter**(**"select \* from customers; select \* from orders", connection**)**;

DataSet dataSet = new DataSet**()**;

dataAdapter.Fill**(**dataSet**)**;

//Setting the table name explicitly

dataSet.Tables**[**0**]**.TableName = "Customers";

dataSet.Tables**[**1**]**.TableName = "Orders";

Console.WriteLine**(**"Customers Table Data"**)**;

//Fetching the table using the custom table name

**foreach** **(**DataRow row in dataSet.Tables**[**"Customers"**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"Name"**]** + ", " + row**[**"Mobile"**])**;

**}**

Console.WriteLine**()**;

// Second Table

Console.WriteLine**(**"Orders Table Data"**)**;

//Fetching the table using the custom table name

**foreach** **(**DataRow row in dataSet.Tables**[**"Orders"**]**.Rows**)**

**{**

Console.WriteLine**(**row**[**"Id"**]** + ", " + row**[**"CustomerId"**]** + ", " + row**[**"Amount"**])**;

**}**

**}**

Console.ReadKey**()**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

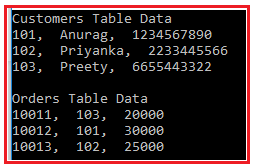
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

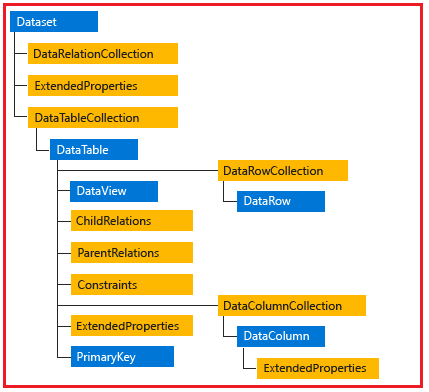


## ****The architecture of DataSet in .NET****

In this article, I am going to discuss the **Architecture of DataSet in .NET** Framework. Please read our previous article where we discussed[**how to use DataSet with the SQL Server**](https://dotnettutorials.net/lesson/dataset-using-sql-server/)database. I hope at the end of this article, you will understand the Architecture of DataSet in .NET.

##### ****What is a DataSet in .NET?****

The DataSet object is central to supporting disconnected, distributed data scenarios with ADO.NET. The DataSet is a memory-resident representation of data that provides a consistent relational programming model regardless of the data source. It can be used with multiple and differing data sources, with XML data, or to manage data local to the application. The DataSet represents a complete set of data, including related tables, constraints, and relationships among the tables. The following image shows the DataSet object model.



The methods and objects in a DataSet are consistent with those in the relational database model. The DataSet can also persist and reload its contents as XML, and its schema as XML schema definition language (XSD) schema. For more information, see Using XML in a DataSet.

##### ****DataTableCollection****

An [**ADO.NET DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/) contains a collection of zero or more tables represented by [**DataTable**](https://dotnettutorials.net/lesson/ado-net-datatable/)objects. The DataTableCollection contains all the DataTable objects in a DataSet.

A [**DataTable**](https://dotnettutorials.net/lesson/ado-net-datatable/)is defined in the **System.Data** namespace and represents a single table of memory-resident data. It contains a collection of columns represented by a DataColumnCollection and constraints represented by a ConstraintCollection, which together define the schema of the table. A DataTable also contains a collection of rows represented by the DataRowCollection, which contains the data in the table. Along with its current state, a DataRow retains both its current and original versions to identify changes to the values stored in the row.

##### ****DataView Class****

A DataView enables you to create different views of the data stored in a [**DataTable**](https://dotnettutorials.net/lesson/ado-net-datatable/), a capability that is often used in data-binding applications. Using a DataView, you can expose the data in a table with different sort orders, and you can filter the data by row state or based on a filter expression. For more information, see DataViews.

##### ****DataRelationCollection****

A [**DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/)contains relationships in its DataRelationCollection object. A relationship, represented by the DataRelation object, associates rows in one DataTable with rows in another DataTable. A relationship is analogous to a join path that might exist between primary and foreign key columns in a relational database. A DataRelation identifies matching columns in two tables of a DataSet.

Relationships enable navigation from one table to another in a DataSet. The essential elements of a DataRelation are the name of the relationship, the name of the tables being related, and the related columns in each table. Relationships can be built with more than one column per table by specifying an array of DataColumn objects as the key columns. When you add a relationship to the DataRelationCollection, you can optionally add a UniqueKeyConstraint and a ForeignKeyConstraint to enforce integrity constraints when changes are made to related column values.

##### ****XML****

You can fill a DataSet from an XML stream or document. You can use the XML stream or document to supply to the DataSet either data, schema information, or both. The information supplied from the XML stream or document can be combined with existing data or schema information already present in the DataSet.

##### ****ExtendedProperties****

The [**DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/), DataTable, and DataColumn all have an ExtendedProperties property. ExtendedProperties is a PropertyCollection where you can place custom information, such as the SELECT statement that was used to generate the result set or the time when the data was generated. The ExtendedProperties collection is persisted with the schema information for the DataSet.

##### ****LINQ to DataSet****

[**LINQ**](https://dotnettutorials.net/lesson/introduction-to-linq/)to DataSet provides language-integrated querying capabilities for disconnected data stored in a [**DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/). [**LINQ**](https://dotnettutorials.net/lesson/introduction-to-linq/)to DataSet uses standard LINQ syntax and provides compile-time syntax checking, static typing, and IntelliSense support when you are using the Visual Studio IDE.

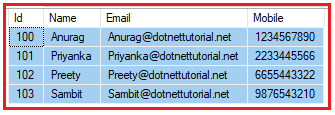
**ADO.NET Using Stored Procedure**

##### ****What is a Stored Procedure in SQL?****

A Stored Procedure in SQL is a database object which contains pre-compiled SQL Statements. In simple words, we can also say that the Stored Procedures are a block of code that is designed to perform a specific task whenever it is called. Please [**click here**](https://dotnettutorials.net/lesson/sql-server-stored-procedure/)to learn [**SQL Server Stored Procedure**](https://dotnettutorials.net/sql-server-stored-procedure/) in detail.

##### ****Example to understand ADO.NET using Stored Procedure:****

We are going to use the following student table in this demo to understand the concept of [**ADO.NET**](https://dotnettutorials.net/lesson/what-is-ado-net/) using Stored Procedure.



**Please use the below SQL Script to create and populate the database StudentDB and table Student with the required sample data.**

**CREATE** **DATABASE** StudentDB;

**GO**

**USE** StudentDB;

**GO**

**CREATE** **TABLE** Student(

[Id] [int] **IDENTITY**(100,1) **PRIMARY** **KEY**,

[Name] [varchar](100) **NULL**,

[Email] [varchar](50) **NULL**,

[Mobile] [varchar](50) **NULL**,

)

**GO**

**INSERT** **INTO** Student **VALUES** ('Anurag','Anurag@dotnettutorial.net','1234567890')

**INSERT** **INTO** Student **VALUES** ('Priyanka','Priyanka@dotnettutorial.net','2233445566')

**INSERT** **INTO** Student **VALUES** ('Preety','Preety@dotnettutorial.net','6655443322')

**INSERT** **INTO** Student **VALUES** ('Sambit','Sambit@dotnettutorial.net','9876543210')

##### ****Stored Procedure without a parameter:****

So, the first thing is always first. Let us create a stored procedure that will not take any input parameter but will return all the records from the student table. Please use the below SQL Script to create the stored procedure.

**CREATE** **PROCEDURE** spGetStudents

**AS**

**BEGIN**

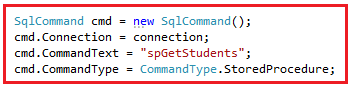
**SELECT** Id, Name, Email, Mobile

**FROM** Student

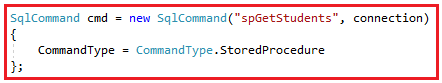
**END**

##### ****How to call a stored procedure using C# ADO.NET.****

Let us see the step-by-step procedure to call the above-stored procedure. Please have a look at the below image. So, what you need to do is, first create an instance of command object and then specify the commandTest property value as the stored procedure name and the most important point is you need to specify the command type as Stored Procedure.



You can also use the other overloaded constructor of the Command class as shown in the below image. As you can see, here we are passing the stored procedure name and the connection object to the constructor of the command object and then specify the command type as a Stored procedure.



###### **The complete example is given below.**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *ADOUsingStoredProcedure*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

SqlCommand cmd = new SqlCommand**(**"spGetStudents", connection**)**

**{**

CommandType = CommandType.StoredProcedure

**}**;

connection.Open**()**;

SqlDataReader sdr = cmd.ExecuteReader**()**;

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"Id"**]** + ", "+sdr**[**"Name"**]** + ", " + sdr**[**"Email"**]** + ", " + sdr**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

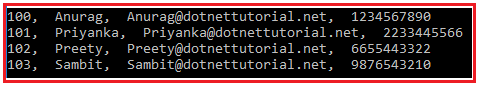
Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**



##### ****Stored procedure with Input Parameter:****

Now, we will see how to call a stored procedure with an input parameter. So, please use the below SQL Script to create the stored procedure which will return the student details by id. Here, student id is the input parameter and that parameter value we need to pass while calling this stored procedure.

**CREATE** **PROCEDURE** spGetStudentById

(

@Id **INT**

)

**AS**

**BEGIN**

**SELECT** Id, Name, Email, Mobile

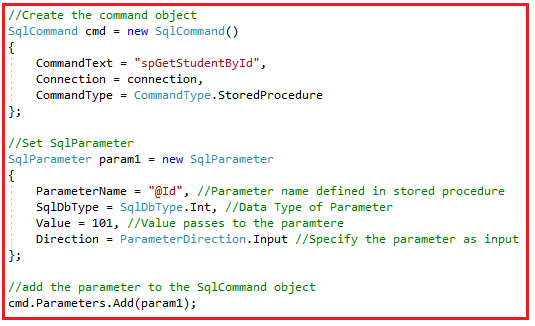
**FROM** Student

**WHERE** Id = @Id

**END**

##### ****How to call a stored procedure with an input parameter in C# ADO.NET?****

In order to understand how to call a stored procedure with an input parameter, please have a look at the following image. We already discussed the command object. The point that you need to focus on here is the SqlParameter object. As you can see here we are creating an instance of the SqlParameter object and then setting the parameter name, the data type, value, and the direction of the parameter.



###### **The complete example is given below.**

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *ADOUsingStoredProcedure*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

//Create the command object

SqlCommand cmd = new SqlCommand**()**

**{**

CommandText = "spGetStudentById",

Connection = connection,

CommandType = CommandType.StoredProcedure

**}**;

//Set SqlParameter

SqlParameter param1 = new SqlParameter

**{**

ParameterName = "@Id", //Parameter name defined in stored procedure

SqlDbType = SqlDbType.Int, //Data Type of Parameter

Value = 101, //Value passes to the paramtere

Direction = ParameterDirection.Input //Specify the parameter as input

**}**;

//add the parameter to the SqlCommand object

cmd.Parameters.Add**(**param1**)**;

connection.Open**()**;

SqlDataReader sdr = cmd.ExecuteReader**()**;

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"Id"**]** + ", "+sdr**[**"Name"**]** + ", " + sdr**[**"Email"**]** + ", " + sdr**[**"Mobile"**])**;

**}**

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Stored procedure with Input Parameter in ADO.NET

##### ****Stored Procedure with both Input and Output Parameter:****

In our previous example, we understand how to call a stored procedure with an input parameter. Now, let us see how to call a stored procedure with both input and output parameters. So, please use the below SQL Script to create the stored procedure with both input and output parameters.

**CREATE** **PROCEDURE** spCreateStudent

(

@Name **VARCHAR**(100),

@Email **VARCHAR**(50),

@Mobile **VARCHAR**(50),

@Id int Out

)

**AS**

**BEGIN**

**INSERT** **INTO** Student **VALUES** (@Name,@Email,@Mobile)

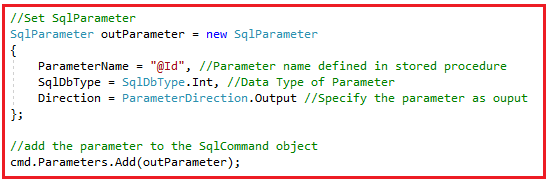
**SELECT** @Id = SCOPE\_IDENTITY()

**END**

As you can see the above-stored procedure, takes four parameters (3 input + 1 output). The above-stored procedure is very simple, takes the Student Name, Email and Mobile and inserts it into the Student table. As we created the student table with Id as an Identity column, so, we don’t require to pass the id value in the insert statement. But what we want here is we need to return the newly created student id and this is where the output parameter comes into the picture. Here, we set the output parameter value with the newly generated student id.

##### ****How to call a stored procedure with both input and output parameters in C#?****

In order to understand how to call a stored procedure with both input and output parameters, please have a look at the following image. As you can see, while defining the Output Parameter you need to specify the parameter direction as Output and you don’t require to set the value property.



**Note:** By default the parameter direction is Input.

##### ****The complete code is given below.****

**using** *System;*

**using** *System.Data;*

**using** *System.Data.SqlClient;*

**namespace** *ADOUsingStoredProcedure*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

string ConnectionString = "data source=.; database=StudentDB; integrated security=SSPI";

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

//Create the command object

SqlCommand cmd = new SqlCommand**()**

**{**

CommandText = "spCreateStudent",

Connection = connection,

CommandType = CommandType.StoredProcedure

**}**;

//Set SqlParameter

SqlParameter param1 = new SqlParameter

**{**

ParameterName = "@Name", //Parameter name defined in stored procedure

SqlDbType = SqlDbType.NVarChar, //Data Type of Parameter

Value = "Test",

Direction = ParameterDirection.Input //Specify the parameter as input

**}**;

//add the parameter to the SqlCommand object

cmd.Parameters.Add**(**param1**)**;

//Another approach to add input parameter

cmd.Parameters.AddWithValue**(**"@Email", "Test@dotnettutorial.net"**)**;

cmd.Parameters.AddWithValue**(**"@Mobile", "1234567890"**)**;

//Set SqlParameter

SqlParameter outParameter = new SqlParameter

**{**

ParameterName = "@Id", //Parameter name defined in stored procedure

SqlDbType = SqlDbType.Int, //Data Type of Parameter

Direction = ParameterDirection.Output //Specify the parameter as ouput

**}**;

//add the parameter to the SqlCommand object

cmd.Parameters.Add**(**outParameter**)**;

connection.Open**()**;

cmd.ExecuteNonQuery**()**;

Console.WriteLine**(**"Newely Generated Student ID : " + outParameter.Value.ToString**())**;

**}**

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong.\n" + e**)**;

**}**

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:**

Stored Procedure with both Input and Output Parameter in ADO.NET

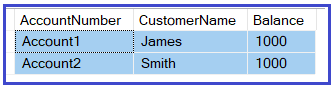
# Transactions in ADO.NET

## ****Transactions in ADO.NET with Examples****

##### ****Why we need Transactions?****

The most important thing in today’s world is data and the most challenging job is to keep the data consistent. The Database systems stores the data and ADO.NET is one of the data access technology to access the data stored in the database.

Let us first understand what do you mean by data consistency and then we will understand why we need transactions. For this please have a look at the following diagram. Here, you can see, we have an Accounts Table with two Account Numbers.



Now, the business requirement is to transfer 500 from Account1 to Account2. For this, we need to write two update statements as shown below. The first update statement deducts 500 from Account1 and the 2nd update statement Adds 500 to Account2.

**UPDATE Accounts SET Balance = Balance – 500 WHERE AccountNumber = ‘Account1’;**  
**UPDATE Accounts SET Balance = Balance + 500 WHERE AccountNumber = ‘Account2’;**

Our intention is data consistency. Once the update statements are executed the data should be in a consistent state. Now let us understand the following cases.

###### **Case1:**

The First update statement executed successfully but the second update statement Failed. In that case, 500 is deducted from Account1 but that amount is not added to Account2 which results in data inconsistency.

###### **Case2:**

The First update statement Failed but the second update statement executed successfully. In that case, 500 is not deducted from Account1 but 500 is added to Account2 which results in data inconsistency.

###### **Case3:**

When both the update statements are Failed, then the data is in a consistent state.

###### **Case4:**

When both the update statements are Successful, then the data is also in a consistent state. That is 500 is deducted from Account1 and 500 is added to Account2.

From the above discussed four cases, we don’t have any issues in Case3 and Case4. At the same time, we also can’t give the guarantee that every time both the update statements are Failed and succeed. That means we need to do something special to handle Case1 and Case2 so that the data is to be in a consistent state and for this, we need to use transactions.

So, in order to keep the data in a consistent state in the database while accessing the data using ADO.NET, we need to use transactions.

##### ****What is a Transaction?****

A Transaction is a set of operations (multiple DML Operations) that ensures either all of the database operations succeed or all of them failed to ensure data consistency. This means the job is never half done, either all of it is done or nothing is done.

##### ****ADO.NET Transactions Supports:****

The ADO.NET supports both single database transactions as well as distributed transactions. The single database transaction is implemented using the .NET managed providers for Transaction and Connection classes which are basically belong to System.Data namespace.

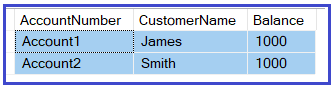
##### ****How to use Transaction in ADO.NET?****

There are many different ways that we can use to implement Transaction using ADO.NET and C#. They are as follows:

1. Single Database Transaction using BeginTransaction
2. Distributed Transaction using TransactionScope which belongs to System.Transactions namespace
3. Distributed Transaction using ServicedComponent

##### ****Single Database Transaction using BeginTransaction****

Let us understand how to implement a Single Database Transaction using BeginTransaction. We are going to implement the same money transfer example. Here, we are executing two update statements using ADO.NET Transaction. For this, we are going to use the following Accounts table in this demo.



##### ****Create Accounts Table using SQL Script****

Please use the below SQL Statements to create and populate the Accounts table with the required data.

**CREATE** **TABLE** Accounts

(

AccountNumber **VARCHAR**(60) **PRIMARY** **KEY**,

CustomerName **VARCHAR**(60),

Balance int

);

**GO**

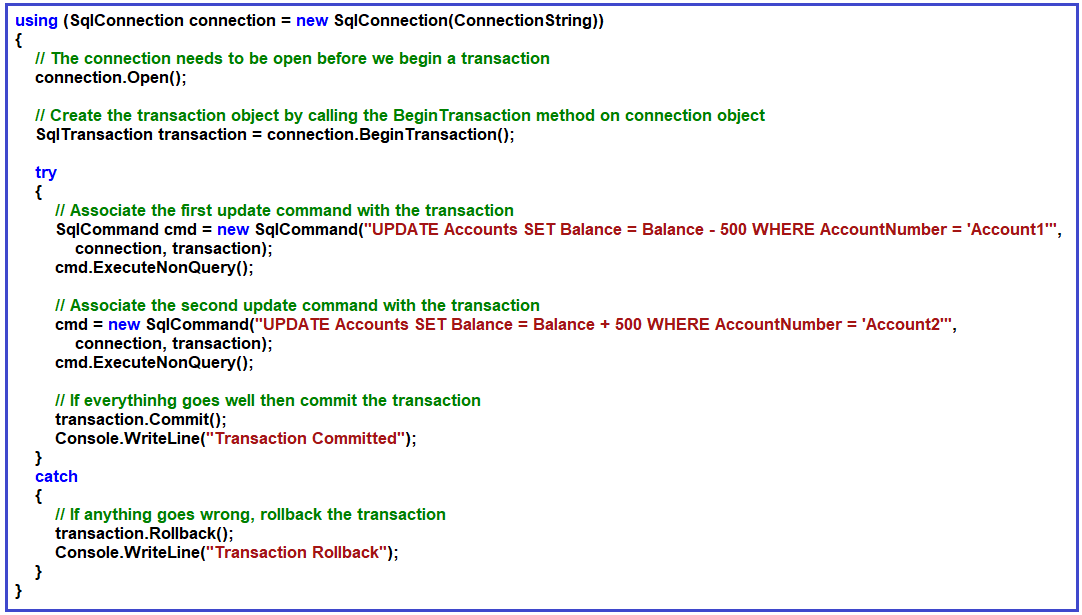
**INSERT** **INTO** Accounts **VALUES**('Account1', 'James', 1000);

**INSERT** **INTO** Accounts **VALUES**('Account2', 'Smith', 1000);

**GO**

##### ****How to Implement a Single Database Transaction using ADO.NET?****

In order to understand how to implement transactions, please have a look at the below image.



**Step1:** First you need to create and open the connection object. The following two statements do the same.  
**SqlConnection connection = new SqlConnection(ConnectionString)**  
**connection.Open();**

**Step2:** Then you need to create the SqlTransaction object and to do so, you need to call the BeginTransaction method on the connection object. The following piece of code does the same.  
**SqlTransaction transaction = connection.BeginTransaction();**

**Step3:** If everything goes well then commit the transaction. To do so call the Commit method on the transaction object as shown below.  
**transaction.Commit();**

**Step4:** If anything goes wrong then rollback the transaction. To do so call the Rollback method on the transaction object as shown below.  
**transaction.Rollback();**

##### ****Create a new console application****

First, open visual studio and then create a new console application with the name ADOTransactionsDemo. Open **Program.cs** class file and then copy and paste the following code in it. As you can see here, we are using ADO.NET Transaction and executing two update statements.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *ADOTransactionsDemo*

**{**

**class** Program

**{**

**public** **static** string ConnectionString = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True";

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

Console.WriteLine**(**"Before Transaction"**)**;

GetAccountsData**()**;

MoneyTransfer**()**;

Console.WriteLine**(**"After Transaction"**)**;

GetAccountsData**()**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong" + e.Message**)**;

**}**

Console.ReadKey**()**;

**}**

**private** **static** **void** MoneyTransfer**()**

**{**

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

// The connection needs to be open before we begin a transaction

connection.Open**()**;

// Create the transaction object by calling the BeginTransaction method on connection object

SqlTransaction transaction = connection.BeginTransaction**()**;

**try**

**{**

// Associate the first update command with the transaction

SqlCommand cmd = new SqlCommand**(**"UPDATE Accounts SET Balance = Balance - 500 WHERE AccountNumber = 'Account1'", connection, transaction**)**;

cmd.ExecuteNonQuery**()**;

// Associate the second update command with the transaction

cmd = new SqlCommand**(**"UPDATE Accounts SET Balance = Balance + 500 WHERE AccountNumber = 'Account2'", connection, transaction**)**;

cmd.ExecuteNonQuery**()**;

// If everythinhg goes well then commit the transaction

transaction.Commit**()**;

Console.WriteLine**(**"Transaction Committed"**)**;

**}**

**catch**

**{**

// If anything goes wrong, rollback the transaction

transaction.Rollback**()**;

Console.WriteLine**(**"Transaction Rollback"**)**;

**}**

**}**

**}**

**private** **static** **void** GetAccountsData**()**

**{**

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

connection.Open**()**;

SqlCommand cmd = new SqlCommand**(**"Select \* from Accounts", connection**)**;

SqlDataReader sdr = cmd.ExecuteReader**()**;

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"AccountNumber"**]** + ", " + sdr**[**"CustomerName"**]** + ", " + sdr**[**"Balance"**])**;

**}**

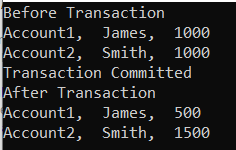
**}**

**}**

**}**

**}**

**Output:** As you can see in the below output the data is in a consistent state i.e. updated in both the Account Number.



##### ****Verifying Data Consistency:****

Let us modify the Program as shown below. In the following code, we deliberately introduce a change that would crash the application at run time after executing the first update statement. Here, in the second update statement in rename the table name as MyAccounts which does not exist in the database.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *ADOTransactionsDemo*

**{**

**class** Program

**{**

**public** **static** string ConnectionString = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True";

**static** **void** Main**(**string**[]** args**)**

**{**

**try**

**{**

Console.WriteLine**(**"Before Transaction"**)**;

GetAccountsData**()**;

MoneyTransfer**()**;

Console.WriteLine**(**"After Transaction"**)**;

GetAccountsData**()**;

**}**

**catch** **(**Exception e**)**

**{**

Console.WriteLine**(**"OOPs, something went wrong" + e.Message**)**;

**}**

Console.ReadKey**()**;

**}**

**private** **static** **void** MoneyTransfer**()**

**{**

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

// The connection needs to be open before we begin a transaction

connection.Open**()**;

// Create the transaction object by calling the BeginTransaction method on connection object

SqlTransaction transaction = connection.BeginTransaction**()**;

**try**

**{**

// Associate the first update command with the transaction

SqlCommand cmd = new SqlCommand**(**"UPDATE Accounts SET Balance = Balance - 500 WHERE AccountNumber = 'Account1'",

connection, transaction**)**;

cmd.ExecuteNonQuery**()**;

// Associate the second update command with the transaction

cmd = new SqlCommand**(**"UPDATE MyAccounts SET Balance = Balance + 500 WHERE AccountNumber = 'Account2'",

connection, transaction**)**;

cmd.ExecuteNonQuery**()**;

// If everythinhg goes well then commit the transaction

transaction.Commit**()**;

Console.WriteLine**(**"Transaction Committed"**)**;

**}**

**catch**

**{**

// If anything goes wrong, rollback the transaction

transaction.Rollback**()**;

Console.WriteLine**(**"Transaction Rollback"**)**;

**}**

**}**

**}**

**private** **static** **void** GetAccountsData**()**

**{**

**using** **(**SqlConnection connection = new SqlConnection**(**ConnectionString**))**

**{**

connection.Open**()**;

SqlCommand cmd = new SqlCommand**(**"Select \* from Accounts", connection**)**;

SqlDataReader sdr = cmd.ExecuteReader**()**;

**while** **(**sdr.Read**())**

**{**

Console.WriteLine**(**sdr**[**"AccountNumber"**]** + ", " + sdr**[**"CustomerName"**]** + ", " + sdr**[**"Balance"**])**;

**}**

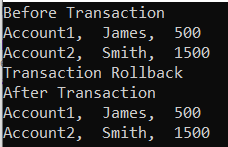
**}**

**}**

**}**

**}**

**Output:** As you can see the transaction is rollback and the data which is updated by the first update statement is rollback to its previous state and hence transaction maintains data consistency.



##### ****Setting Isolation Level in ADO.NET Transaction:****

It is also possible in ADO.NET to set the Transaction IsolationLevel while creating the transaction object from the connection object as: **SqlTransaction transaction = connection.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);**

Please read our [**different transaction isolation levels and their needs**](https://dotnettutorials.net/lesson/sql-server-concurrent-transactions/) in real-time applications.

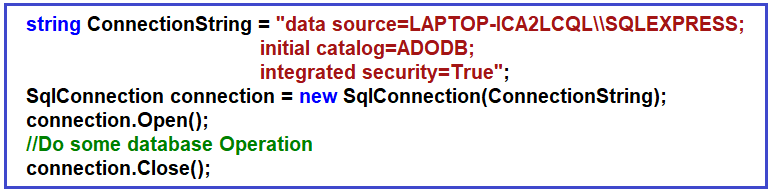
**ADO.NET Connection Pooling**

**ADO.NET Connection Pooling**

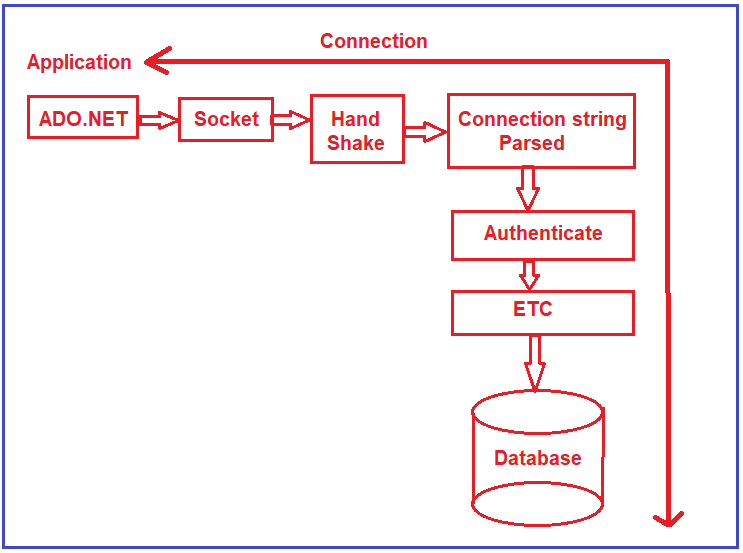
In this article, I am going to discuss **ADO.NET Connection Pooling** with Examples. Please read our previous article where we discussed [**Transactions in ADO.NET**](https://dotnettutorials.net/lesson/transactions-in-ado-net/).

**Connection Pooling in ADO.NET:**

When we use ADO.NET in our C# applications, what we do is, first we will create the connection object, then open the connection, then perform some database operations and finally close the connection as shown in the below image.

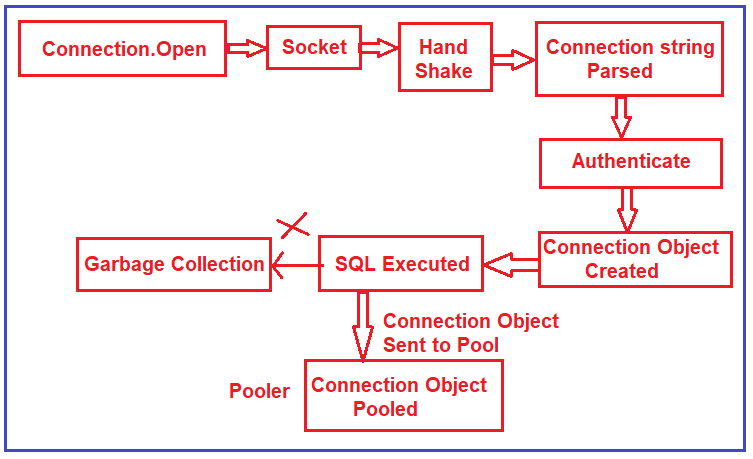


Now the creation of a connection object and opening the connection object is quite intensive. In other words, when we say, open the connection, opens the socket, some kind of handshaking is happening, the connection string is parsed to check whether the connection string format is proper or not, the Authentication mechanism is executed, and lots of other series of steps have happened internally before the connection object gets connected to the underlying database. For a better understanding, please have a look at the below image.



Once the connection object is open, you can perform the database CRUD operation and once the DB Operations performed, you can close the connection. This creation of a connection object is quite intensive and you would like to avoid the above things again and again when you need to create the connection object.

So, in other words, we would like to say that, open the connection object, do all the series of steps (socket, handshake, connection string parsed, authenticate, etc.), do the operations, and close the connection object. But when we say close, don’t make this connection object and go for garbage collector rather than cache it in a pool. So that, the next time when someone says open connection, then get the connection object from the pool rather than going and executing the series of intensive steps. For a better understanding, please have a look at the following image.



So, connection pooling means, once the connection object is open, rather than going and recreating the connection object again and again, what ADO.NET does, it takes the connection object and puts it into a thing called a pooler. In the pooler, the object will be cached, and later if somebody says connection.open then rather than executing the series of steps, it takes the connection object from the pool and start executing.

**Connection Pooling Example in ADO.NET:**

Let us first create a console application with the name ConnectionPooling. The most important point that you need to remember is by default connection pooling is enabled in ADO.NET. Please have a look at the below example. Here, we are using a big for loop and in each iteration, we are creating the connection object, opening the connection, doing some operation (intentionally using thread sleep to check the connection pooling), and closing the connection object.

**Example: Using Connection Pooling**

By default, the connection pooling is enabled in ADO.NET. If you want then you can Pooling=true; in the connection string which will enable the connection pooling in ADO.NET. In the following example, we set the pooling value true and then creating 1000 connection objects. Please execute the below code and see the time taken by ADO.NET when connection pooling is enabled.

**using** *System;*

**using** *System.Data.SqlClient;*

**using** *System.Diagnostics;*

**namespace** *ConnectionPooling*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var stopwatch = new Stopwatch**()**;

string ConnectionString = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=true;";

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 1000; i++**)**

**{**

SqlConnection connection = new SqlConnection**(**ConnectionString**)**;

connection.Open**()**;

connection.Close**()**;

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**$"Pooling=true, Time : {stopwatch.ElapsedMilliseconds} ms"**)**;

Console.ReadKey**()**;

**}**

**}**

**}**

**Output:** **Pooling=true, Time : 163 ms**

Even though the loop is going to be executed 1000 times, we should not see too many connection objects get created rather the connection objects are going to be fetched from the connection pool. It will use the same connection object from the pool again and again. And hence you can see, it simply taking 163 ms. If you remove the Pooling=true; from the connection string, then also it is going to fetch the connection object from the pool as by default connection pooling is enabled in ADO.NET.

**Example: Without Connection Pooling**

If you don’t want to use connection pooling, then you need to set Pooling=false; in the connection string to disabled the connection pooling in ADO.NET. The following is the same example as the previous one, except here we are setting the pooling value to false. Please execute the below code and see the time taken by ADO.NET when connection pooling is disabled.

**using** *System;*

**using** *System.Data.SqlClient;*

**using** *System.Diagnostics;*

**namespace** *ConnectionPooling*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

var stopwatch = new Stopwatch**()**;

string ConnectionString = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=false;";

stopwatch.Start**()**;

**for** **(int** i = 0; i **<** 1000; i++**)**

**{**

SqlConnection connection = new SqlConnection**(**ConnectionString**)**;

connection.Open**()**;

connection.Close**()**;

**}**

stopwatch.Stop**()**;

Console.WriteLine**(**$"Pooling=false, Time : {stopwatch.ElapsedMilliseconds} ms"**)**;

Console.ReadKey**()**;

**}**

**}**

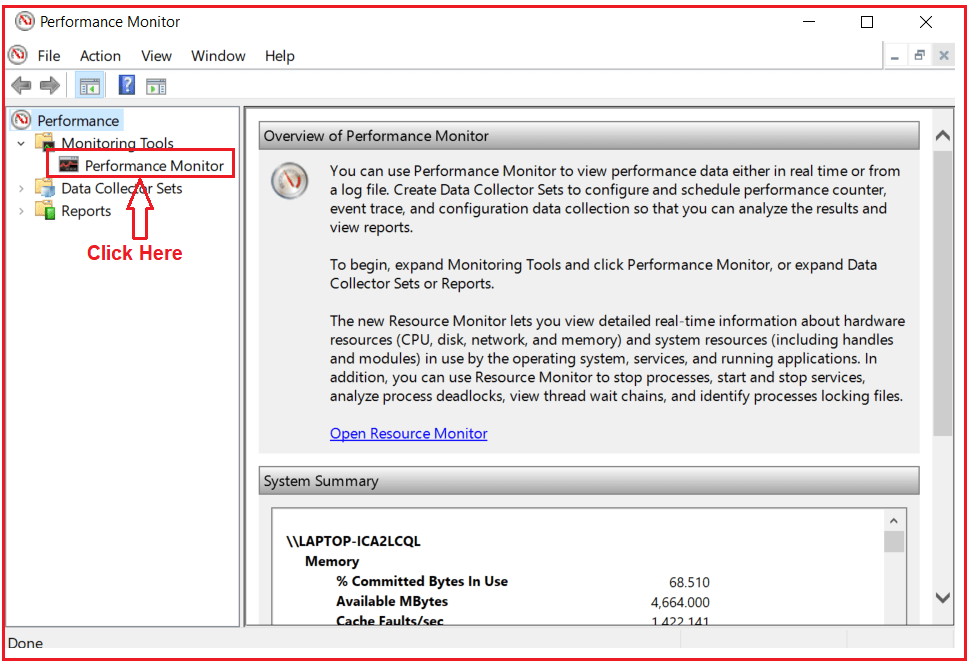
**}**

**Output: Pooling=false, Time : 3976 ms**

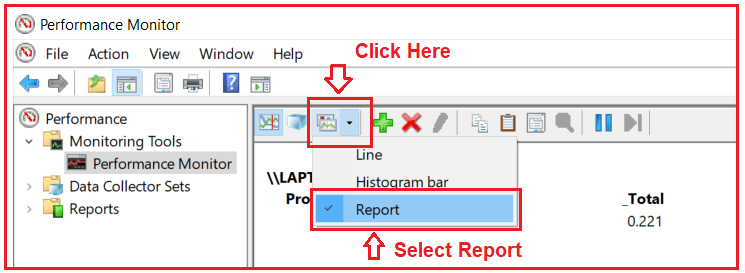
As you can see, it is taking 3976 ms as compared to 163 ms when connection pooling is disabled.

**How to Verify Connection Pooling is Used in ADO.NET?**

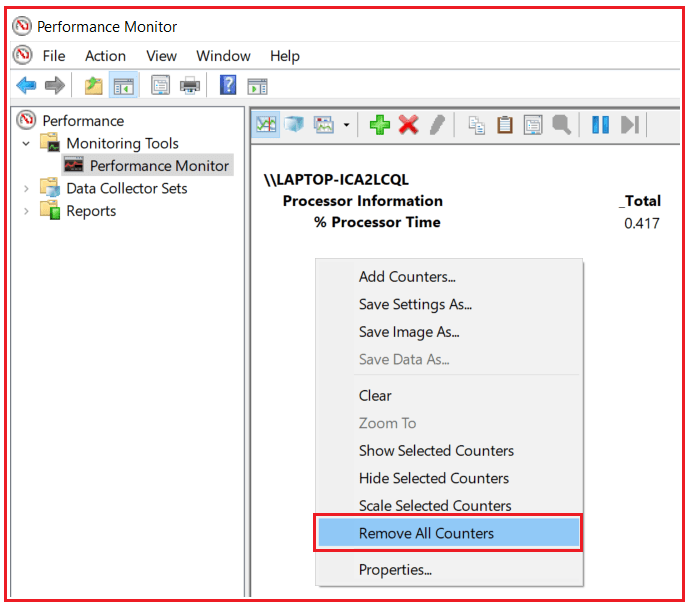
In order to check whether the connection objects are fetching from the connection pooled or not, we are going to use a tool called **Performance Monitor** (perfmon) which is available in Windows machines. Open the Performance Monitor (perfmon) tool and then click on the Performance Monitor button as shown in the below image.



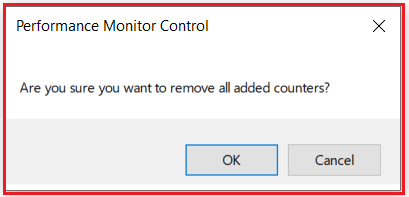
Once you click on the Performance Monitor button, it will open the below window. Here, click on the Change Graph Type menu, and from the drop-down list select the graph option as shown in the below image.



Once you click on the Report option, it will open the below window. Here, right-click on the blank surface and click on the Remove All Counters from the context menu as shown in the below image.



Once you click on the Remove All Counters option, one popup will be opened, simply click on the OK button as shown below.



**Now modify the code as shown below.**

**using** *System;*

**using** *System.Data.SqlClient;*

**using** *System.Threading;*

**namespace** *ConnectionPooling*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string ConnectionString = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=true;";

**for** **(int** i = 0; i **<** 1000; i++**)**

**{**

SqlConnection connection = new SqlConnection**(**ConnectionString**)**;

connection.Open**()**;

Thread.Sleep**(**100**)**;

connection.Close**()**;

**}**

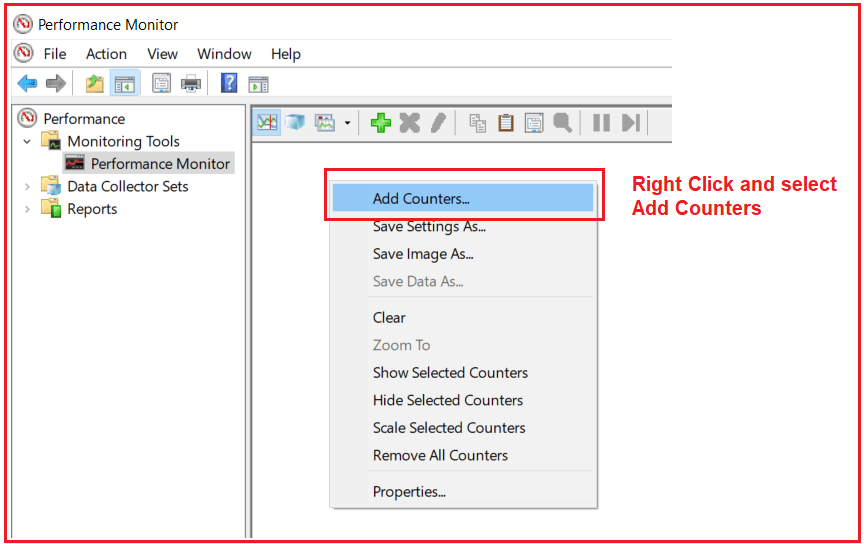
Console.ReadKey**()**;

**}**

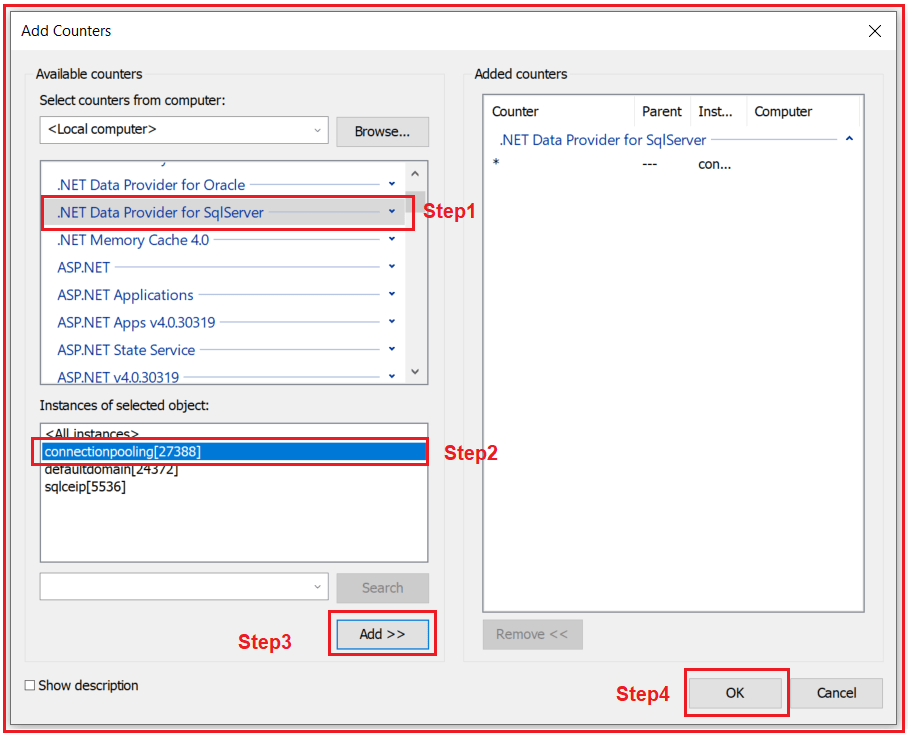
**}**

**}**

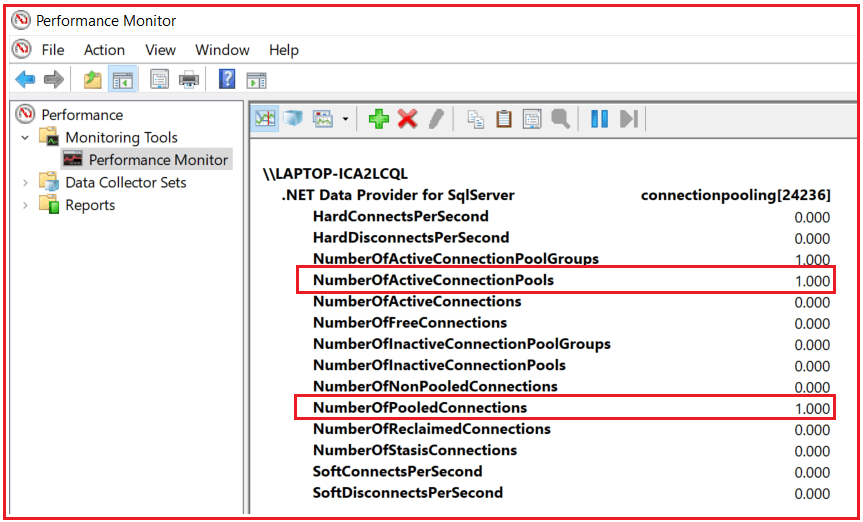
With the above changes in place, now, run the application. Once your application starts executing, go to the performance monitor tool and add a performance counter. In order to Add Counter, simply right-click and then select the Add Counters option from the context menu as shown in the below image.



Once you click on the Add Counter option, it will open the below Add Counters window First, select the **.NET Data Provider for SQL Server** as we are using SQL Server database, and then select the console application which should be in running mode. Then click on the **Add** button which adds the performance monitor and finally click on the **OK** button as shown in the below image.

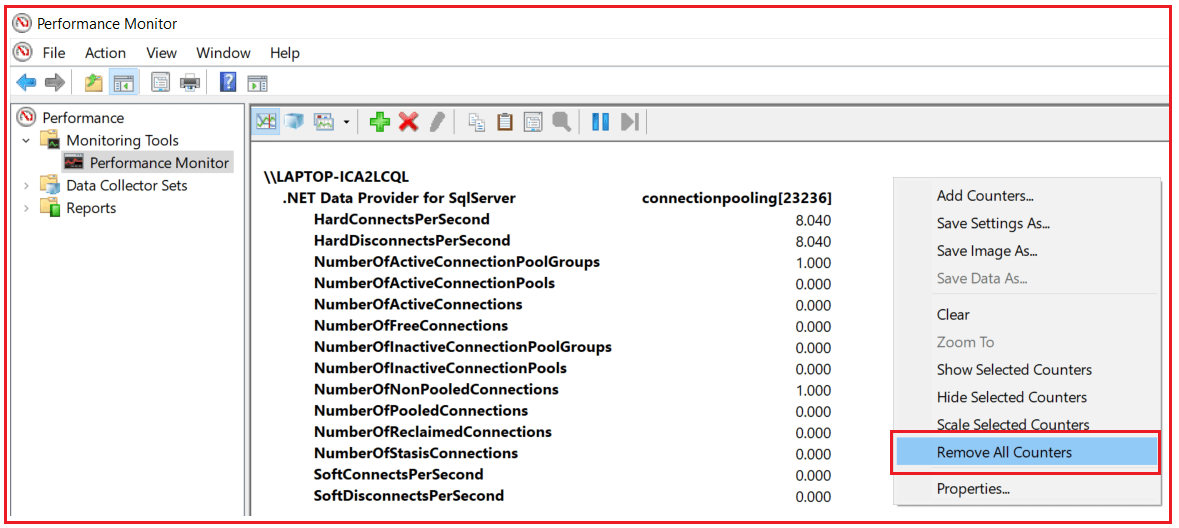


Once you click on the OK button, it will open the following report. As you can see in the below image, the number of active connections is 1. This is because as the for loop is running at any given moment of time, there will be only one connection object is opened. Further, if you notice, the number of pooled connections is also 1. That means it is fetching the object from the connection pool. Also, the number of active connections in pools is showing 1. From this data we conclude, it is using connection pooling or the connection pooling is enabled.



**What happens in the Performance Monitor when connection pooling is Disabled?**

The data in the performance tool is not going to be updated automatically. You need to remove and add a new counter every time. First Remove all the counter from the Performance monitor tool as shown below.



Then again Modify the Code as shown below. Here, we are setting the Pooling to false which will disable connection pooling.

**using** *System;*

**using** *System.Data.SqlClient;*

**using** *System.Threading;*

**namespace** *ConnectionPooling*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string ConnectionString = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=false;";

**for** **(int** i = 0; i **<** 1000; i++**)**

**{**

SqlConnection connection = new SqlConnection**(**ConnectionString**)**;

connection.Open**()**;

Thread.Sleep**(**100**)**;

connection.Close**()**;

**}**

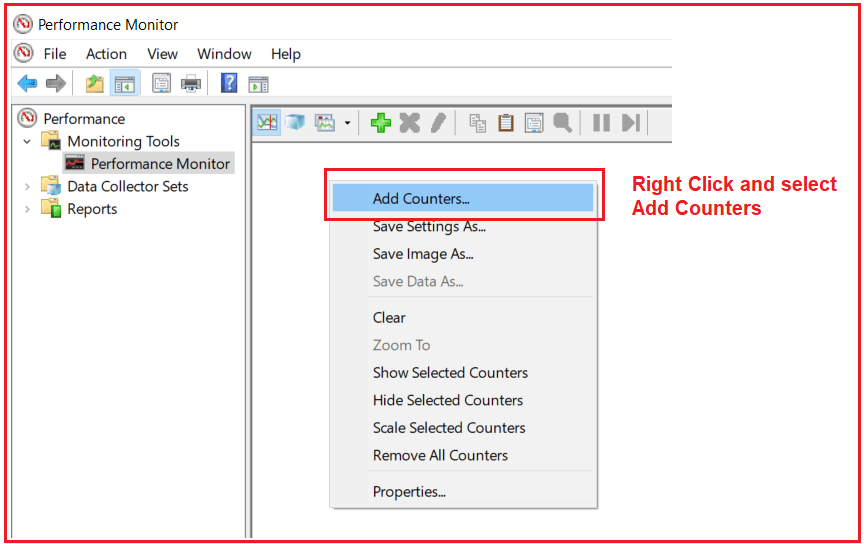
Console.ReadKey**()**;

**}**

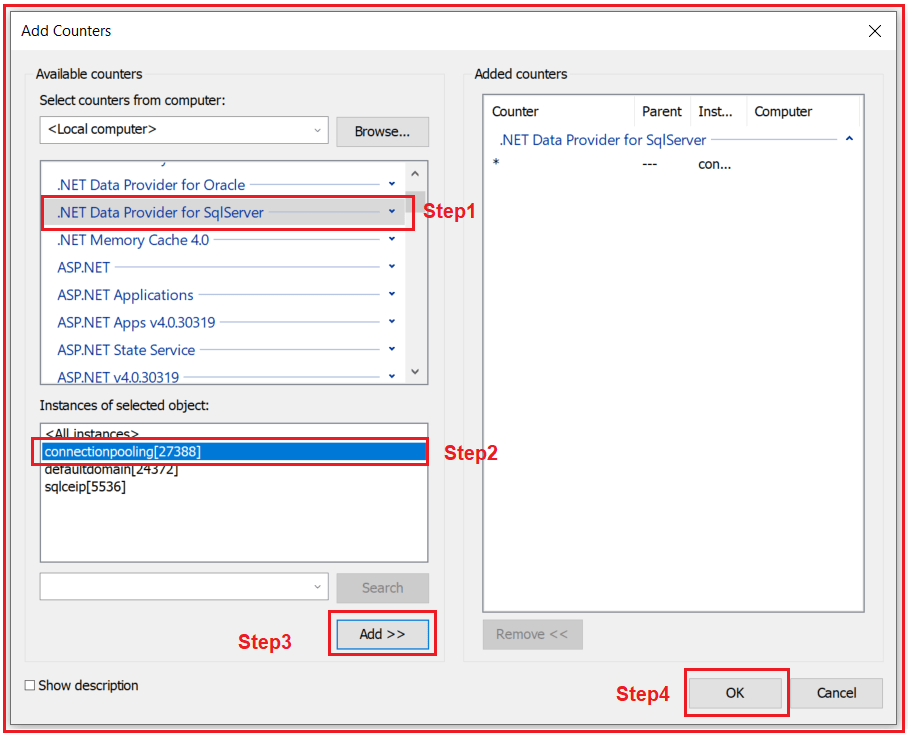
**}**

**}**

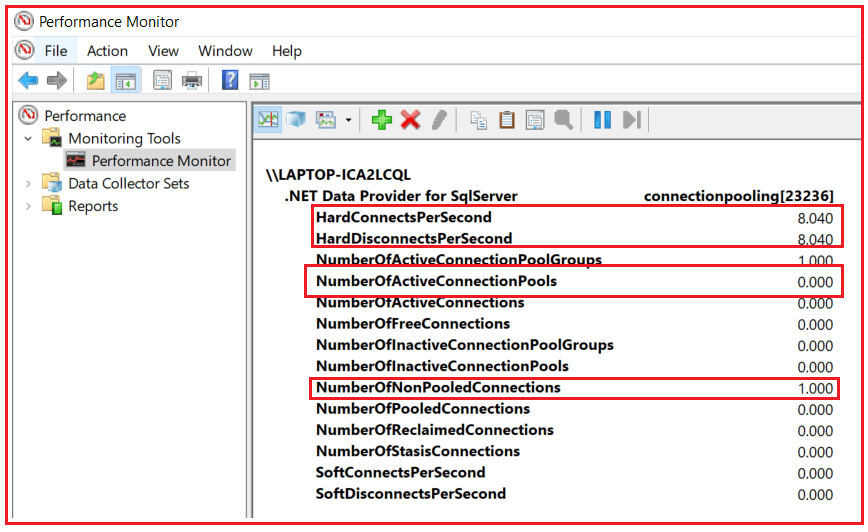
With the above changes in place, now, run the application. Once you run the application, go to the performance monitor tool and add a new performance counter. In order to Add Counter, simply right-click and then select the Add Counters option from the context menu as shown in the below image.



Once you click on the Add Counter, it will open the below Add Counters window. First, select the **.NET Data Provider for SQL Server** and then select the console application which should be in running mode. Then click on the **Add** button which adds the performance counter and finally click on the **OK** button as shown in the below image.



Once you click on the OK button, it will open the following report. As you can see in the below image, lots of hard connects and disconnects are happening. In other words, the ADO.NET open command is actually connecting to the SQL Server database. The second thing you can see there are no active connections in the pool. But you can see the Number of Non-Pooled Connections is 1. That means connection pooling is disabled.



**Note:** 1 connection pool is created for a unique connection string. A slight change in the connection string will create a new pool.

**Example: two connection objects with the same connection string**

Please have a look at the following example. As you can see both the connection strings are identical. So, whether we use ConnectionString1 or ConnectionString2, it will take the connection object from the pool.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *ConnectionPooling*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string ConnectionString1 = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=true;";

string ConnectionString2 = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=true;";

SqlConnection connection1 = new SqlConnection**(**ConnectionString1**)**;

connection1.Open**()**;

connection1.Close**()**;

SqlConnection connection2 = new SqlConnection**(**ConnectionString2**)**;

connection2.Open**()**;

connection2.Close**()**;

**}**

**}**

**}**

**Example: two connection objects with the different connection string**

Please have a look at the following example. As you can see both the connection strings are communicated to the same database and to the same computer. But, in connectionstring1 we have specified the computer name and in connectionstring2 we have specified localhost, In this case, the ADO.NET will create one connection pool for ConnectionString1 and another connection pool for ConnectionString2.

**using** *System;*

**using** *System.Data.SqlClient;*

**namespace** *ConnectionPooling*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

string ConnectionString1 = "data source=LAPTOP-ICA2LCQL\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=true;";

string ConnectionString2 = "data source=localhost\\SQLEXPRESS; initial catalog=ADODB; integrated security=True; Pooling=true;";

SqlConnection connection1 = new SqlConnection**(**ConnectionString1**)**;

connection1.Open**()**;

SqlConnection connection2 = new SqlConnection**(**ConnectionString2**)**;

connection2.Open**()**;

Console.ReadLine**()**;

connection1.Close**()**;

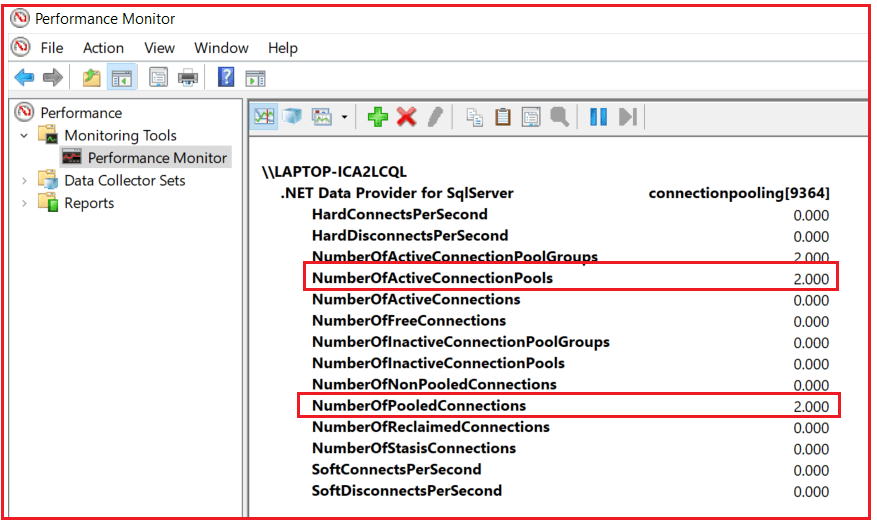
connection2.Close**()**;

**}**

**}**

**}**

Run the application and open the performance monitor tool and create a new counter and you should see two connection pools are created as shown in the below image.

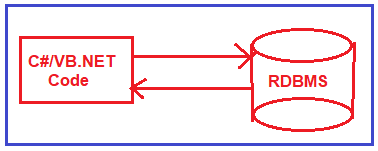


ADO.NET Architecture

**ADO.NET Architecture**

**What is ADO.NET?**

ADO stands for Active Data Objects. ADO is nothing but a component in .NET Framework that helps us to fetch data from different data sources to our C# and VB.NET code and probably you can send data from your C# or VB.NET code to different data sources.

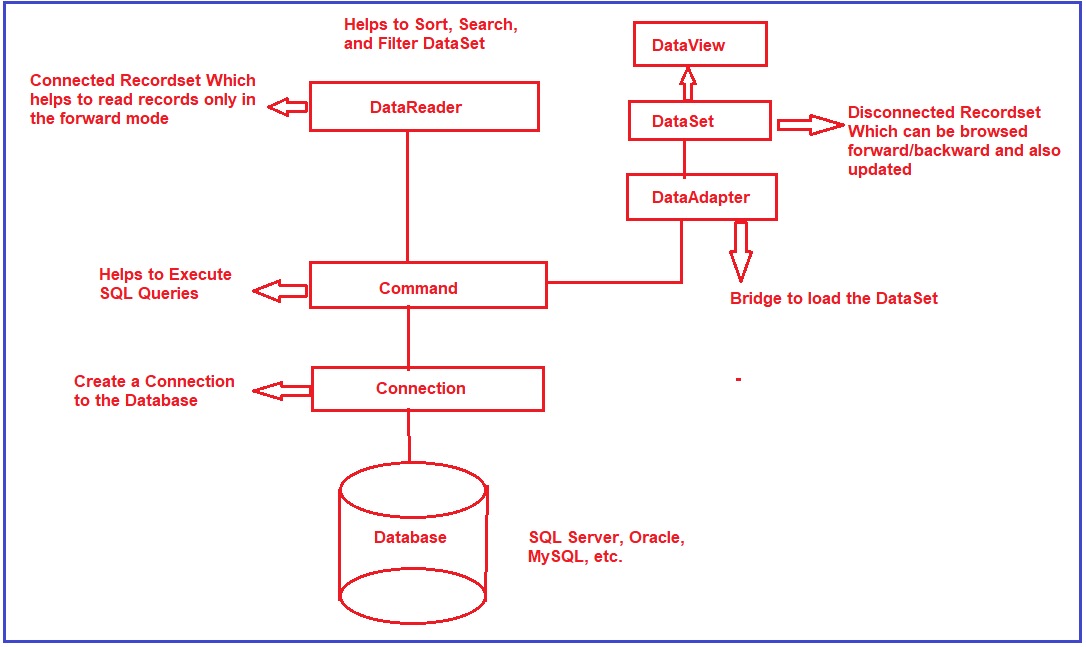


**The architecture of ADO.NET:**

The ADO.NET Architecture is comprising of 6 important components. They are as follows:

1. [**Connection**](https://dotnettutorials.net/lesson/ado-net-sqlconnection-class/)
2. [**Command**](https://dotnettutorials.net/lesson/ado-net-sqlcommand-class/)
3. [**DataReader**](https://dotnettutorials.net/lesson/ado-net-sqldatareader/)
4. [**DataAdapter**](https://dotnettutorials.net/lesson/ado-net-sqldataadapter/)
5. [**DataSet**](https://dotnettutorials.net/lesson/ado-net-dataset/)
6. **DataView**

From the above components, two components are compulsory. One is the command object and the other one is the connection object. Irrespective of the operations like Insert, Update, Delete and Select, the command and connection object you always need. For better understanding, please have a look at the following image.



Let us understand each of the components in detail.

**Connection:**

The first important component is the connection object. The connection object is required to connect with your backend database which can be SQL Server, Oracle, MySQL, etc. To create a connection object, you need at least two things. The first one is where is your database located i.e. the Machine name or IP Address or someplace where your database is located. And the second thing is the security credentials i.e. whether it is a windows authentication or user name and password-based authentication. So, the first is to create the connection object and the connection is required to connect to the backend data source.

**Command:**

The second important component is the command object. When we talk about databases like SQL Server, Oracle, MySQL, then understand SQL. The command object is the component where you go and write your SQL queries. Later you take the command object and execute it over the connection. Then you can fetch data or send data to the database using the command object and SQL queries.

**Note:** From the command object onwards, you can go in two different ways. One is you can go with the DataSet way and the other is, you can go with the DataReader way. Which way you need to choose, basically it will depend on the situation.

**DataReader:**

DataReader is a read-only connected recordset that helps us to read the records only in the forward mode. Here, you need to understand three things i.e. read-only, connected, and forward mode.

**DataSet:**

It is a disconnected recordset that can be browsed in both i.e. forward and backward. It is also possible to update via dataset. DataSet gets filled by somebody called DataAdapter.

**DataAdapter:**

The DataAdapter acts as a bridge between the command object and the dataset. What the DataAdapter does, it takes the data from the command object and fills the data set.

**DataView Class**

A DataView enables you to create different views of the data stored in a DataTable, a capability that is often used in data-binding applications. Using a DataView, you can expose the data in a table with different sort orders, and you can filter the data by row state or based on a filter expression.