## Connection Object

A primary function of any database application is connecting to a data source and retrieving the data that it contains. The .NET Framework data providers of ADO.NET serve as a bridge between an application and a data source, allowing you to execute commands as well as to retrieve data by using a **DataReader** or a **DataAdapter**. A key function of any database application is the ability to update the data that is stored in the database

In ADO.NET you use a **Connection** object to connect to a specific data source by supplying necessary information in a connection string. The **Connection** object you use depends on the type of data source.

Each .NET Framework data provider included with the .NET Framework has a **Connection** object: the .NET Framework Data Provider for OLE DB includes an OleDbConnection object, the .NET Framework Data Provider for SQL Server includes a SqlConnection object, the .NET Framework Data Provider for ODBC includes an OdbcConnection object, and the .NET Framework Data Provider for Oracle includes an OracleConnection object.

Steps to connect database:

1. Create Database.
2. Start writing connection string in VB.Net.
3. Set the provider.
4. Specify the data source.

Connection object have ConnectionString property. Depends on the parameter specified in the Connection String, ADO.Net Connection Object connects to the specified Database.

**Properties :**

**ConnectionString** : Gets/sets the connection string to open a database.
**Database** : Gets the name of the database to open
**DataSource** : Gets the name of the SQL Server to use.
**State** : Gets the connection's current state

**Methods :**

**Close** : Closes the connection to the data provider.
**Open** : Opens a database connection.

## Command Object :

- ❖ It's depended on Connection Object.
- ❖ Command objects are used to execute commands to a database across a data connection.
- ❖ The Command object in ADO.NET executes SQL statements and stored procedures against the data source specified in the connection object.
- ❖ The Command objects has a property called Command Text, which contains a String (Query) value that represents the command that will be executed in the Data Source.

There are many ways to initialize Command object:

For Example

```
Dim con As New SqlConnection
Dim cmd As SqlCommand
Dim str1 As String
Str1 = "Insert into stud values('"+ TextBox1.Text +"', '"+ TextBox1.Text +"')
con.Open()
cmd = New SqlCommand(str1, con)
cmd.ExecuteNonQuery()
```

```
con.close()
```
**OR**
```
Dim con As New SqlConnection
Dim cmd As SqlCommand
con.Open()
cmd.Connection = con
cmd.CommandText = “Insert into stud values(‘”+ TextBox1.Text +”’, ‘”+ TextBox1.Text
+”’)”
cmd.ExecuteNonQuery()
con.close()
```

**Property**

**CommandText** - Gets/sets the SQL statement (or stored procedure) for this command to execute.
**CommandType**- Gets/sets the type of the CommandText property (typically set to text for SQL).
**Connection**- Gets/sets the SqlConnection to use.
**Parameters-** Gets the command parameters.

**Methods**
**ExecuteNonQuery** Executes a non-row returning SQL statement, returning the number of affected rows.
**ExecuteReader** Creates a data reader using the command
**ExecuteScalar** Executes the command and returns the value in the first column in the first row of the result.

## DataAdapter Object :

The **SqlDataAdapter**, serves as a bridge between a DataSet and SQL Server for retrieving and saving data. The **SqlDataAdapter** provides this bridge by mapping Fill, which changes the data in the DataSet to match the data in the data source, and Update, which changes the data in the data source to match the data in the DataSet, using the appropriate Transact-SQL statements against the data source. The update is performed on a by-row basis. For every inserted, modified, and deleted row, the Update method determines the type of change that has been performed on it (**Insert**, **Update**, or **Delete**). Depending on the type of change, the **Insert**, **Update**, or **Delete** command template executes to propagate the modified row to the data source.
When the **SqlDataAdapter** fills a DataSet, it creates the necessary tables and columns for the returned data if they do not already exist. **SqlDataAdapter** is used in conjunction with SqlConnection and SqlCommand to increase performance when connecting to a SQL Server database.
The **SqlDataAdapter** also includes the SelectCommand, InsertCommand, DeleteCommand, UpdateCommand properties to facilitate the loading and updating of data.
When an instance of **SqlDataAdapter** is created, the read/write properties are set to initial values. For a list of these values, see the **SqlDataAdapter** constructor.
The InsertCommand, DeleteCommand, and UpdateCommand are generic templates that are automatically filled with individual values from every modified row through the parameters mechanism.
For every column that you propagate to the data source on Update, a parameter should be added to the **InsertCommand**, **UpdateCommand**, or **DeleteCommand**. The SourceColumn property of the DbParameter object should be set to the name of the column. This setting indicates that the

value of the parameter is not set manually, but is taken from the particular column in the currently processed row.

**Properties**
**DeleteCommand** Gets or sets a Transact-SQL statement or stored procedure to delete records from the data set.
**InsertCommand** Gets or sets a Transact-SQL statement or stored procedure to insert new records into the data source.
**SelectCommand** Gets or sets a Transact-SQL statement or stored procedure used to select records in the data source.
**TableMappings** Gets a collection that provides the master mapping between a source table and a DataTable. (Inherited from DataAdapter.)
**UpdateCommand** Gets or sets a Transact-SQL statement or stored procedure used to update records in the data source.

**Methods**
Fill Adds or updates rows in a data set to match those in the data source.
Creates a table named "Table" by default Update Updates the data store by calling the INSERT, UPDATE, or DELETE statements for each inserted, updated, or deleted row in the given dataset.

**For Example :**
```
Dim da As SqlDataAdapter
Dim ds As New DataSet
str1 = "select * from stud"
da = New SqlDataAdapter(str1, con)
da.Fill(ds)
```

## DataSet Object

- ❖ ADO.NET caches data locally on the client and store that data into DataSet.
- ❖ The dataset is a disconnected, I-memory representation of data.
- ❖ It’s not exact copy the database.
- ❖ It can be considered as a local copy of the some portions of the database.
- ❖ The DataSet contains a collection of one or more DataTable objects made up of rows and columns of data.
- ❖ Tables can be identified in DataSet using DataSet’s Tables property.
- ❖ It also contains primary key, foreign key, constraint and relation information about the data in the DataTable objects.
- ❖ DataSet are also fully XML-featured.
- ❖ Whaterer operations are made by the user it is stored temporary in the DataSet, when the use of this DataSet is finished, changes can be made back to the central database for updating.
- ❖ DataSet doesn’t “know” where the data it contains came from and if fact it can contain data from multiple sources.
- ❖ The DataSet is populated DataAdapter’s Fill method.

The **DataSet** is a major component of the ADO.NET architecture. The **DataSet** consists of a collection of DataTable objects that you can relate to each other with Data Relation objects. You can also enforce data integrity in the **DataSet** by using the UniqueConstraint and ForeignKeyConstraint objects. For further details about working with **DataSet** objects.

Whereas DataTable objects contain the data, the DataRelationCollection allows you to navigate though the table hierarchy. The tables are contained in a DataTableCollection accessed through the Tables property. When accessing DataTable objects, note that they are conditionally case sensitive.

For example, if one DataTable is named "mydatatable" and another is named "Mydatatable", a string used to search for one of the tables is regarded as case sensitive. However, if "mydatatable" exists and "Mydatatable" does not, the search string is regarded as case insensitive. For more information about working with DataTable objects.

A **DataSet** can read and write data and schema as XML documents. The data and schema can then be transported across HTTP and used by any application, on any platform that is XMLenabled. You can save the schema as an XML schema with the WriteXmlSchema method, and both schema and data can be saved using the WriteXml method. To read an XML document that includes both schema and data, use the ReadXml method.

In a typical multiple-tier implementation, the steps for creating and refreshing a **DataSet**, and in turn, updating the original data are to:

1. Build and fill each DataTable in a **DataSet** with data from a data source using a DataAdapter.
2. Change the data in individual DataTable objects by adding, updating, or deleting DataRow objects.
3. Invoke the GetChanges method to create a second **DataSet** that features only the changes to the data.
4. Call the Update method of the DataAdapter, passing the second **DataSet** as an argument.
5. Invoke the Merge method to merge the changes from the second **DataSet** into the first.
6. Invoke the AcceptChanges on the **DataSet**. Alternatively, invoke RejectChanges to cancel the changes.

**Properties**
**Relations** Get the collection of relations that link tables and allow navigation from parent tables to child tables.
**Tables** Gets the collection of tables contained in the DataSet.

**For Example :**

```
Dim da As SqlDataAdapter
Dim ds As New DataSet
str1 = "select * from stud"
da = New SqlDataAdapter(str1, con)
da.Fill(ds)
DataGridView1.DataSource = ds.Tables(0)
```

## DataReader Object

Provides a way of reading a forward-only stream of rows from a SQL Server database

To create a **SqlDataReader**, you must call the ExecuteReader method of the SqlCommand object, instead of directly using a constructor.

While the **SqlDataReader** is being used, the associated SqlConnection is busy serving the **SqlDataReader**, and no other operations can be performed on the SqlConnection other than closing it. This is the case until the Close method of the **SqlDataReader** is called. For example, you cannot retrieve output parameters until after you call Close.

Changes made to a result set by another process or thread while data is being read may be visible to the user of the **SqlDataReader**. However, the precise behavior is timing dependent.

IsClosed and RecordsAffected are the only properties that you can call after the **SqlDataReader** is closed. Although the RecordsAffected property may be accessed while the **SqlDataReader** exists, always call Close before returning the value of RecordsAffected to guarantee an accurate return value.

**Properties**
**Connection** Gets the SqlConnection associated with the SqlDataReader.
**IsClosed** Retrieves a Boolean value that indicates whether the specified SqlDataReader instance has been closed. (Overrides DbDataReader.IsClosed.)
**Item** Overloaded. Gets the value of a column in its native format.

**Methods**
**Close** Closes the SqlDataReader object. (Overrides DbDataReader. Close ().)
**Read** Advances the SqlDataReader to the next record. (Overrides DbDataReader. Read ().)

```
Dim reader As SqlDataReader
sql = " Select * from stud"
Try
        con.Open()
        cmd = New SqlCommand(sql, con)
        reader = cmd.ExecuteReader()
        While reader.Read()
                MsgBox(reader.Item(0) & " - " & reader.Item(1))
        End While
        reader.Close()
        cmd.Dispose()
        con.Close()
Catch ex As Exception
        MsgBox("Can not open connection ! ")
End Try
```

## DataGridView Control

The **DataGridView** control provides a powerful and flexible way to display data in a tabular format. You can use the **DataGridView** control to show read-only views of a small amount of data, or you can scale it to show editable views of very large sets of data.

You can extend the **DataGridView** control in a number of ways to build custom behaviors into your applications. For example, you can programmatically specify your own sorting algorithms, and you can create your own types of cells. You can easily customize the appearance of the

**DataGridView** control by choosing among several properties. Many types of data stores can be used as a data source, or the **DataGridView** control can operate with no data source bound to it.

The **DataGridView** control provides a customizable table for displaying data. The **DataGridView** class allows customization of cells, rows, columns, and borders through the use of properties such as DefaultCellStyle, ColumnHeadersDefaultCellStyle, CellBorderStyle, and GridColor.

You can use a **DataGridView** control to display data with or without an underlying data source. Without specifying a data source, you can create columns and rows that contain data and add them directly to the **DataGridView** using the Rows and Columns properties. You can also use the Rows collection to access DataGridViewRow objects and the DataGridViewRow.Cells property to read or write cell values directly. The Item indexer also provides direct access to cells.

As an alternative to populating the control manually, you can set the DataSource and DataMember properties to bind the **DataGridView** to a data source and automatically populate it with data. For more information, see Displaying Data in the Windows Forms DataGridView Control.

When working with very large amounts of data, you can set the VirtualMode property to **true** to display a subset of the available data. Virtual mode requires the implementation of a data cache from which the **DataGridView** control is populated. For more information, see Data Display Modes in the Windows Forms DataGridView Control.

For additional information about the features available in the **DataGridView** control, see DataGridView Control (Windows Forms). The following table provides direct links to common tasks.

**Explain the steps to bind DataGridView.**

| | |
|---|---|
| Step : 1 | Take New VB.NET Project |
| Step : 2 | Add database in your project (Named : dbemp.mdf)<br>Add New Table in database file (Named : emp ) from the Server Explorer Window Add some records in that table |
| Step : 3 | Now add emp table to the DBEmpDataSet.xsd file |
| Step : 4 | Add DataGridView control on the form |
| Step : 5 | Select the Choose Data Source<br>In that click on Other Data Source<br>Project Data Source<br>DBEmpDataSet<br>Emp Table Name |
| Step : 6 | At the end, Run the project |

**Binding Data Grids**
As we've already seen, you can use data grids to display entire data tables. To bind a data grid to a table, you can set the data grid's **DataSource** property (usually to a dataset, such as **dsDataSet**) and **DataMember** property (usually to text naming a table like "authors"). At run time, you can set both of these properties at once with the built-in data grid method **SetDataBinding** (data grids are the only controls that have this method):

```
DataGrid1.SetDataBinding(dsDataSet, "authors")
```

You can use these data sources with the data grid's **DataSource** property:

- **DataTable** objects
- **DataView** objects
- **DataSet** objects
- **DataViewManager** objects
- single dimension arrays

To determine which cell was selected by the user, use the **CurrentCell** property. You can change the value of any cell using the **Item** property, which can take either the row or column indexes of the cell. And you can use the **CurrentCell Changed** event to determine when the user selects another cell.