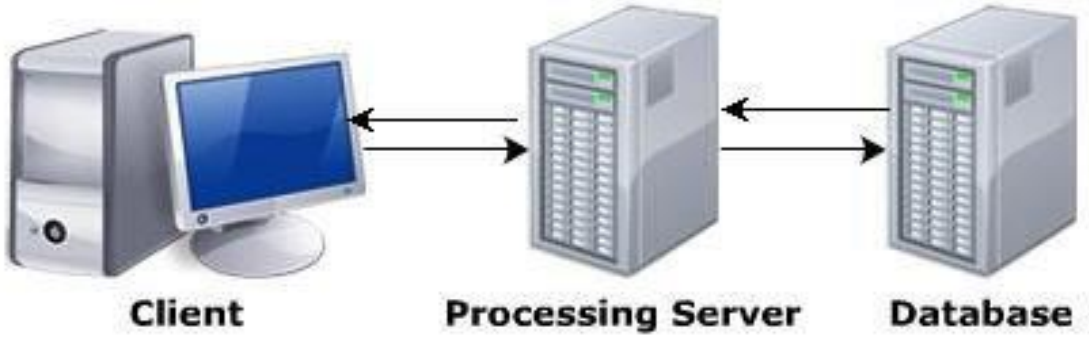
Accessing Database with ADO.NET

Objectives

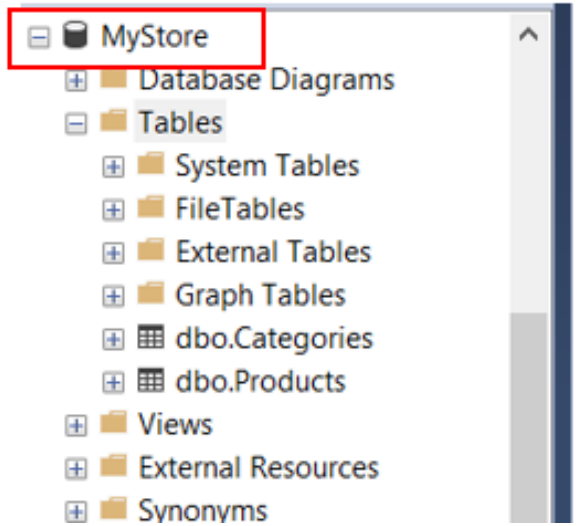
- Overview ADO.NET
- Explain ADO.NET data access history
- Describe data access architecture in .NET
- Explain the benefits of ADO.NET
- Describe the connected and disconnected data access approach
- Demo using ADO.NET Data Provider Factory Model
- Demo accessing database in WinForm Application using ADO.NET
- Demo using Store procedures in ADO.NET
- Overview about 3-Layers and 3-Tiers Architecture

What is Database?

- Database is a collection of related records
- The information in DB is stored in such a way that it is easier to access, manage, and update the data
- Data from the DB can be accessed using any one of the following architectures:
  - Single-tier architecture
  - Two-tier architecture
  - Three-tier architecture



- Create a sample database named **MyStore** for demonstrations



Products

	Column Name	Data Type	Nullable	Identity
🔑	ProductID	int	No	☑
	ProductName	nvarchar(40)	No	☐
	UnitPrice	money	No	☐
	UnitsInStock	smallint	No	☐
	CategoryID	int	No	☐
				☐

Categories

	Column Name	Data Type	Nullable	Identity
🔑	CategoryID	int	No	☑
	CategoryName	nvarchar(15)	No	☐
				☐

T470S.MyStore - dbo.Products

ProductID	ProductName	UnitPrice	UnitsInStock	CategoryID
1	Genen Shouyu	50.0000	39	1
2	Alice Mutton	30.0000	17	1
3	Aniseed Syrup	40.0000	13	3
4	Perth Pasties	22.0000	53	2
5	Carnarvon Tigers	21.3500	0	4
6	Gula Malacca	25.0000	120	2
7	Steeleye Stout	30.0000	15	7
8	Chocolade	40.0000	6	5
9	Mishi Kobe Niku	97.0000	29	6
10	Ikura	31.0000	31	8

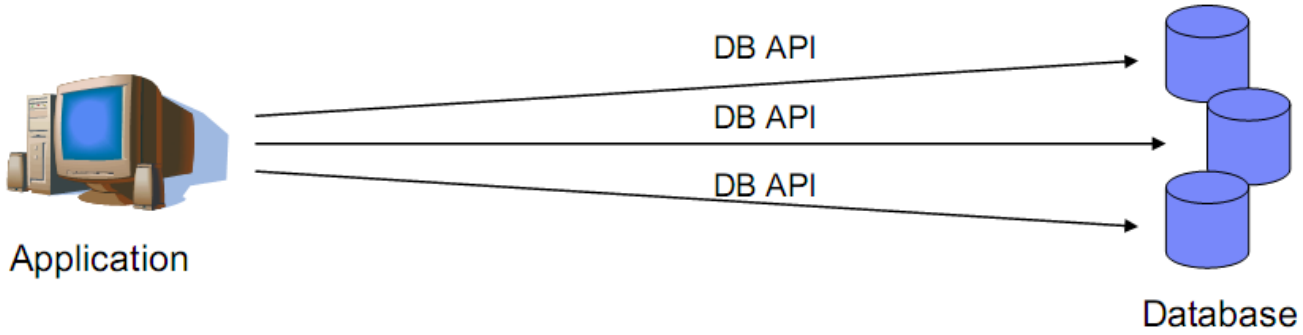
.MyStore - dbo.Categories

CategoryID	CategoryName
1	Beverages
2	Condiments
3	Confections
4	Dairy Products
5	Grains/Cereals
6	Meat/Poultry
7	Produce
8	Seafood
NULL	*NULL*

ADO.NET Data Access History

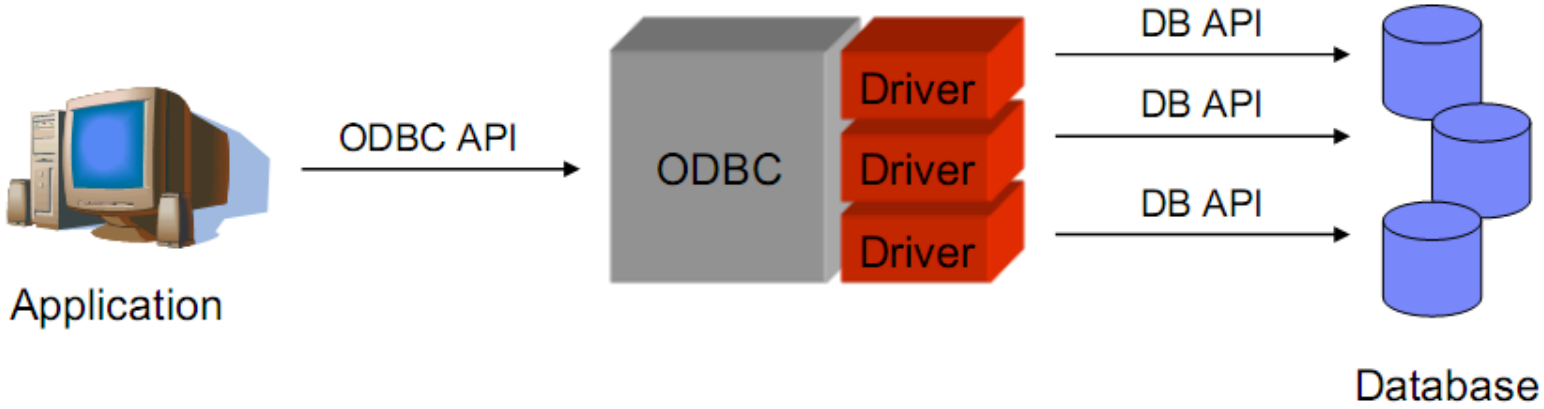
Universal Data Access (Native API)

- Database management systems provide APIs that allow application programmers to create and access databases
- The set of APIs that each manufacturer's system supplies is unique to that manufacturer. Microsoft has long recognized that it is inefficient and error prone for an applications programmer to attempt to master and use all the APIs for the various available database management systems
- What's more, if a new database management system is released, an existing application can't make use of it without being rewritten to understand the new APIs



Open Database Connectivity (ODBC)

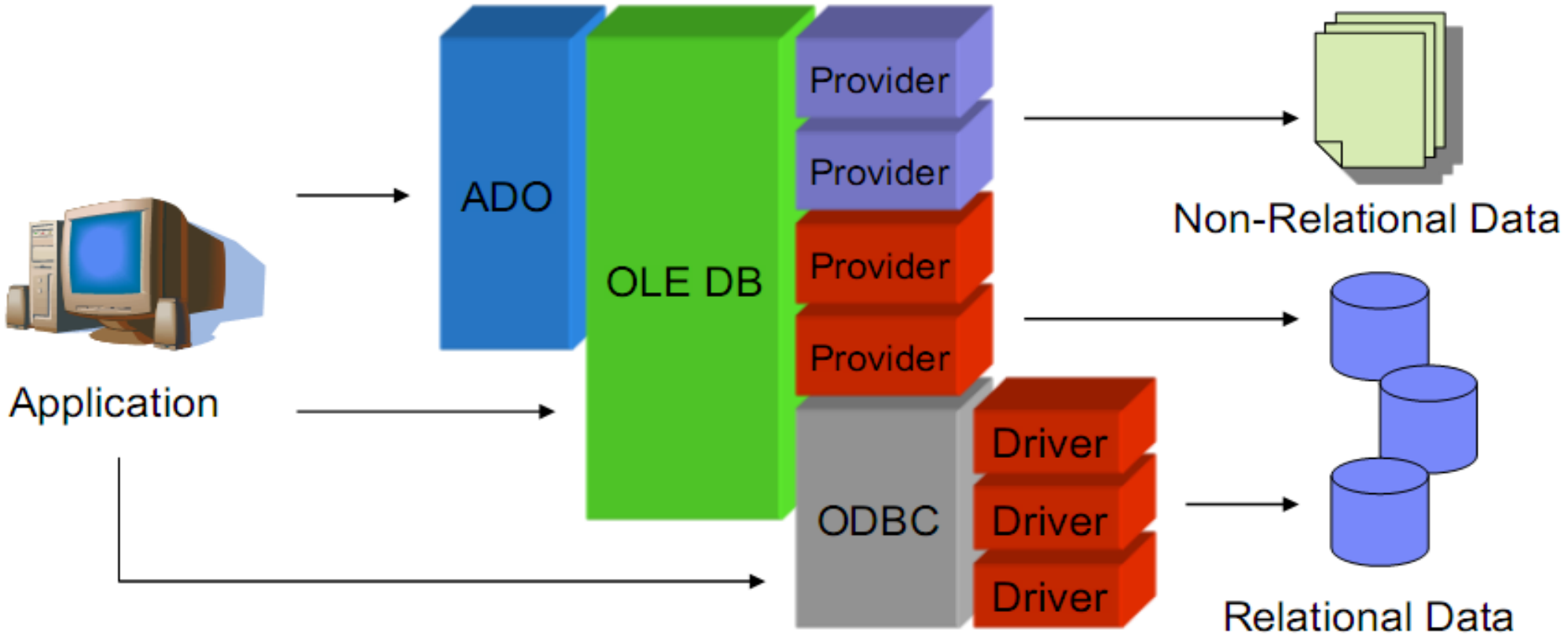
- Open Database Connectivity (ODBC) helped address the problem of needing to know the details of each DBMS used. ODBC provides a single interface for accessing a number of database systems
- ODBC provides a driver model for accessing data. Any database provider can write a driver for ODBC to access data from their database system. This enables developers to access that database through the ODBC drivers instead of talking directly to the database system



OLEDB and ADO (ActiveX Data Objects)

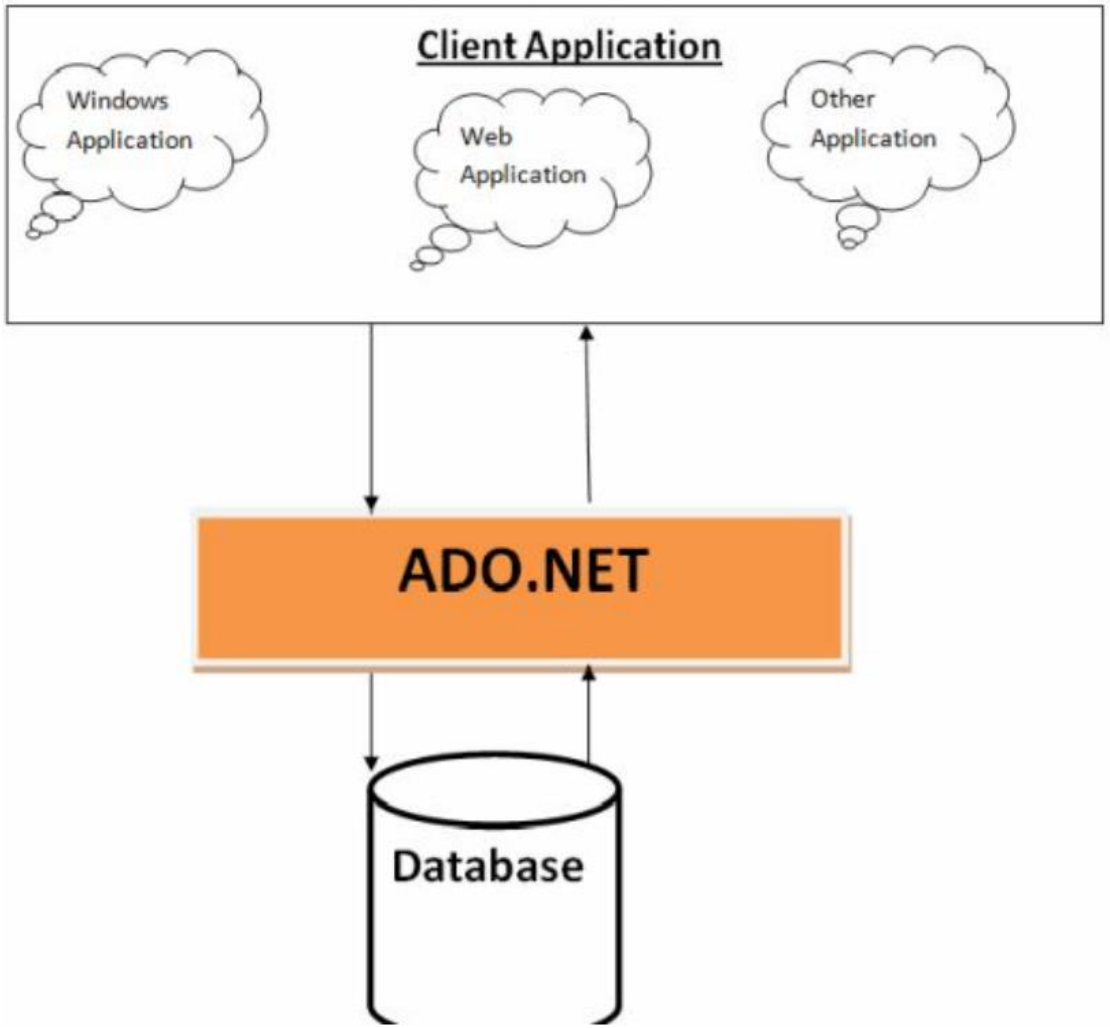
- OLE-DB is also much less dependent upon the physical structure of the database. It supports both relational and hierarchical data sources, and does not require the query against these data sources to follow a SQL structure
- Microsoft introduced ActiveX Data Objects (ADO) primarily to provide a higher-level API for working with OLE-DB. With this release, Microsoft took many of the lessons from the past to build a lighter, more efficient, and more universal data access API

OLEDB and ADO (ActiveX Data Objects)

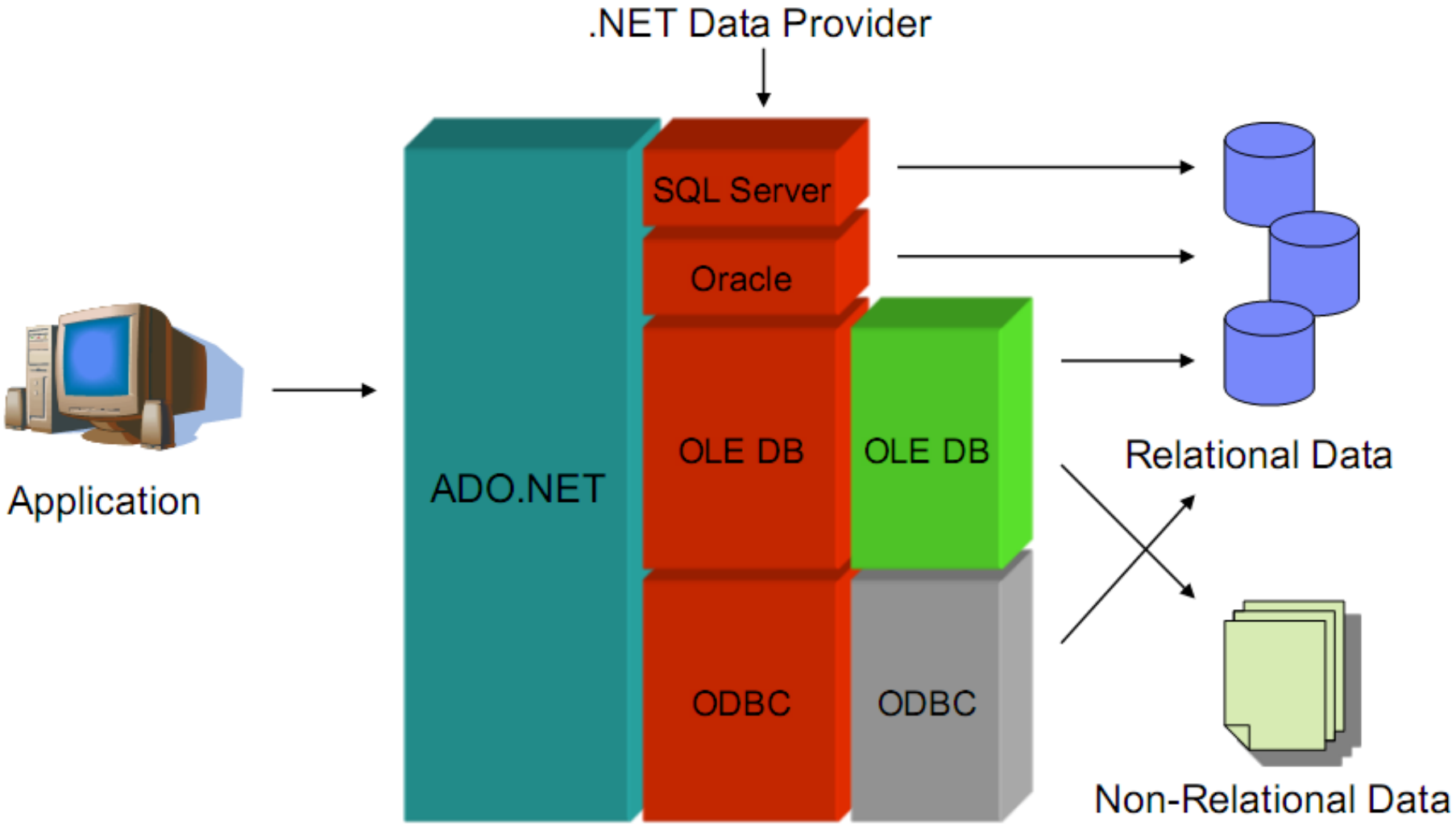


ADO.NET Overview

- Is the data access technology, which allows to access data from various data sources
- Is a part of .NET: The technology can be used for all .NET-base applications
- Supports disconnected data architecture: Connection to the data source is established only required



ADO.NET Overview



ADO.NET Overview

- ADO.NET provides consistent access to data sources such as SQL Server and XML, and to data sources exposed through OLE DB and ODBC
- Use XML to interact with the database: All the data in the database is converted into XML format for database related operations
- Data-sharing consumer applications can use ADO.NET to connect to these data sources and retrieve, handle, and update the data that they contain
- ADO.NET separates data access from data manipulation into discrete components that can be used separately or in tandem

ADO.NET Overview

- ADO.NET provides functionality to developers who write managed code similar to the functionality provided to native component object model (COM) developers by ActiveX Data Objects (ADO)
- ADO.NET includes .NET data providers for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in an ADO.NET DataSet object in order to be exposed to the user in an ad hoc manner, combined with data from multiple sources, or passed between tiers
- The DataSet object can also be used independently of a .NET data provider to manage data local to the application or sourced from XML
- ADO.NET separates data access from data manipulation into discrete components that can be used separately or in tandem

ADO.NET Features

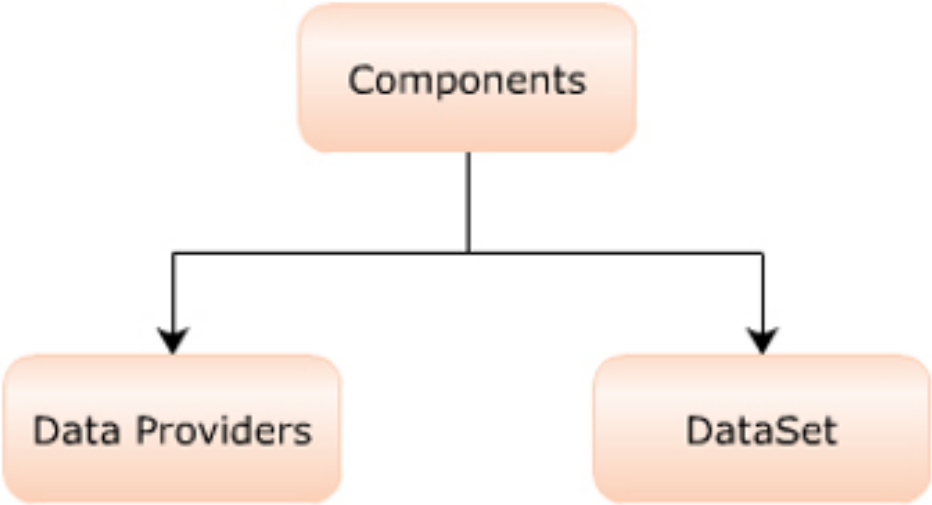
- Asynchronous processing: Enable time-consuming application running in the background
- Multiple Active Result Sets (MARS): Allow to execute multiple batches in a connection
- Bulk copy operations: Allow to copy large files into tables or views
- Batch processing
- Tracing: Monitor the excution of code, identify problems when executing code and fix them
- Connection pooling control: Collects all the opened Database connections in a connection pool and get a connection from the pool for client rather than create new one

Benefits of ADO.NET

- Asynchronous processing: Enable time-consuming application running in the background
- Simplified Programming Model
- Interoperability: XML is the default format used for transmitting datasets across network, any component can read XML format is able to process data
- Maintainability and Programmability
- Performance: Does not require data-type conversion while transmitting data through the tier
- Scalability

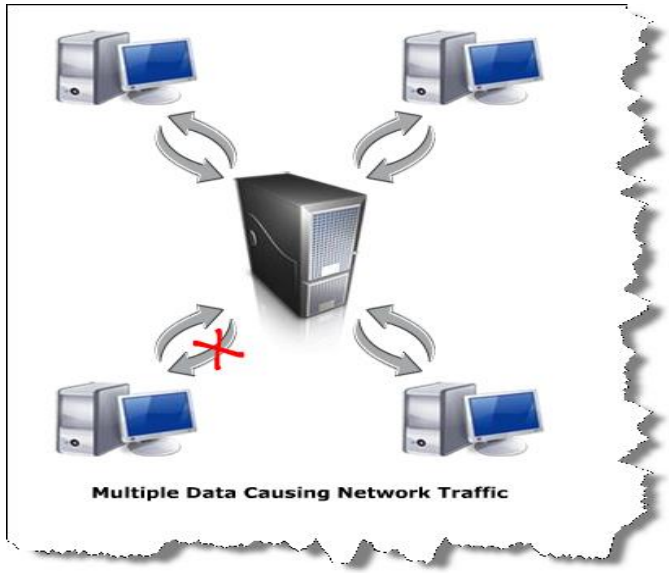
ADO.NET Architecture

- The two important components of ADO.NET used for processing the data in Database are:
  - **Data providers**: Provide and maintain connection to the database
  - **Dataset**: Is the required portion in database that is extracted and maintained in the form of a table as a local copy in the client system

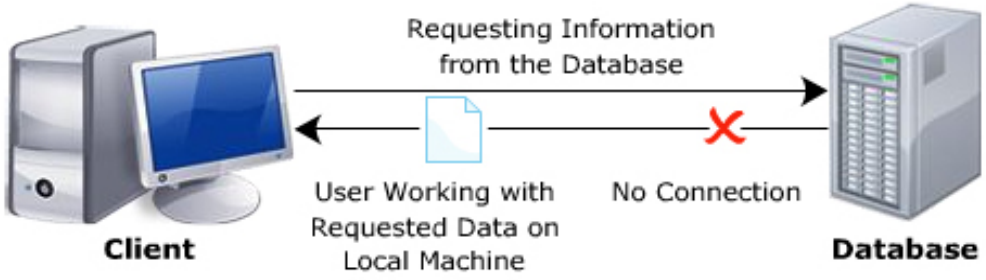


Data Access Models of ADO.NET

- Connected data access:
  - Connection to the DB is established when requested by an application
  - This connection is kept open till the application is closed

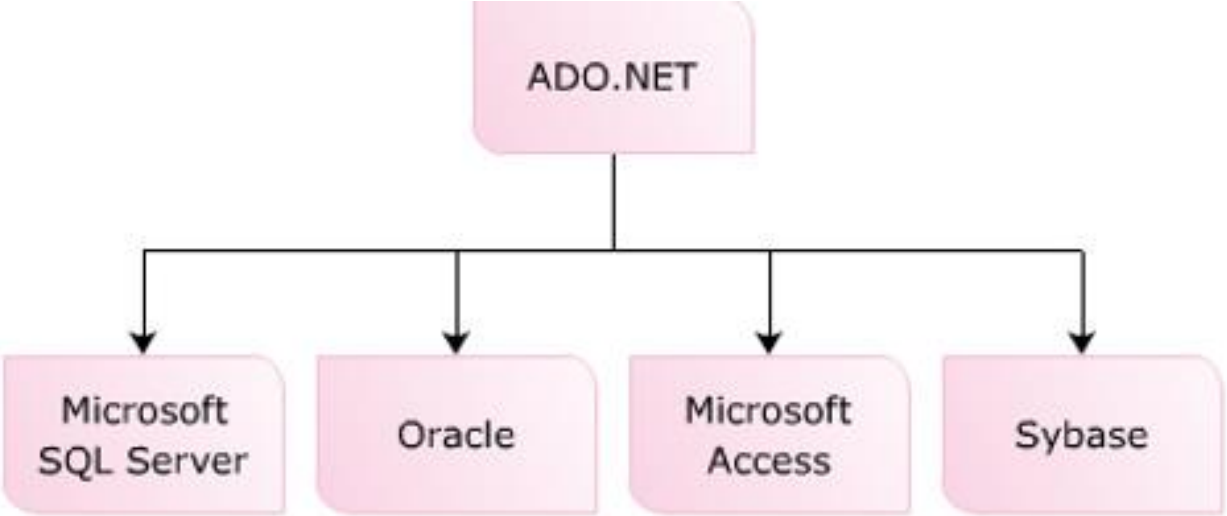


- Disconnected data access:
  - Connection to the DB is established when the application forwards a request
  - Once the request is processed, connection is automatically closed



.NET Data Providers

- A .NET data provider is used for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in a DataSet in order to be exposed to the user as needed, combined with data from multiple sources, or remoted between tiers
- NET data providers are lightweight, creating a minimal layer between the data source and code, increasing performance without sacrificing functionality



.NET Data Providers

- As with all of .NET (.NET Core), data providers ship as NuGet packages. There are several supported by Microsoft as well as a multitude of third-party providers available
- The followings table documents some of the data providers supported by Microsoft:

.NET Data Provider	Namespace/NuGet Package Name
Microsoft SQL Server	**Microsoft.Data.SqlClient**
ODBC	System.Data.Odbc
OLE DB (Windows only)	System.Data.OleDb

- Oracle Data Provider for .NET (ODP.NET) supported by Oracle in Nuget Package

Core Objects of .NET Data Providers

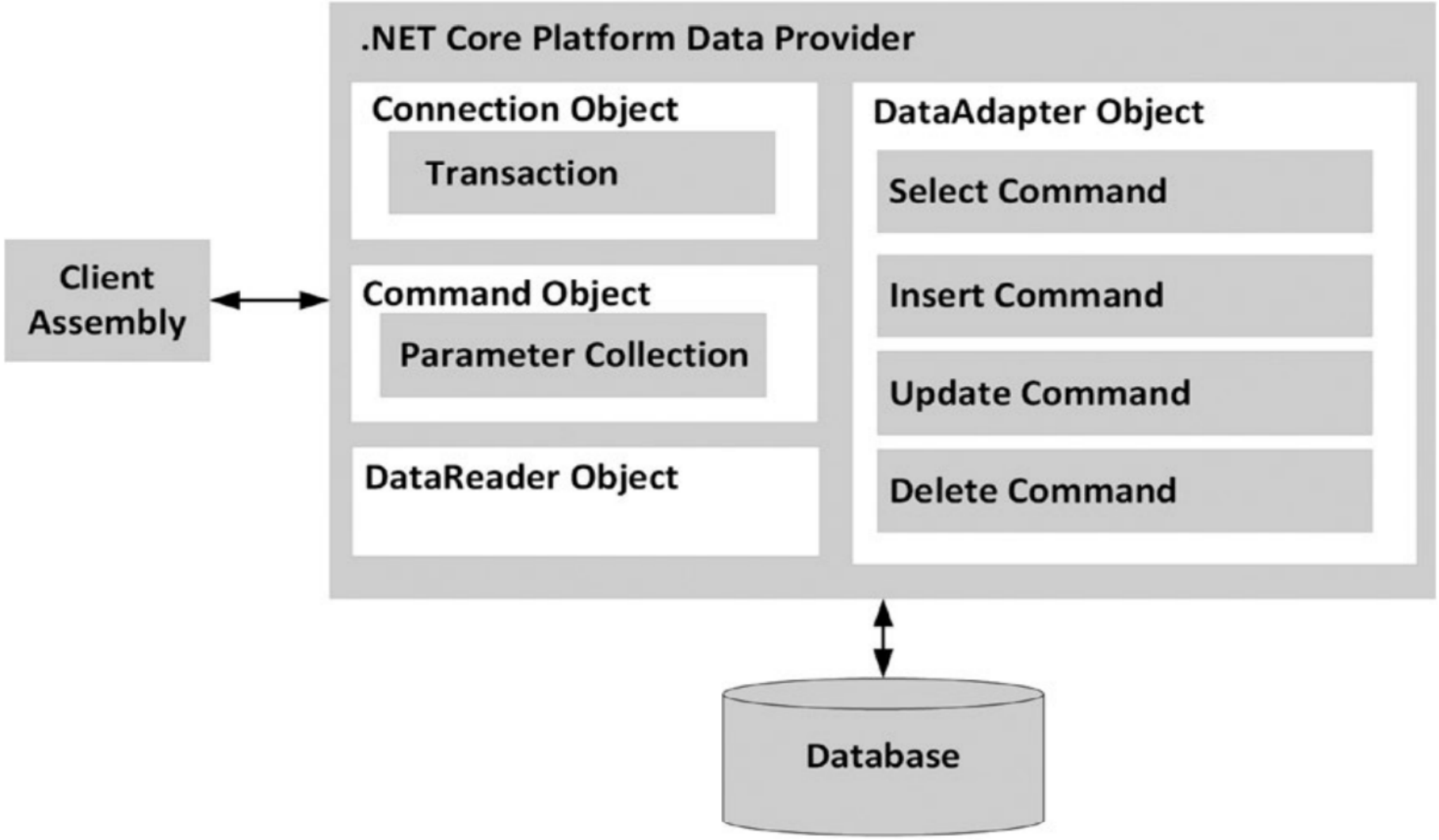
Base Class	Interface	Description
DbConnection	**IDbConnection**	Provides the ability to connect to and disconnect from the data store. Connection objects also provide access to a related transaction object
DbCommand	**IDbCommand**	Represents a SQL query or a stored procedure. Command objects also provide access to the provider's data reader object
DbDataReader	**IDataReader, IDataRecord**	Provides forward-only, read-only access to data using a server-side cursor
DbDataAdapter	**IDataAdapter, IDbDataAdapter**	Transfers DataSets between the caller and the data store. Data adapters contain a connection and a set of four internal command objects used to select, insert, update, and delete information from the data store
DbParameter	**IDataParameter, IDbDataParameter**	Represents a named parameter within a parameterized query
DbTransaction	**IDbTransaction**	Encapsulates a database transaction

Core Objects of .NET Data Providers

- The following table outlines the four core objects that make up a .NET data provider:

Object	Description
Connection	Establishes a connection to a specific data source. The base class for all Connection objects is the DbConnection class
Command	Executes a command against a data source. Exposes Parameters and can execute in the scope of a Transaction from a Connection. The base class for all Command objects is the DbCommand class
DataReader	Reads a forward-only, read-only stream of data from a data source. The base class for all DataReader objects is the DbDataReader class
DataAdapter	Populates a DataSet and resolves updates with the data source. The base class for all DataAdapter objects is the DbDataAdapter class

Core Objects of .NET Data Providers



System.Data Namespace

- This namespace contains types that are shared among all ADO.NET data providers, regardless of the underlying data store
- System.Data contains types that represent various database primitives (e.g., tables, rows, columns, and constraints), as well as the common interfaces implemented by data provider objects

Type	Description
Constraint	Represents a constraint for a given DataColumn object
DataColumn	Represents a single column within a DataTable object
DataRelation	Represents a parent-child relationship between two DataTable objects
DataRow	Represents a single row within a DataTable object
DataSet	Represents an in-memory cache of data consisting of any number of interrelated DataTable objects

System.Data Namespace

Type	Description
DataTable	Represents a tabular block of in-memory data
DataTableReader	Allows us to treat a DataTable as a fire-hose cursor (forward-only, read-only data access)
DataView	Represents a customized view of a DataTable for sorting, filtering, searching, editing, and navigation
IDataAdapter	Defines the core behavior of a data adapter object
IDataParameter	Defines the core behavior of a parameter object
IDataReader	Defines the core behavior of a data reader object
IDbCommand	Defines the core behavior of a command object
IDbDataAdapter	Extends IDataAdapter to provide additional functionality of a data adapter object
IDbTransaction	Defines the core behavior of a transaction object

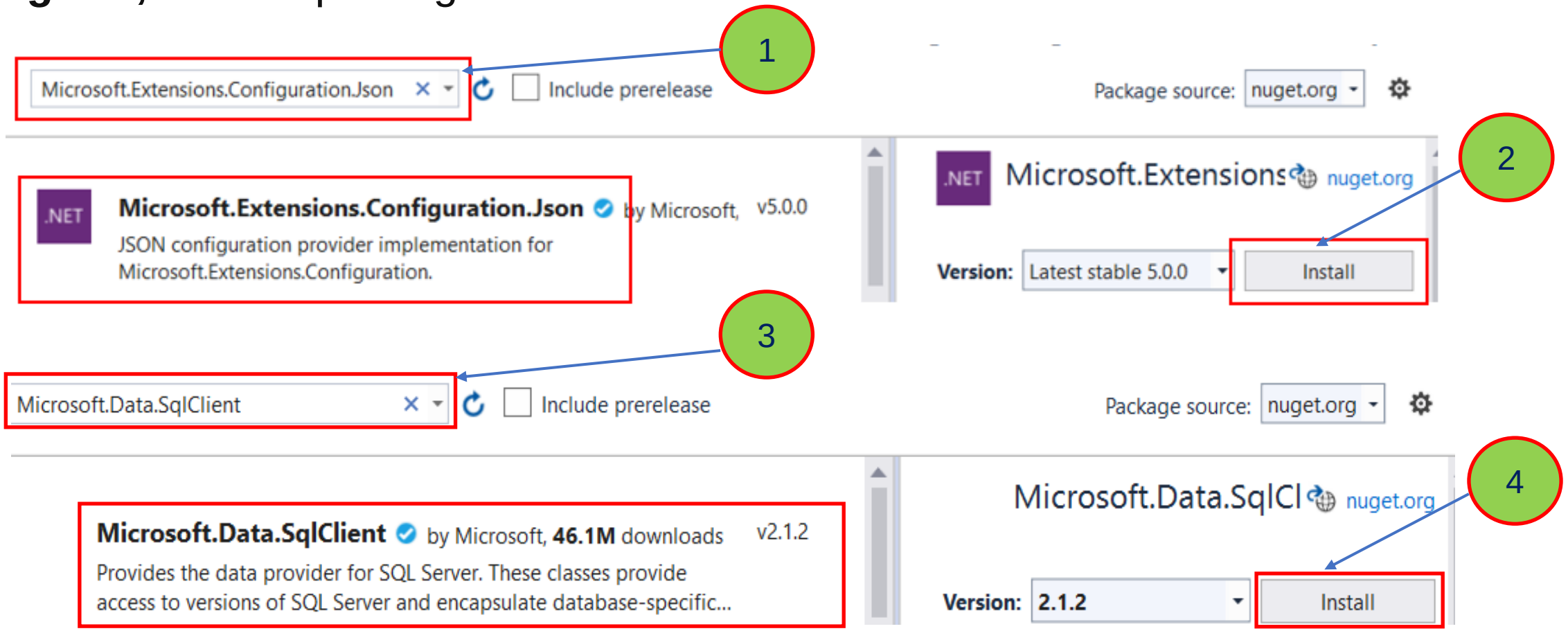
ADO.NET Data Provider Factory Model

- The .NET data provider factory pattern allows us to build a single codebase using generalized data access types
- The classes within a data provider all derive from the same base classes defined within the System.Data.Common namespace:
  - DbCommand: The abstract base class for all command classes
  - DbConnection: The abstract base class for all connection classes
  - DbDataAdapter: The abstract base class for all data adapter classes
  - DbDataReader: The abstract base class for all data reader classes
  - DbParameter: The abstract base class for all parameter classes
  - DbTransaction: The abstract base class for all transaction classes

ADO.NET Data Provider Factory Model Demonstration

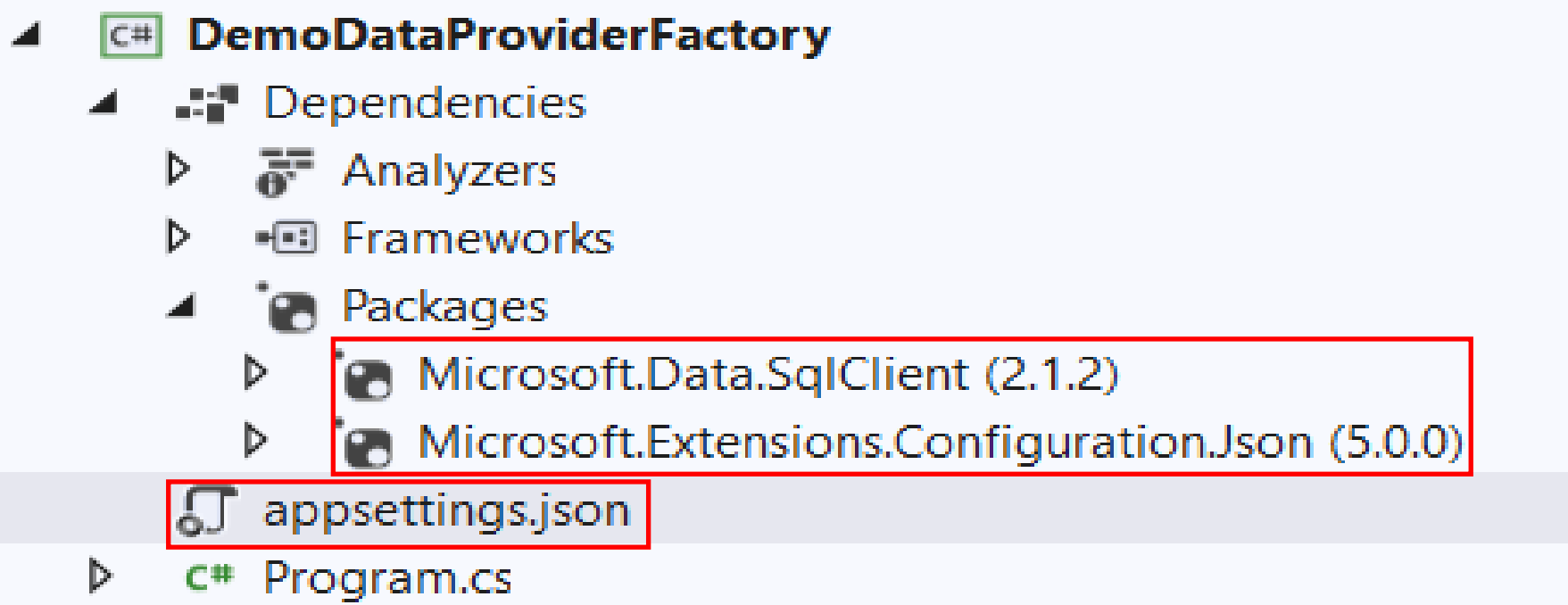
1.Create a Console app named **DemoDataProviderFactory**

2.Install **02** packages from Nuget: Right-click on the Project, seclect **Manage Nuget Package… , search** package name then click **install** as follows:



3. Right-click on the project | **Add | New Item,** select **JavaScript JSON Configuration File** then rename to **appsettings.json ,** click **Add** and write contents as follows:

```
{
  "ConnectionString": {
    "MyStoreDB": "Server=(local);uid=sa;pwd=123;database=MyStore"
  }
}
```



4.Right-click on the project, select **Edit Project File** and write config information as follows then press **Crtl+S** to save:

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Microsoft.Data.SqlClient" Version="2.1.2" />
    <PackageReference Include="Microsoft.Extensions.Configuration.Json" Version="5.0.0" />
  </ItemGroup>
  <ItemGroup>
    <None Update="appsettings.json">
      <CopyToOutputDirectory>Always</CopyToOutputDirectory>
    </None>
  </ItemGroup>
</Project>
```

5. Write codes for the **Program.cs** as follows then press **Crtl+F5** to run project:

```
using System.Data.Common;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;
using System.IO;
using System;
using System.Data;
```

```
class Program{
    // Get connection string from appsettings.json
    static string GetConnectionString(){
        IConfiguration config = new ConfigurationBuilder()
                                    .SetBasePath(Directory.GetCurrentDirectory())
                                    .AddJsonFile("appsettings.json", true, true)
                                    .Build();
        var strConnection = config["ConnectionString:MyStoreDB"];
        return strConnection;
    }//end GetConnectionString
    static void ViewProducts(){
        DbProviderFactory factory = SqlClientFactory.Instance;
        // Get the connection object.
        using DbConnection connection = factory.CreateConnection();
        if (connection == null){
            Console.WriteLine($"Unable to create the connection object.");
            return;
        }
        connection.ConnectionString = GetConnectionString();
        connection.Open();
```

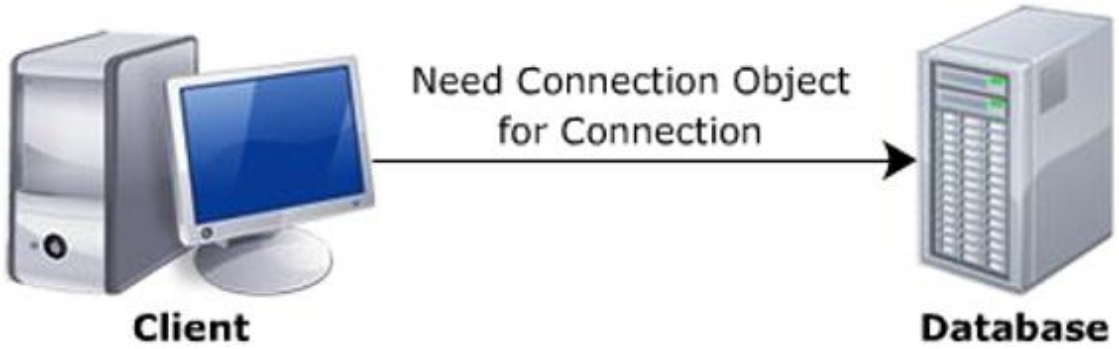
```
        // Make command object.
        DbCommand command = factory.CreateCommand();
        if (command == null){
            Console.WriteLine($"Unable to create the command object.");
            return;
        }
        command.Connection = connection;
        command.CommandText = "Select ProductID,ProductName From Products";
        // Print out data with data reader.
        using DbDataReader dataReader = command.ExecuteReader();
        Console.WriteLine("***** Product List *****");
        while (dataReader.Read()){
            Console.WriteLine($"ProductID: {dataReader["ProductId"]}, " +
                $"ProductName: {dataReader["ProductName"]}.");
        }
    }//end ViewProducts
    static void Main(string[] args){
        ViewProducts();
        Console.ReadLine();
    }//end Main
}//end Program
```

```
D:\Demo\FU\Basic.NET\Slot_17_18_ADO.NET\DemoDataProviderFactory\bin\Debu
***** Product List *****
ProductID: 1, ProductName: Genen Shouyu.
ProductID: 2, ProductName: Alice Mutton.
ProductID: 3, ProductName: Aniseed Syrup.
ProductID: 4, ProductName: Perth Pasties.
ProductID: 5, ProductName: Carnarvon Tigers.
ProductID: 6, ProductName: Gula Malacca.
ProductID: 7, ProductName: Steeleye Stout.
ProductID: 8, ProductName: Chocolade.
ProductID: 9, ProductName: Mishi Kobe Niku.
ProductID: 10, ProductName: Ikura.
```

Connection Objects

- Establish a session with the data source
- ConnectionString property: identify the name of the machine we wish to connect to, required security settings, the name of the database on that machine, and other data provider–specific information

Properties	ConnectionString, State
Methods	CreateCommand, Open, Close
Event	StateChange




```
string strConnect = "server=(local);database=pubs;uid=sa;pwd=sa";
SqlConnection con = new SqlConnection(strConnect);
```

ConnectionStringBuilder Objects

- The .NET compliant data providers support connection string builder objects, which allow us to establish the name-value pairs using strongly typed

```
var connectionStringBuilder = new SqlConnectionStringBuilder()
{
    InitialCatalog = "BookStore",
    DataSource = "(local)",
    UserID = "sa",
    Password = "P@ssw0rd",
    ConnectTimeout = 30
};
string ConnectionString = connectionStringBuilder.ConnectionString;
SqlConnection connection = new SqlConnection(ConnectionString);
```

Command Objects

- The SqlCommand,which derives from DbCommand, is an OO (Object-Oriented) representation of a SQL query, table name, or stored procedure
- Parameterized queries execute much faster than a literal SQL string, in that they are parsed exactly once. To associate a parameter within a SQL query to a member in the command object's parameters collection, prefix the SQL text parameter with an at @ symbol
- The type of command is specified using the CommandType property, which includes:
  - **StoredProcedure**: The name of a stored procedure
  - **TableDirect**: The name of a table
  - **Text**: An SQL text command (Default)

Command Objects

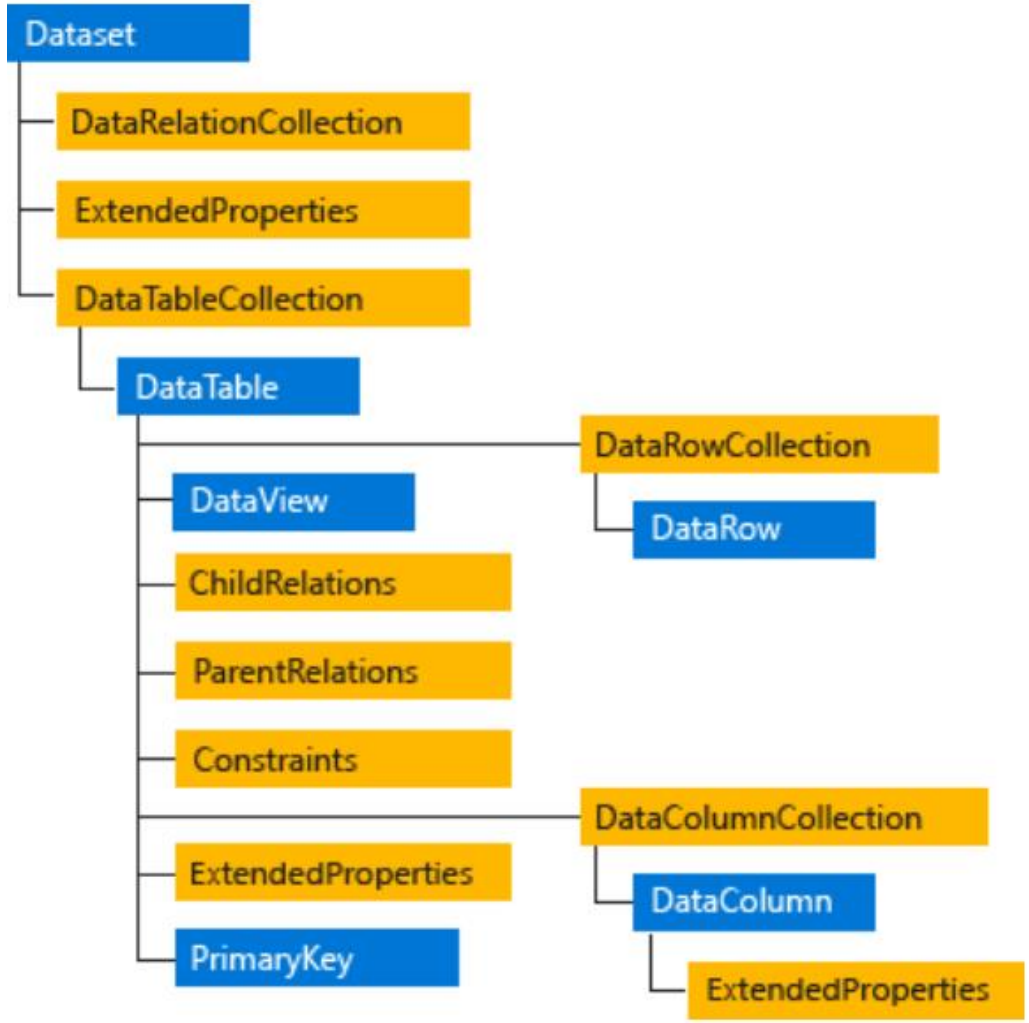
Member	Description
CommandTimeout	Gets or sets the time to wait while executing the command before terminating the attempt and generating an error. The default is 30 seconds
Connection	Gets or sets the DbConnection used by this instance of the DbCommand
Parameters	Gets the collection of DbParameter objects used for a parameterized query
Cancel()	Cancels the execution of a command
ExecuteReader()	Executes a SQL query and returns the data provider's DbDataReader object, which provides forward-only, read-only access for the result of the query
ExecuteNonQuery()	Executes a SQL nonquery (e.g., an insert, update, delete, or create table)
ExecuteScalar()	A lightweight version of the ExecuteReader() method that was designed specifically for singleton queries (e.g., obtaining a record count)
Prepare()	Creates a prepared (or compiled) version of the command on the data source. As we might know, a *prepared query* executes slightly faster and is useful when we need to execute the same query multiple times (typically with different parameters each time)

The DataSet

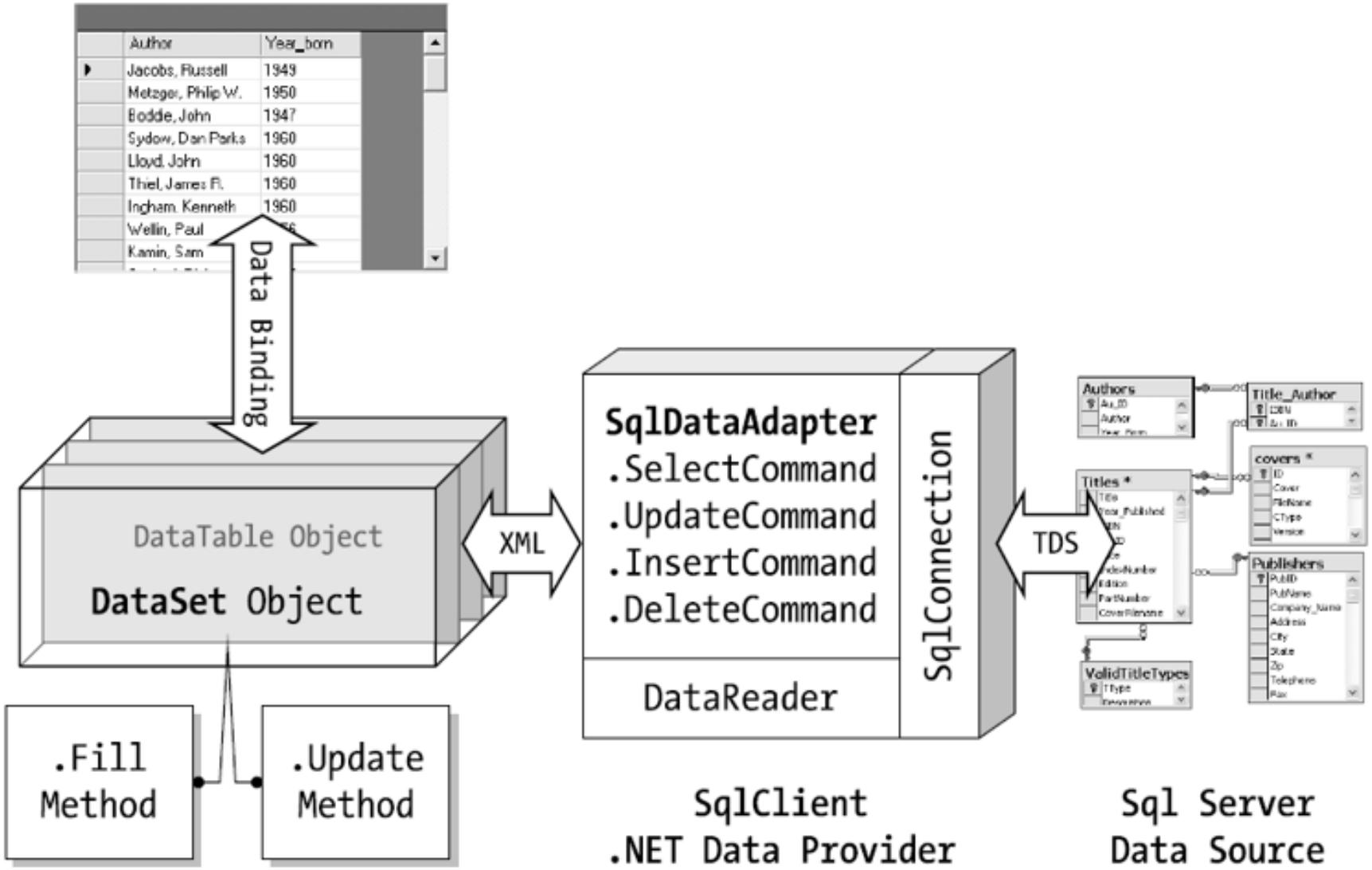
- The **ADO.NET DataSet** is explicitly designed for data access independent of any data source. As a result, it can be used with multiple and differing data sources, used with **XML** data, or used to manage data local to the application
- The **DataSet** object is central to supporting disconnected, 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

The DataSet

- The DataSet contains a collection of one or more **DataTable** objects consisting of rows and columns of data, and also *primary key*, *foreign key, constraint,* and *relation* information about the data in the **DataTable** objects



The DataSet

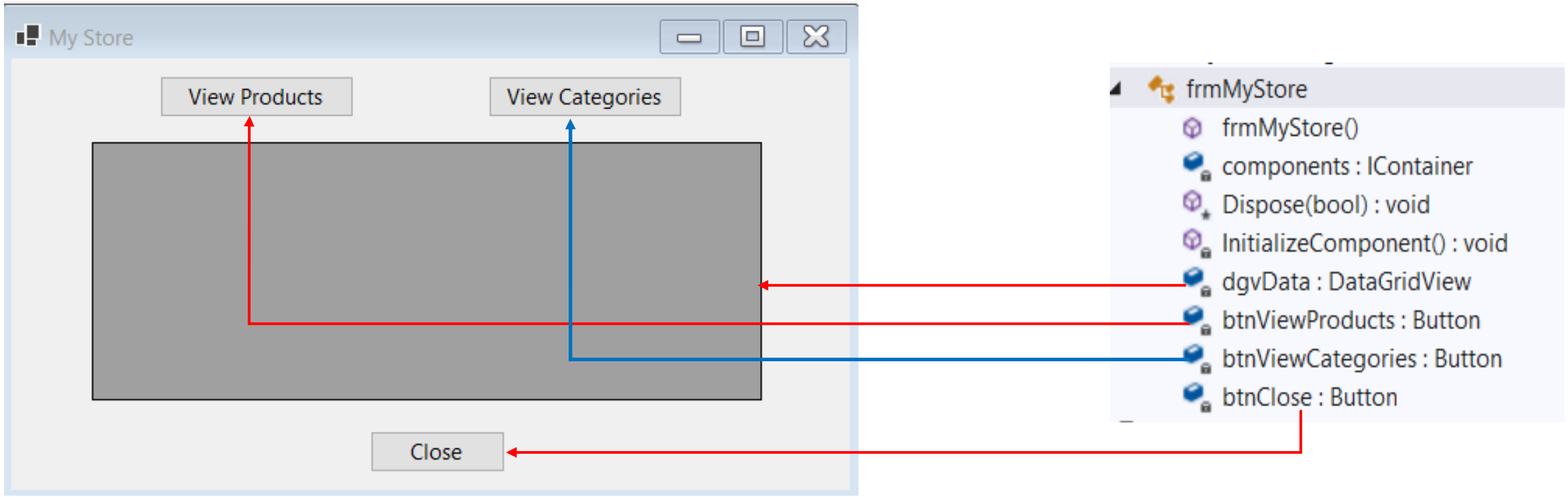

Author
Year_born
Jacobs, Russell
1949
Metzger, Philip W.
1950
Bodde, John
1947
Sydow, Dan Parks
1960
Lloyd, John
1960
Thiel, James R.
1960
Ingham, Kenneth
1960
Wellin, Paul
Kamin, Sam
Data Binding
DataTable Object
DataSet Object
XML
SqlDataAdapter
.SelectCommand
.UpdateCommand
.InsertCommand
.DeleteCommand
DataReader
SqlConnection
TDS
Authors
Title_Author
covers *
Titles *
Publishers
ValidTitleTypes
.Fill Method
.Update Method
SqlClient .NET Data Provider
Sql Server Data Source

DataAdapter Objects

- DataAdapter (which extends the abstract **DbDataAdapter**) is used to fetch and update data
- DataAdapter objects make use of **DataSet** objects to move data between the caller and data source
- DataAdapter objects move **DataSets** to and from the client tier
- The DataAdapter handles the database connection automatically and keeps the connection open for the shortest possible amount of time
- Once the caller receives the **DataSet** object, the connection is completely disconnected from the **DBMS** and left with a local copy of the remote data
- The caller is free to insert, delete, or update rows from a given **DataTable**, but the physical database is not updated until the caller explicitly passes the **DataSet** to the data adapter for updating

Disconnected Data Access Demonstration

1.Create a Winform app named **DemoDisconnectedLayer** includes a form named **frmMyStore** and has controls as follows :



2.Trigger **Click** event of the buttons: **btnClose**, **btnViewProducts**, and **btnViewCategories**

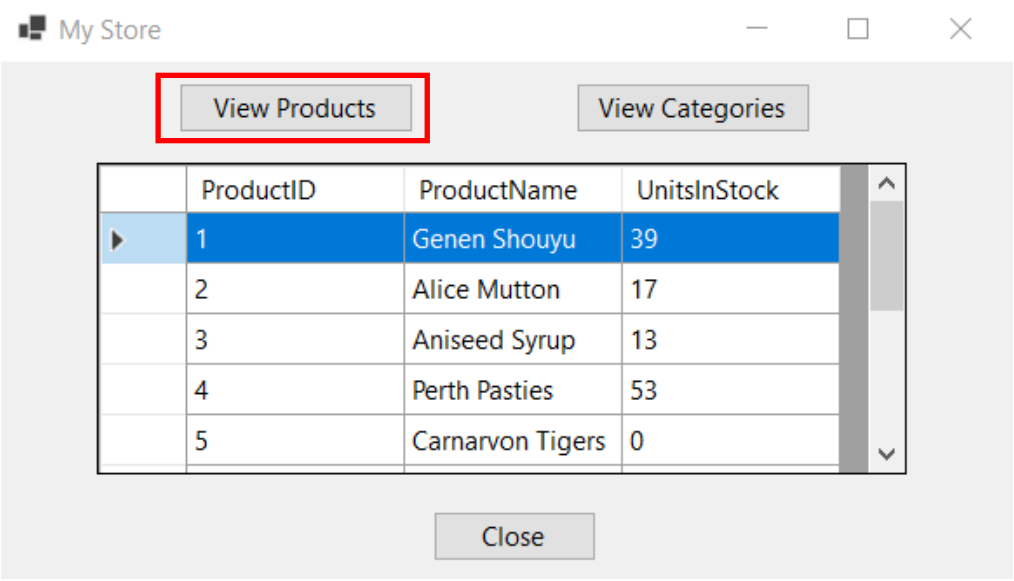
3.Trigger **Load** event of the **frmMyStore** form

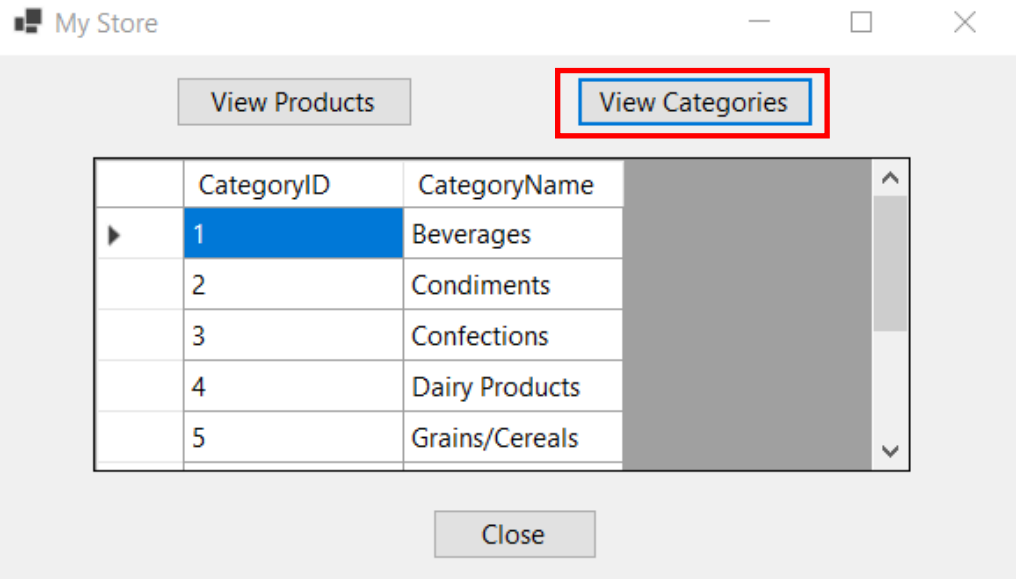
4. Install **Microsoft.Data.SqlClient** package from Nuget package

5. Write codes in **frmMyStore.cs** as follows then press **Ctrl+F5** to run project:

```
public partial class frmMyStore : Form{
    public frmMyStore()...
    //Create a Dataset to store data
    DataSet dsMyStore = new DataSet();
    private void frmMyStore_Load(object sender, EventArgs e){
        string ConnectionString = "Server=(local);uid=sa;pwd=123;database=MyStore";
        string SQL = "Select ProductID,ProductName,UnitsInStock From Products ; Select * From Categories";
        try {
            SqlDataAdapter dataAdapter = new SqlDataAdapter(SQL, ConnectionString);
            dataAdapter.Fill(dsMyStore);
        }
        catch (Exception ex){
            MessageBox.Show(ex.Message,"Get Data From Database");
        }
    }
```

```
    private void btnViewProducts_Click(object sender, EventArgs e){
        //Show Products table
        dgvData.DataSource = dsMyStore.Tables[0];
    }
    private void btnViewCategories_Click(object sender, EventArgs e){
        //Show Categories table
        dgvData.DataSource = dsMyStore.Tables[1];
    }
    private void btnClose_Click(object sender, EventArgs e) => this.Close();
}//end Form
```





DataTable Objects

- The DataTable class in ADO.NET is a database table representation and provides a collection of columns and rows to store data in a grid form

Properties	Description
Columns	Represents all table columns
Constraints	Represents all table constraints
DataSet	Returns the dataset for the table
DefaultView	Customized view of the data table
ChildRelation	Return child relations for the data table
ParentRelation	Returns parent relations for the data table
PrimaryKey	Represents an array of columns that function as primary key for the table
Rows	All rows of the data table

Method	Description
AcceptChanges	Commits all the changes made since last AcceptChanges was called
Clear	Deletes all data table data
Clone	Creates a clone of a DataTable including its schema
Copy	Copies a data table including its schema
NewRow	Creates a new row, which is later added by calling the Rows.Add method
RejectChanges	Reject all changed made after last AcceptChanges was called

DataView Objects

- Represents a databindable, customized view of a DataTable for sorting, filtering, searching, editing, and navigation
- The DataView does not store data, but instead represents a connected view of its corresponding DataTable

Properties	Description
RowFilter	Gets or sets the expression used to filter which rows are viewed in the DataView
Sort	Gets or sets the sort column or columns, and sort order for the DataView
Table	Gets or sets the source DataTable

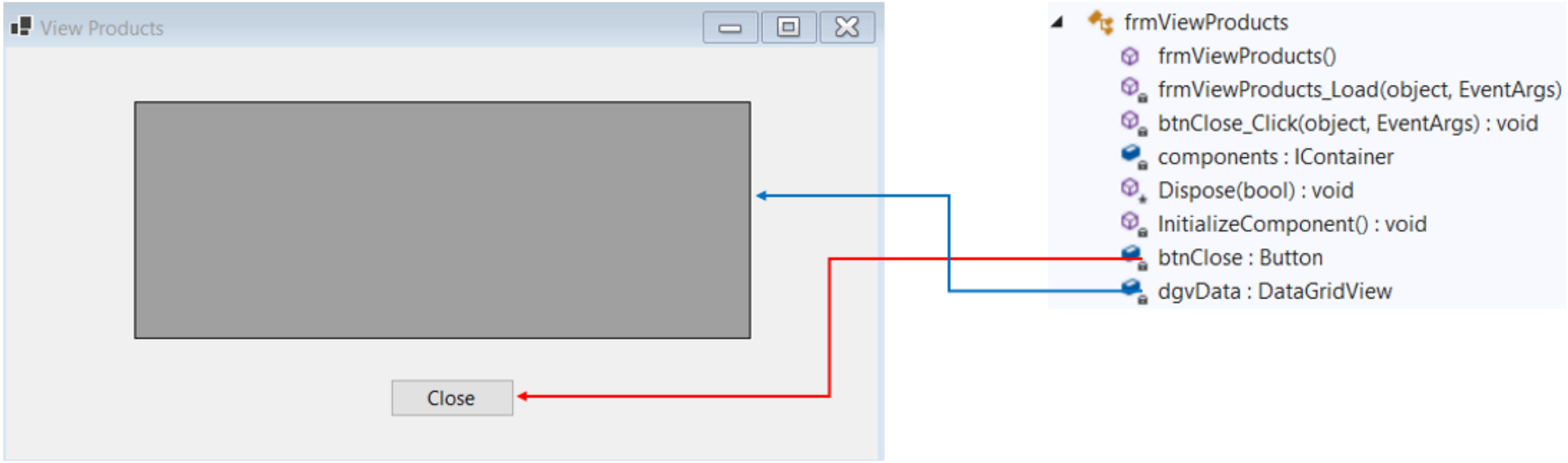
Methods	Description
Find(Object)	Finds a row in the DataView by the specified sort key value
FindRows(Object)	Returns an array of DataRowView objects whose columns match the specified sort key value

DataReaders Objects

- The DbDataReader type (which implements IDataReader) is the simplest and fastest way to obtain information from a data store. Recall that Data readers represent a read-only, forward-only stream of data returned one record at a time
- Increases the application performance. However, the DataReader object requires an exclusive use of an open connection object fot its whole life span
- Data readers are useful when we need to iterate over large amounts of data quickly and we do not need to maintain an in-memory representation

Connected Data Access Demonstration

1.Create a Winform app named **DemoConnectedLayer** includes a form named **frmViewProducts** and has controls as follows :

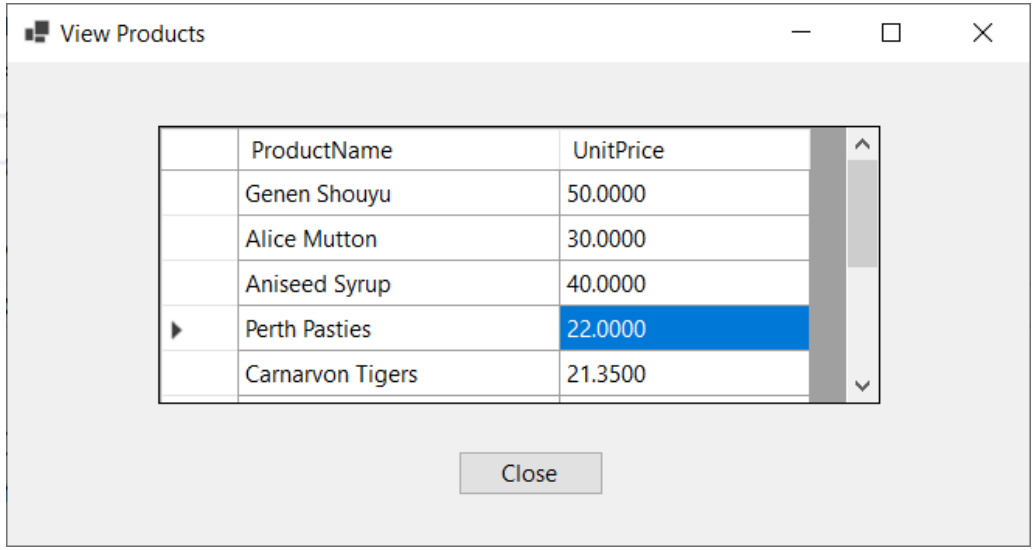


2.Trigger **Click** event of the **btnClose** button and **Load** event of **frmViewProducts** form

3. Install **Microsoft.Data.SqlClient** package from Nuget

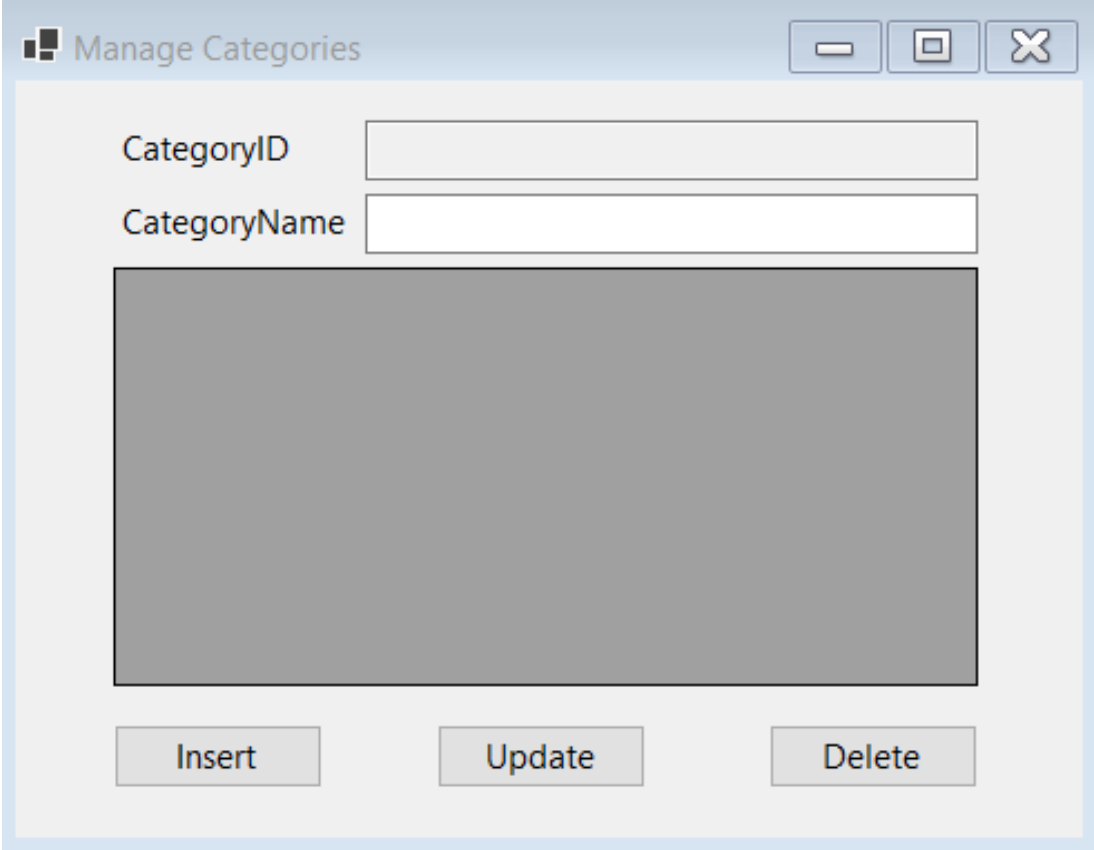
4. Write codes in **frmViewProducts.cs** as follows then press **Ctrl+F5** to run project:

```
private void frmViewProducts_Load(object sender, EventArgs e){
    //Create a list to store Products
    List<dynamic> products = new List<dynamic>();
    string ConnectionString = "server=(local);database=MyStore;uid=sa;pwd=123";
    SqlConnection connection = new SqlConnection(ConnectionString);
    SqlCommand command = new SqlCommand("Select ProductName, UnitPrice from Products",connection);
    connection.Open();
    SqlDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection);
    if (reader.HasRows == true){
        while (reader.Read()) {
            products.Add(new
            {
                ProductName = reader.GetString("ProductName"),
                UnitPrice = reader.GetDecimal("UnitPrice")
            });
        }//end while
        //Binding with DataGridView: dgvData
        dgvData.DataSource = products;
    }//end if
}//end frmViewProducts_Load
//---------------------------
private void btnClose_Click(object sender, EventArgs e)=>Close();
```



Create, Update, Delete and Queries Demonstration

1.Create a Winform app named **ManageCategoriesApp** includes a form named **frmManageCategories** and has controls as follows :



Object Type	Object name	Properties / Events
Label	lbCategoryID	Text: CategoryID
Label	lbCategoryName	Text: CategoryName
TextBox	txtCategoryID	ReadOnly: True
TextBox	txtCategoryName	
Button	btnInsert	Text: Insert Event Handler: Click
Button	btnUpdate	Text: Update Event Handler: Click
Button	btnDelete	Text: Delete Event Handler: Click
DataGridView	dgvCategories	ReadOnly: True SelectionMode:FullRowSelect
Form	frmManageCategories	StartPosition: CenterScreen Text: Manage Categories Event Handler: Load

2.Right-click on the project | **Add** | **Class**, named **ManageCategories.cs** then write codes as follows:

```
//Declaring record Category
public record Category
{
    public int CategoryID { get; set; }
    public string CategoryName { get; set; }
}
//-----------------------------------------
public class ManageCategories{
    SqlConnection connection;
    SqlCommand command;
    string ConnectionString = "Server=(local);uid=sa;pwd=123;database=MyStore";
    public List<Category> GetCategories() {
        List<Category> categories = new List<Category>();
        connection = new SqlConnection(ConnectionString);
        string SQL = "Select CategoryID, CategoryName from Categories";
        command = new SqlCommand(SQL, connection);
        try{
            connection.Open();
            SqlDataReader reader = command.ExecuteReader(CommandBehavior.CloseConnection);
```

```
        if (reader.HasRows == true){
            while (reader.Read()){
                categories.Add(new Category{
                    CategoryID = reader.GetInt32("CategoryID"),
                    CategoryName = reader.GetString("CategoryName")
                });
            }//end while
        }//end if
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
    finally
    {
        connection.Close();
    }
    return categories;
}//end GetCategories
```

```
//-----------------------------------------
public void InsertCategory(Category category){
    connection = new SqlConnection(ConnectionString);
    //CategoryID is auto increment
    command = new SqlCommand("Insert Categories values(@CategoryName)", connection);
    command.Parameters.Add("@CategoryName", SqlDbType.NVarChar).Value
        = category.CategoryName;
    try {
        connection.Open();
        command.ExecuteNonQuery();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
    finally{
        connection.Close();
    }
}//end InsertCategory
```

```
//-----------------------------------------
public void UpdateCategory(Category category){
    connection = new SqlConnection(ConnectionString);
    string SQL = "Update Categories set CategoryName=@CategoryName where CategoryID=@CategoryID";
    command = new SqlCommand(SQL, connection);
    command.Parameters.AddWithValue("@CategoryID",category.CategoryID);
    command.Parameters.AddWithValue("@CategoryName",category.CategoryName);
    try {
        connection.Open();
        command.ExecuteNonQuery();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
    finally{
        connection.Close();
    }
}//end UpdateCategory
```

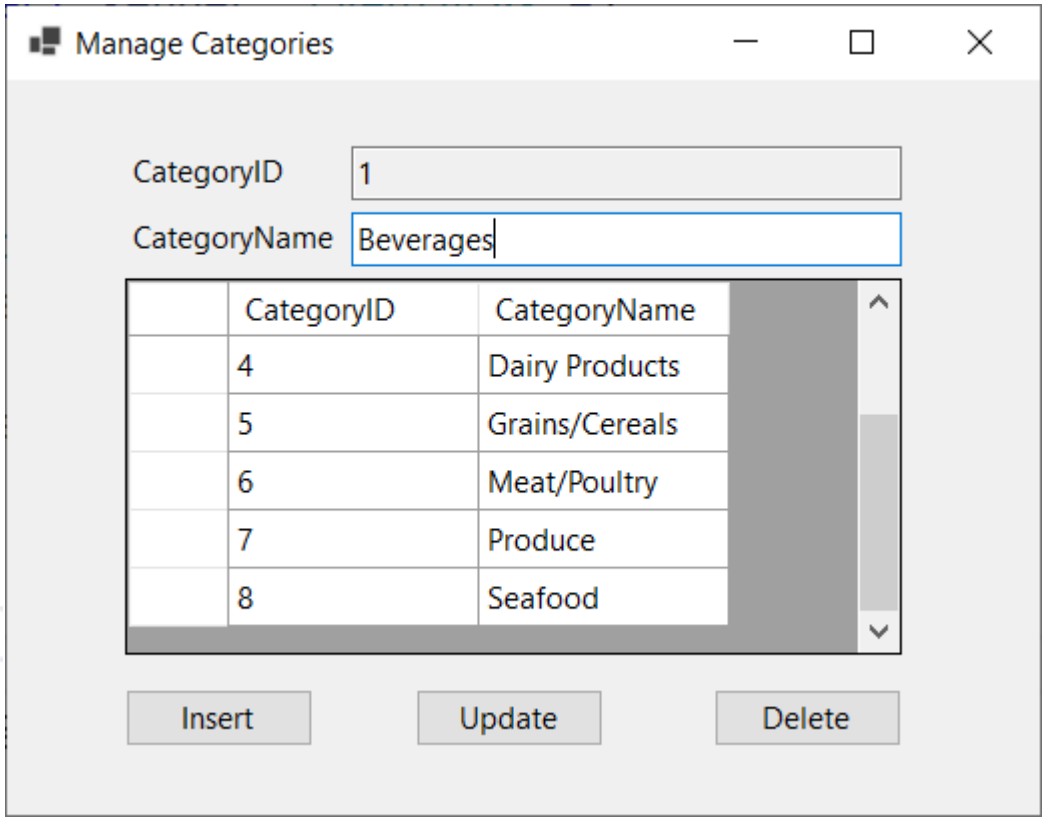
```
    //-----------------------------------------
    public void DeleteCategory(Category category) {
        connection = new SqlConnection(ConnectionString);
        string SQL = "Delete Categories where CategoryID=@CategoryID";
        command = new SqlCommand(SQL, connection);
        command.Parameters.AddWithValue("@CategoryID", category.CategoryID);
        try {
            connection.Open();
            command.ExecuteNonQuery();
        }
        catch (Exception ex){
            throw new Exception(ex.Message);
        }
        finally{
            connection.Close();
        }
    }//end DeleteCategory
}//end ManageCategories
```

3.Write codes in **frmManageCategories.cs** as follows then press **Ctrl+F5** to run project:

```
public partial class frmManageCategories : Form {
    public frmManageCategories()...
    ManageCategories manageCategories = new ManageCategories();
    private void LoadCategories() {
        var categories = manageCategories.GetCategories();
        txtCategoryID.DataBindings.Clear();
        txtCategoryName.DataBindings.Clear();
        //Binding to TextBoxes
        txtCategoryID.DataBindings.Add("Text", categories, "CategoryID");
        txtCategoryName.DataBindings.Add("Text", categories, "CategoryName");
        //Binding to DataGridView
        dgvCategories.DataSource = categories;
    }
    //-----------------------------------------------------------------------------
    private void frmManageCategories_Load(object sender, EventArgs e) => LoadCategories();
    //-----------------------------------------------------------------------------
```

```
private void btnInsert_Click(object sender, EventArgs e){
    try {
        var category = new Category { CategoryName = txtCategoryName.Text };
        manageCategories.InsertCategory(category);
        LoadCategories();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message, "Insert Category");
    }
}
//-----------------------------------------------------------------------------
private void btnUpdate_Click(object sender, EventArgs e) {
    try{
        var category = new Category{
            CategoryID = int.Parse(txtCategoryID.Text),
            CategoryName = txtCategoryName.Text
        };
        manageCategories.UpdateCategory(category);
        LoadCategories();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message,"Update Category");
    }
}
//-----------------------------------------------------------------------------
```

```
    private void btnDelete_Click(object sender, EventArgs e)
    {
        try
        {
            var category = new Category{
                CategoryID = int.Parse(txtCategoryID.Text)
            };
            manageCategories.DeleteCategory(category);
            LoadCategories();
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message, "Delete Category");
        }
    }
}//end class
```



Working with Store Procedures

- Stored procedures stored in the database which are a key ingredient in any successful large-scale database applications
- One advantage of stored procedures is improved performance. Stored procedures typically execute faster than ordinary SQL statements because the database can create, optimize, and cache a data access plan in advance

```
CREATE [ OR ALTER ] { PROC | PROCEDURE } [schema_name.]
procedure_name [ ; number ] [ { @parameter [ type_schema_name. ]
data_type } [ VARYING ] [ = default ] [ OUT | OUTPUT | [READONLY]
] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
 AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] } [;]
```

Benefits of Store Procedures

- Improve security. A client can be granted permissions to execute a stored procedure to add or modify a record in a specify way, without having full permissions on the underlying tables
- Are easy to maintain, because they are stored separately from the application code. Thus, we can modify a stored procedure without recompiling and redistributing the .NET application that uses it
- Add an extra layer of indirection, potentially allowing some database details to change without breaking your code. For example, a stored procedure can remap field names to match the expectations of the client program
- Reduce network traffic, because SQL statements can be executed in batches

Store Procedures Demonstration

- Create store procedures to count Products by CategoryID

```
Use MyStore
GO
Create Proc spCountProductsUsingOutputValue(@CategoryID int,@NumberOfProducts int Output)  As
      Select  @NumberOfProducts = Count(ProductID)
      From Products
      where CategoryID = @CategoryID
      Group by CategoryID
GO
Create Proc spCountProductsUsingReturnValue(@CategoryID int)  As
      Declare  @NumberOfProducts int
      Select  @NumberOfProducts = Count(ProductID)
      From Products
      where CategoryID = @CategoryID
      Group by CategoryID
      Return  @NumberOfProducts
```

- Create Console App then write codes as follows:

```
class Program
{
    static (int OutputValue, int ReturnValue) CountProductsByCategoryID(int CategoryID)
    {
        //Declare a Tuple
        (int OutputValue  , int ReturnValue) result;
        string ConnectionString = "Server=(local);uid=sa;pwd=123;database=MyStore";
        SqlConnection connection = new SqlConnection(ConnectionString);
        SqlCommand command = new SqlCommand();
        command.Connection = connection;
        command.CommandType = CommandType.StoredProcedure;
        connection.Open();
        CountProductsUsingOutputValue();
        CountProductsUsingReturnValue();
        //Declare Local Functions
        void CountProductsUsingOutputValue() {
            command.CommandText = "spCountProductsUsingOutputValue";
            command.Parameters.AddWithValue("@CategoryID", CategoryID);
            //Using Output value
            command.Parameters.Add("@NumberOfProducts", SqlDbType.Int).
                Direction = ParameterDirection.Output;
            command.ExecuteNonQuery();
            result.OutputValue = (int)command.Parameters["@NumberOfProducts"].Value;
        }//end function
```

```
        void CountProductsUsingReturnValue(){
            command.CommandText = "spCountProductsUsingReturnValue";
            //Using Return value
            command.Parameters["@NumberOfProducts"].Direction = ParameterDirection.ReturnValue;
            command.ExecuteNonQuery();
            result.ReturnValue = (int)command.Parameters["@NumberOfProducts"].Value;
        }//end function
        connection.Close();
        return result;
    }//end CountProductsByCategoryID
    static void Main(string[] args)
    {
        int CategoryID = 1;
        var result = CountProductsByCategoryID(CategoryID);
        Console.WriteLine($"Number of products by CategoryID:{CategoryID}");
        Console.WriteLine($"-->OutputValue: {result.OutputValue}, ReturnValue: {result.ReturnValue}");
        CategoryID = 3;
        Console.WriteLine($"Number of products by CategoryID:{CategoryID}");
        result = CountProductsByCategoryID(CategoryID);
        Console.WriteLine($"-->OutputValue: {result.OutputValue}, ReturnValue: {result.ReturnValue}");
        Console.ReadLine();
    }//end Main
}//end Program
```

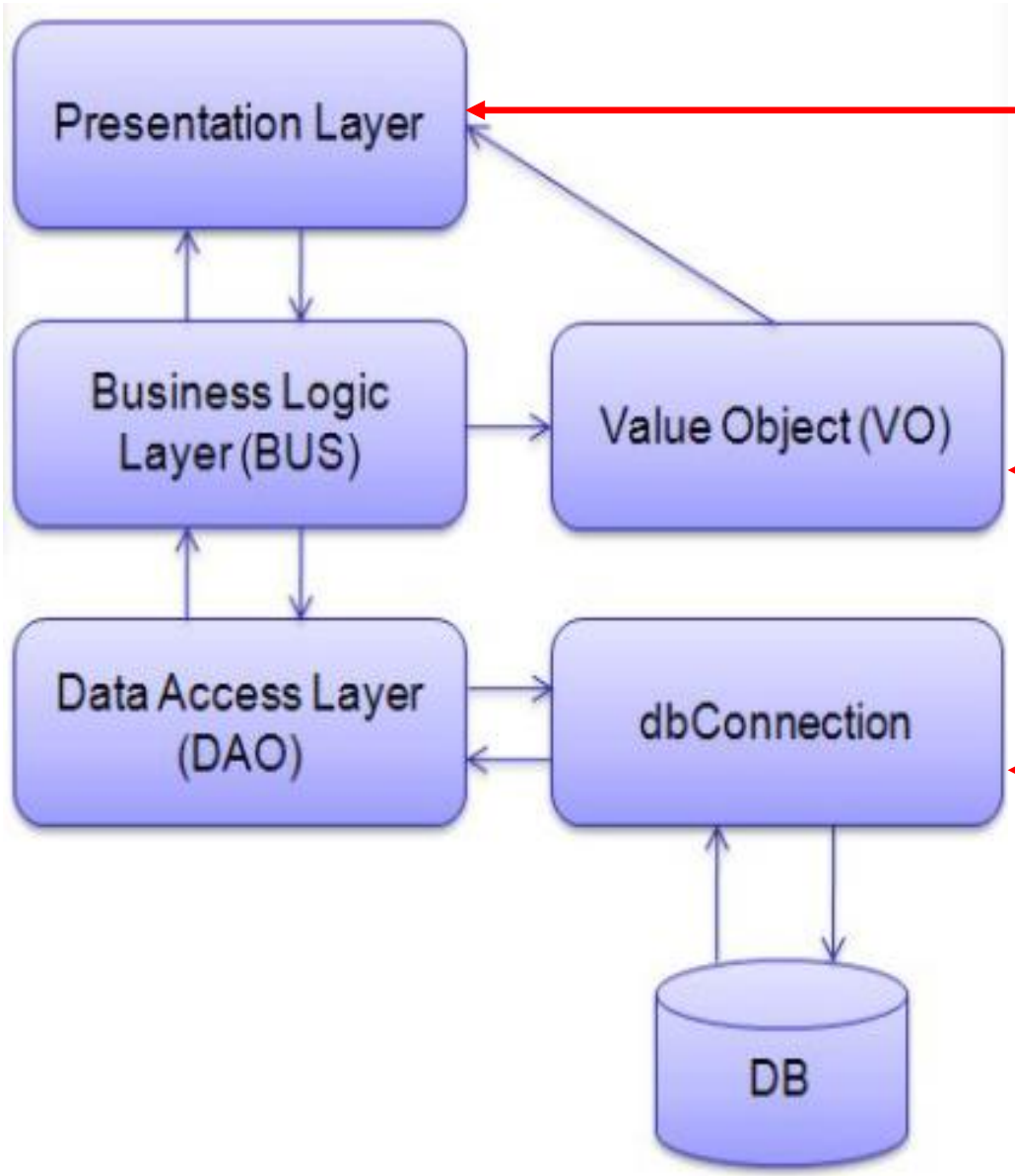
```
D:\Demo\FU\Basic.NET\Slot_17_18_ADO.NET\DemoUsingStore
Number of products by CategoryID:1
-->OutputValue: 2, ReturnValue: 2
Number of products by CategoryID:3
-->OutputValue: 1, ReturnValue: 1
```

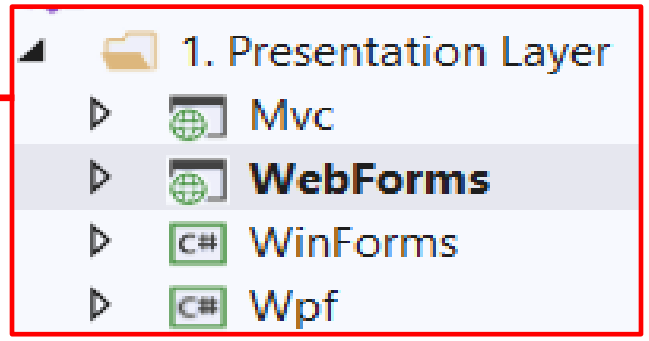
What is .NET 3-Layers Architecture?

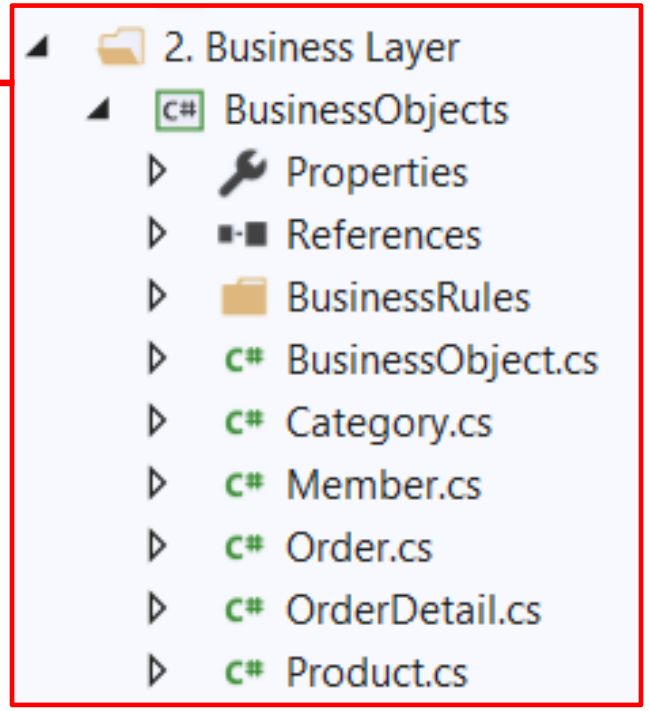
- Three-layer architecture is dividing the project into three layers that are **User interface layer**, **Business layer** and **Data (database) layer** where we separate UI, Logic, and Data in three divisions

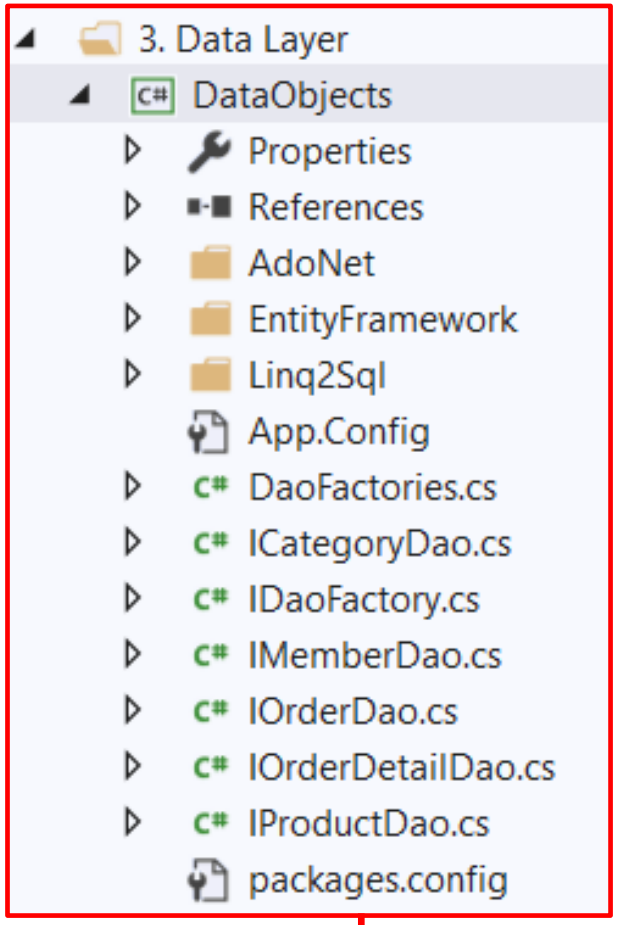
- Suppose we want to change the UI from windows to the phone than he has to only make change in UI layer, other layers are not affected by this change Similarly, if the we want to change the database then we have to only make a change in the data layer, rest everything remains the same

The .NET 3-Layers Architecture





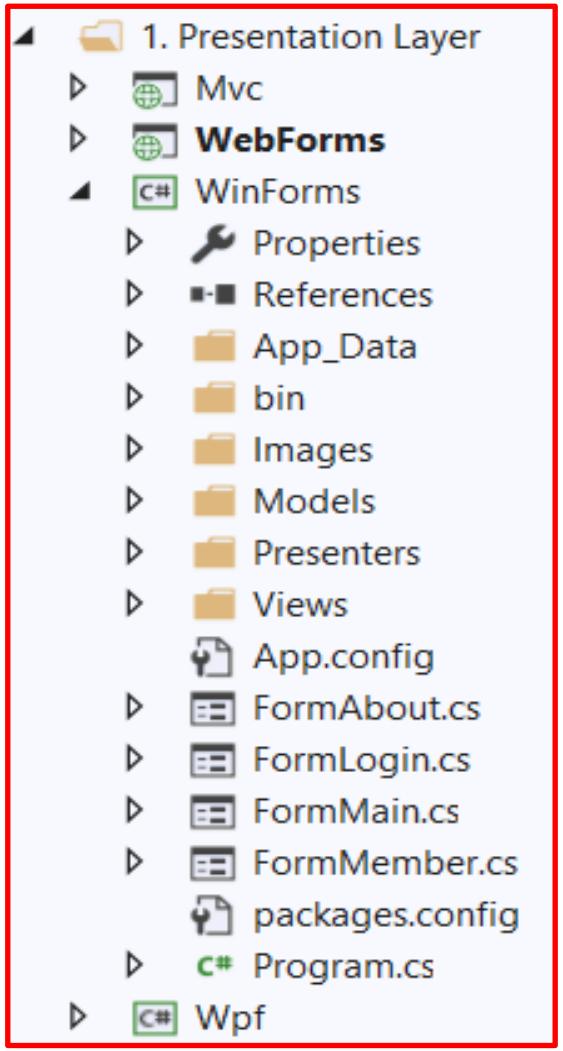




What is .NET 3-Layers Architecture?

Presentation Layer

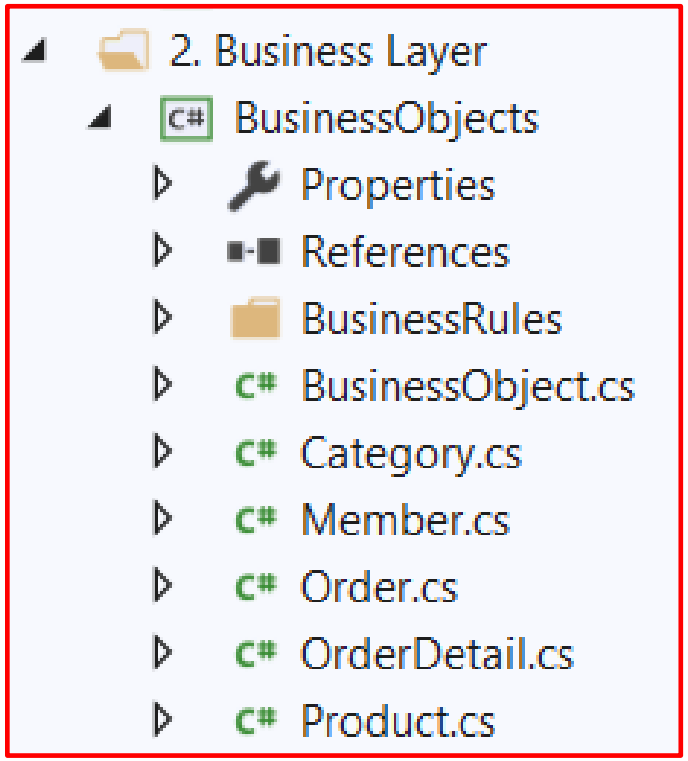
- This is the top layer of architecture. The topmost level of application is the user interface
- It is related to the user interface that is what the user sees. The main function of this layer is to translate tasks and results in something which the user can understand
- It contains pages like web forms, windows form where data is presented to the user and use to take input from the user. The presentation layer is the most important layer because it is the one that the user sees and good UI attracts the user and this layer should be designed properly



What is .NET 3-Layers Architecture?

Business Layer

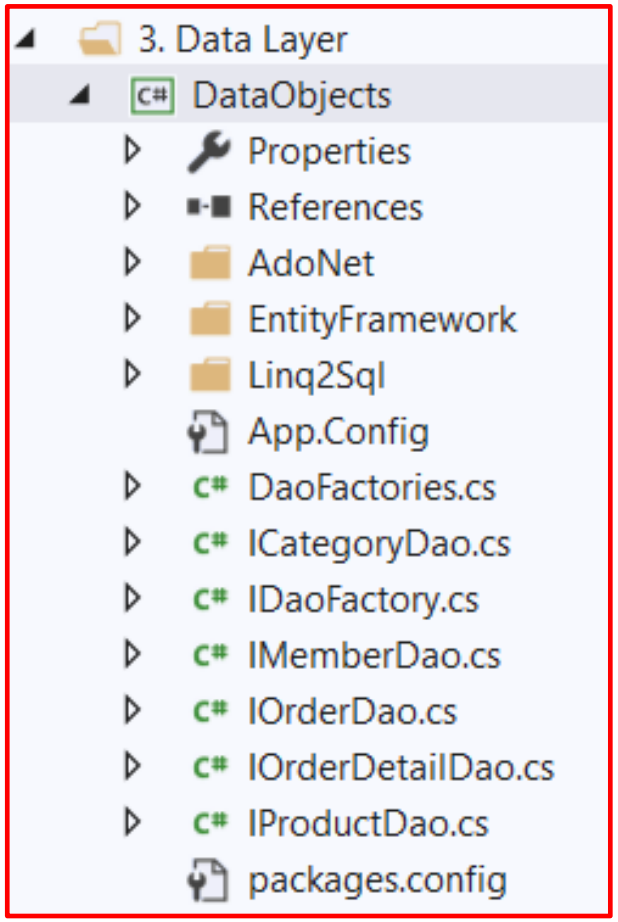
- This is the middle layer of architecture. This layer involves C# classes and logical calculations and operations are performed under this layer
- It processes the command, makes logical decisions and perform calculations. It also acts as a middleware between two surrounded layers that is presentation and data layer
- It processes data between these two layers. This layer implements business logic and calculations and validates the input conditions before calling a method from the data layer. This ensures the data input is correct before proceeding, and can often ensure that the outputs are correct as well. This validation of input is called business rules



What is .NET 3-Layers Architecture?

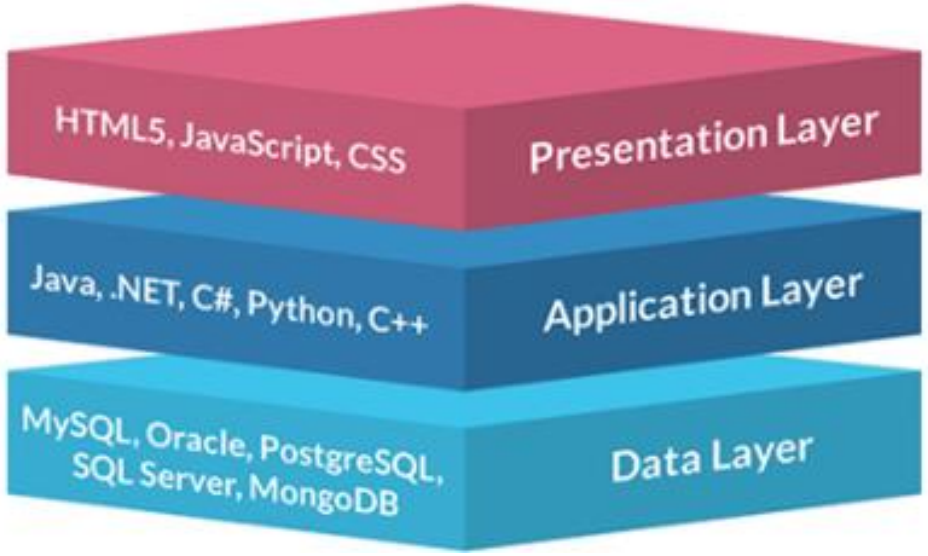
Data Layer

- This layer is used to connect the business layer to the database or data source
- It contains methods which are used to perform operations on database like insert, delete, update, etc
- This layer contains stored procedures which are used to query database. Hence this layer establishes a connection with the database and performs functions on the database



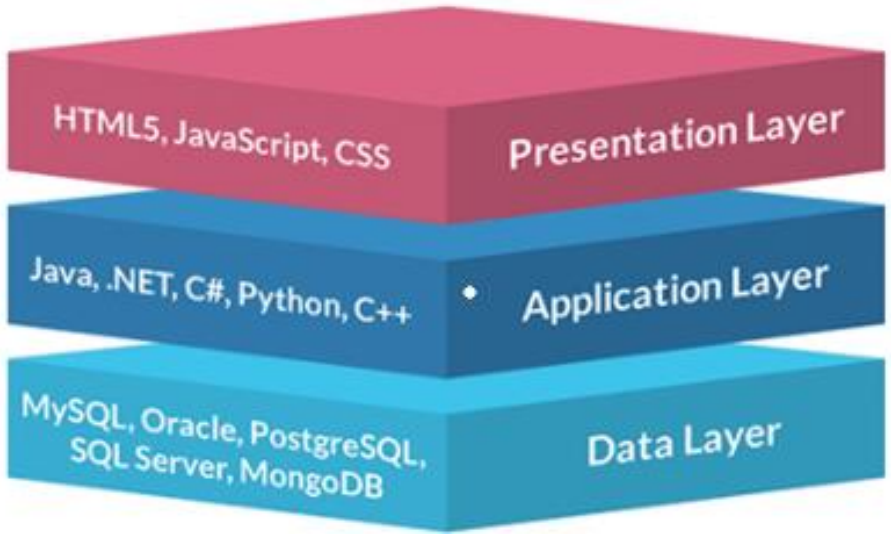
What is 3-Tiers Architecture?

- A 3-tier application architecture is a modular client-server architecture that consists of **a Presentation tier**, **an Application tier** and **a Data tier**
- The data tier stores information, the application tier handles logic and the presentation tier is a graphical user interface (GUI) that communicates with the other two tiers. The three tiers are logical, not physical, and may or may not run on the same physical server



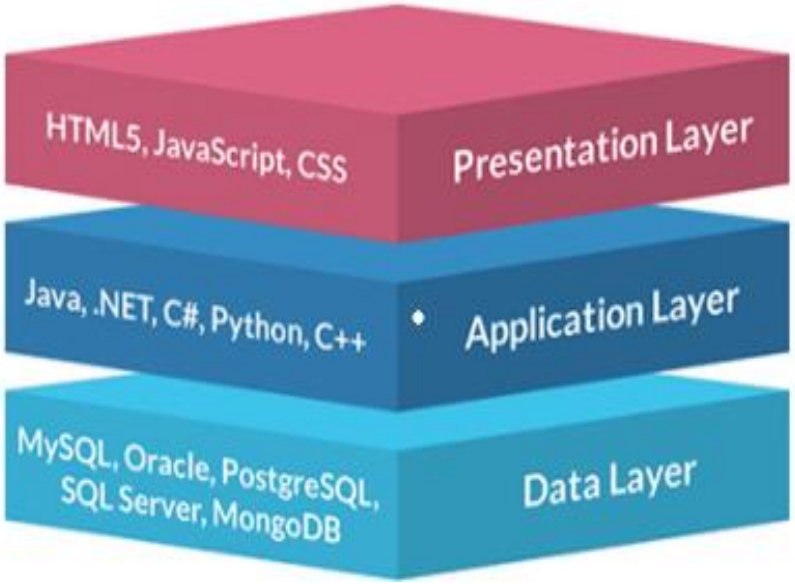
What is 3-Tiers Architecture?

- **Presentation Tier**
  - The presentation tier is the front end layer in the 3-tier system and consists of the user interface
  - This user interface is often a graphical one accessible through a web browser or web-based application and which displays content and information useful to an end user
  - This tier is often built on web technologies such as HTML5, JavaScript, CSS, or through other popular web development frameworks, and communicates with others layers through API calls



What is 3-Tiers Architecture?

- **Application Tier:** The application tier contains the functional business logic which drives an application's core capabilities. It's often written in Java, .NET, C#, Python, C++, etc
- **Data Tier:** The data tier comprises of the database/data storage system and data access layer. Examples of such systems are MySQL, Oracle, PostgreSQL, Microsoft SQL Server, MongoDB, etc. Data is accessed by the application layer via API calls



Summary

- Concepts were introduced:
  - ADO.NET is a data access technology – supports disconnected data architecture
  - A data provider establishes and maintains connection to the database. The .NET Framework provides various data providers which are used for SQL Server, OLE DB, ODBC
  - .NET Data Providers and Dataset are used for accessing data source and then storing the retrieved records into tables : Connection, Command, DataAdapter, DataReader, DataTable, DataView
  - Overview about 3-Layers and 3-Tiers Architecture
  - List the benefits of ADO.NET
  - Demo using ADO.NET Data Provider Factory Model
  - Demo accessing database in WinForm Application using ADO.NET
  - Demo using Store procedures in ADO.NET
  - Overview about 3-Layers and 3-Tiers Architecture