Introduction to ADO.NET

Introduction

- ADO.NET is a set of classes (a framework) to interact with data sources such as databases, XML files, text files or Excel files etc.
- ADO is the acronym for ActiveX Data Objects.
- It allows us to connect to underlying data or databases.
- It has classes and methods to retrieve and manipulate data.

Connection Architectures

- There are following two types of connection architectures:
  - **Connected Architecture:**
    - The application remains connected with the database throughout the processing.
  - **Disconnected Architecture:**
    - The application automatically connects/disconnects during the processing.
    - The application uses temporary data on the application side called a DataSet.

Connected Architecture

- A connected environment is one in which an application is constantly connected to a data source.

- Advantages of connected environment:
  - Data concurrency issues are easily controlled.
    - At a time only one user can update the data in database.
  - Data is always updated.

- Disadvantages of connected environment:
  - A constant network connection is required that may lead to network traffic logging.
  - Scalability and performance issues in application.
    - If more than one user connected to database, query processing may be slow.

Disconnected Architecture

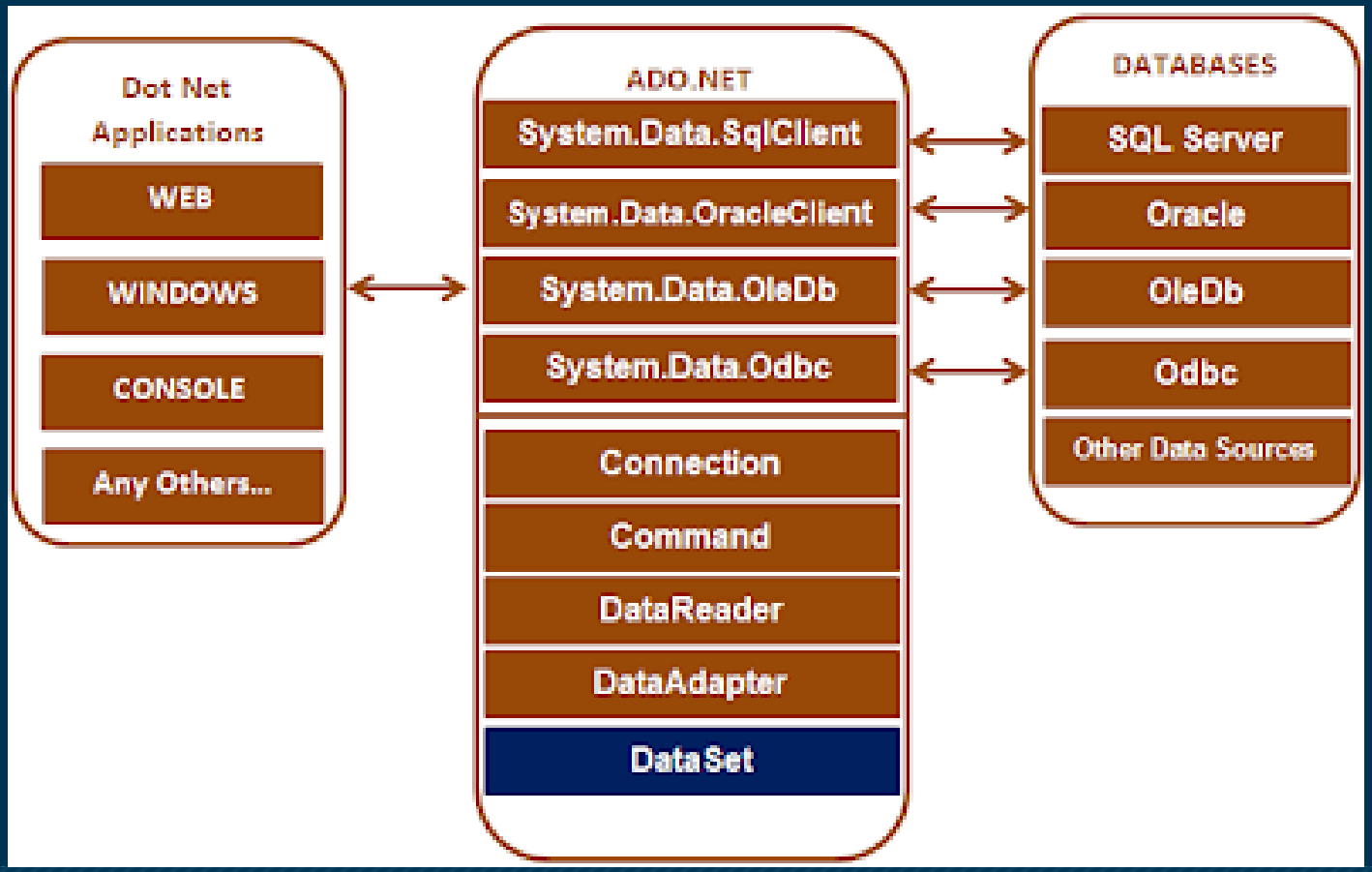
- A disconnected environment is one in which a user is not necessarily connected with a database.
- Connection is required only at the time of retrieval or update and after that connection is closed.

- Advantages of disconnected environment:
  - Multiple applications can simultaneously interact with the database.
  - Improve scalability and performance of applications.

- Disadvantages of disconnected environment:
  - Data is not always updated.
  - Data concurrency issues can occur when multiple users are updating the data to the data source.

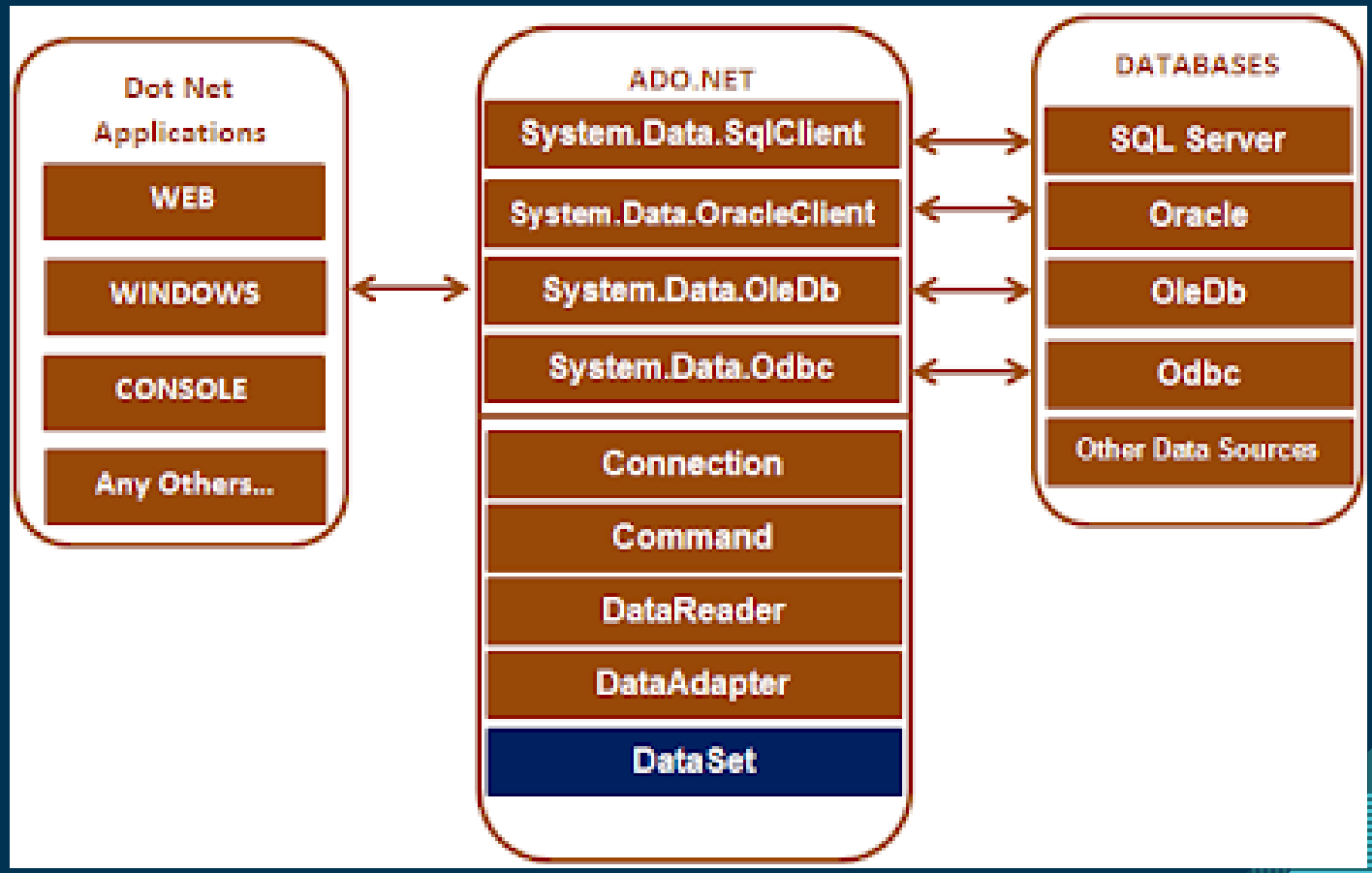
ADO.NET's Class Library

- In this diagram, we can see that there are various types of applications (Web Application, Console Application, Windows Application and so on) that use ADO.NET to connect to databases (SQL Server, Oracle, OleDb, ODBC, XML files and so on).



Important Classes in ADO.NET

- We can also observe various classes in the preceding diagram.
- They are:
  - Connection class
  - Command class
  - DataReader class
  - DataAdapter class
  - DataSet class



Connection Class

- ADO.NET connection is an object that provides database connectivity and the entry point to a database.
- When the connection of an object is instantiated, the constructor takes a connection string that contains the information about the database server, server type, database name, connection type, and database user credentials.
- Once the connection string is passed and the connection object is created, you can establish a connection with the database.
- A connection string is usually stored in the configuration file of an application.

What Namespace or Provider is used for Connection Class?

- ADO.NET provides connection to multiple providers.
- Each provider has a functionality to connect with different database.
- Here is a list of data providers in ADO.NET and their purpose:
  - Data Provider for SQL Server - `System.Data.SqlClient`
  - Data Provider for Oracle - `System.Data.OracleClient`
  - Data Provider for MS Access - `System.Data.OleDb`
  - Data Provider for MySQL - `System.Data.Odbc`

- How to use connection class with these providers:
  - Connection object for SQL Server - `SqlConnection`
  - Connection object for Oracle - `OracleConnection`
  - Connection object for MS Access - `OleDbConnection`
  - Connection object for MySQL - `OdbcConnection`

Command Class

- The Command class provides methods for storing and executing SQL statements and Stored Procedures.
- The following are the various commands that are executed by the Command class:
- `ExecuteReader`:
  - Returns data to the client as rows.
  - This would typically be an SQL SELECT statement or a Stored Procedure that contains one or more select statements.
  - This method returns a `DataReader` object that can be used to fill a `DataTable` object or used directly for printing reports and so forth.
- `ExecuteNonQuery`:
  - Executes a command that changes the data in the database, such as an UPDATE, DELETE, or INSERT statement, or a Stored Procedure that contains one or more of these statements.
  - This method returns an integer that is the number of rows affected by the query.
- `ExecuteScalar`:
  - This method only returns a single value.
  - This kind of query returns a count of rows or a calculated value.

DataReader Class

- The DataReader is used to retrieve data.
- It is used in conjunction with the Command class to execute an SQL SELECT statement and then access the returned rows.
- DataReader object provides a read only, forward only, high performance mechanism to retrieve data from a data source as a data stream, while staying connected with the data source.

DataAdapter Class

- This class is used in the Disconnected Model of ADO.NET.
- The purpose of the DataAdapter is embedded in its name – it performs the activities necessary to get the data from the data source on the server into the database that's held in the `DataSet`.
- To do that, the DataAdapter lets us specify the commands that should be carried out to retrieve and update data.
- DataAdapter had two very useful methods (discussed later):
  - `Fill`: Fills the `DataSet` with data from the database.
  - `Update`: Updates the database with data from the `DataSet`.

DataSet Class

- The `DataSet` is also used in the Disconnected Model.
- The `DataSet` is essentially a collection of `DataTable` objects.
- Each `DataTable` object contains a collection of `DataColumn` and `DataRow` objects.
- The `DataSet` also contains a `Relations` collection that can be used to define relations among `DataTable` objects.
- In other words, `DataSet` could look like a copy of the database.

Database Setup

Getting the database ready

Database Server

- We have three different ways to work with the database:

1. LocalDB in Visual Studio
2. Stand-alone SQL Server
3. Database file (.MDF) in Visual Studio

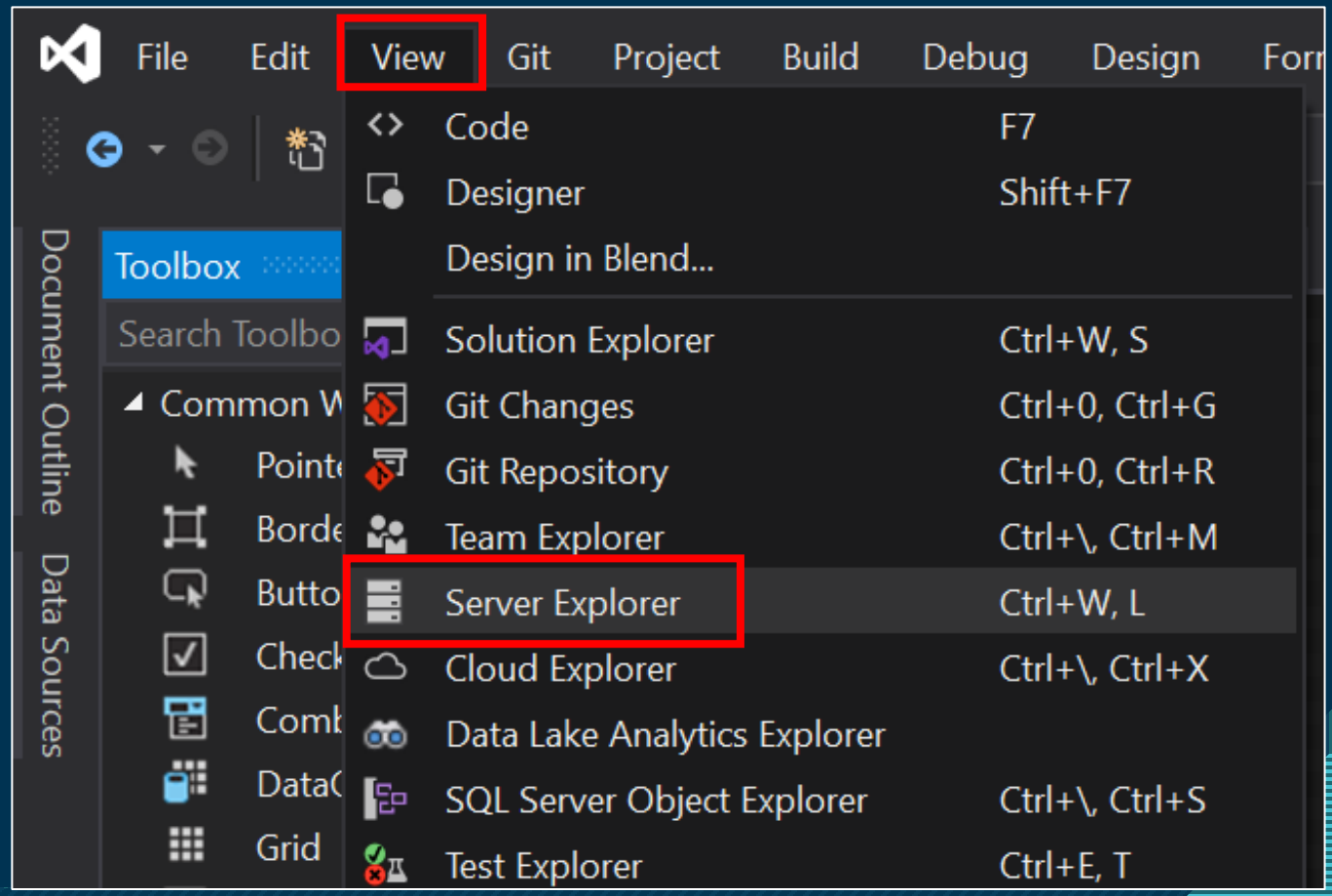
- We'll see the first two ways in this slide, but you can choose any method that suits you or pick a method preferred by your professor.
- We can work with any type of database in .NET.
- However, we'll be using the Visual Studio's built-in SQL Server.

Download Sample Database

- Go to SLATE → Content → Module 5 - Database Programming → Sample Databases.
- Download Northwind-Sample-Database-for-LocalDB file.
- Open this file with any text editor such as Notepad.

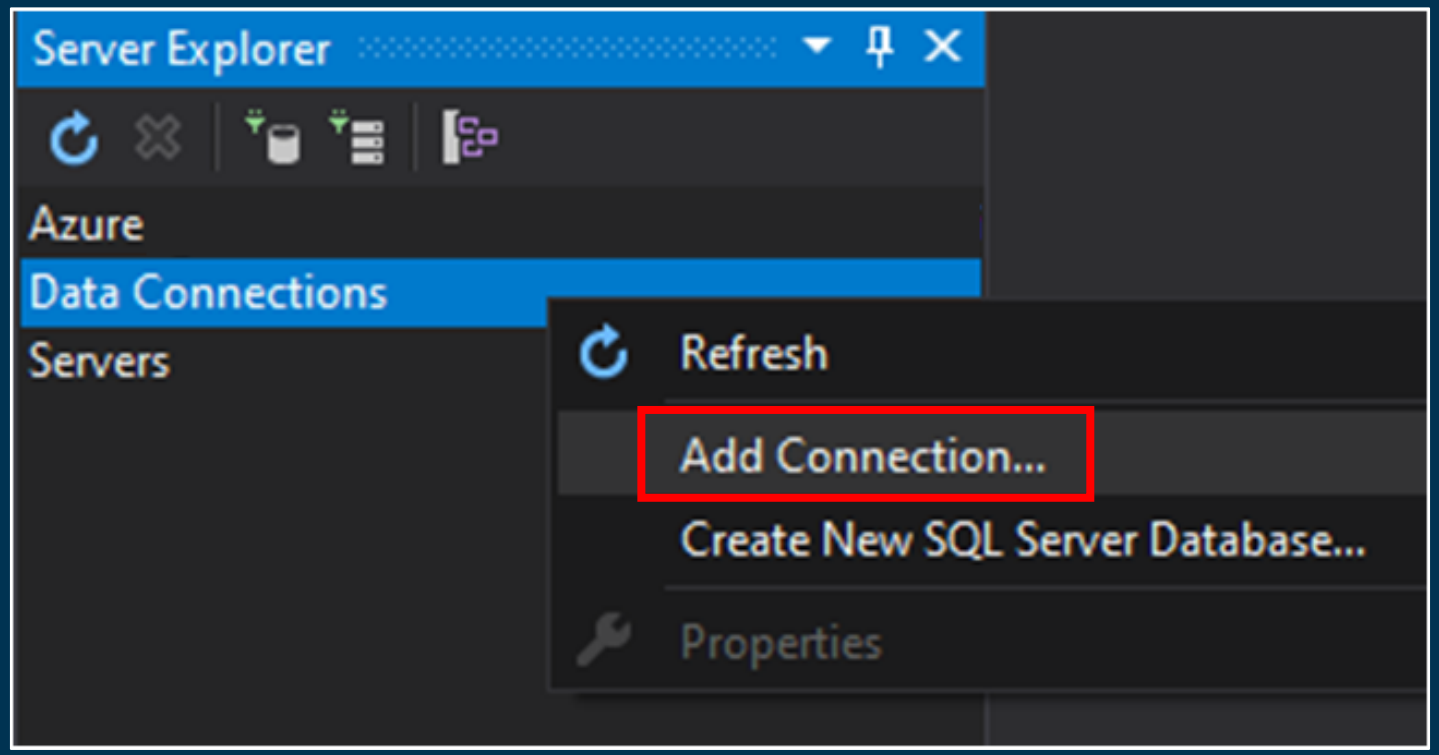
1. LocalDB in Visual Studio

- Create a new WPF App in Visual Studio and name it IntroToDatabaseProgramming.
- Open Server Explorer from the View menu.
  - You can pin the Server Explorer.



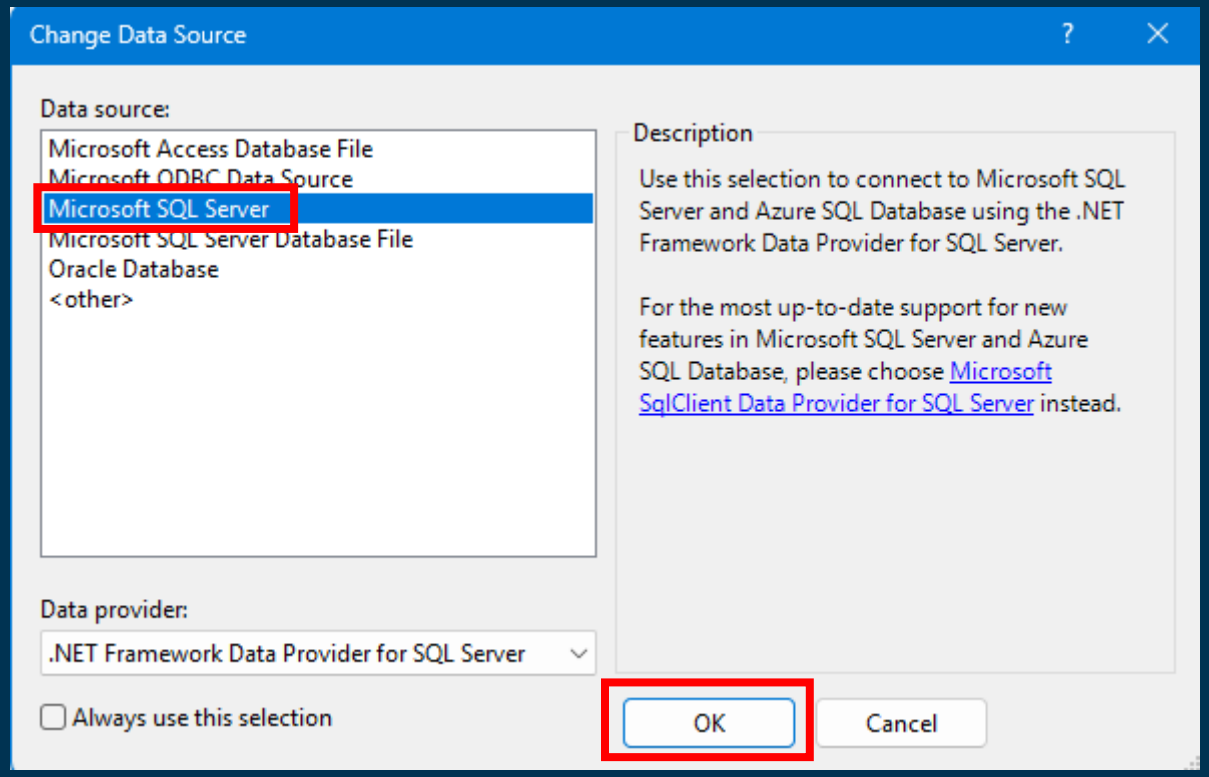
1. LocalDB in Visual Studio

- To work with LocalDB, in the Server Explorer, right-click on Data Connections and select Add Connection….



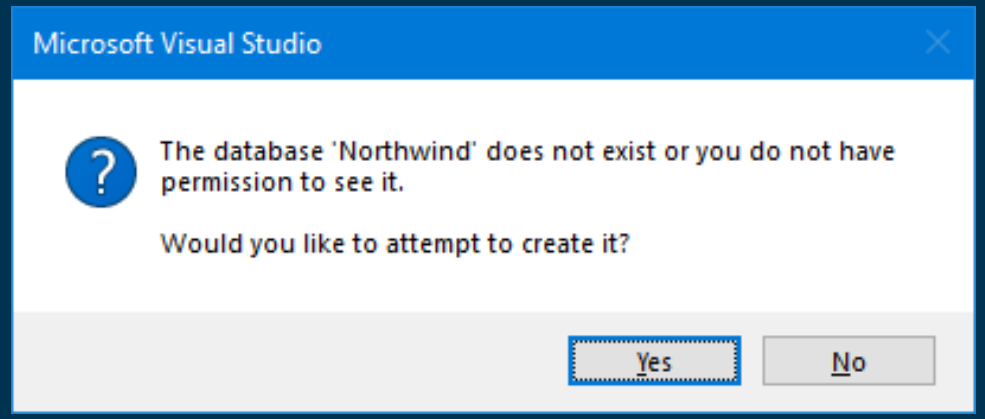
1. LocalDB in Visual Studio

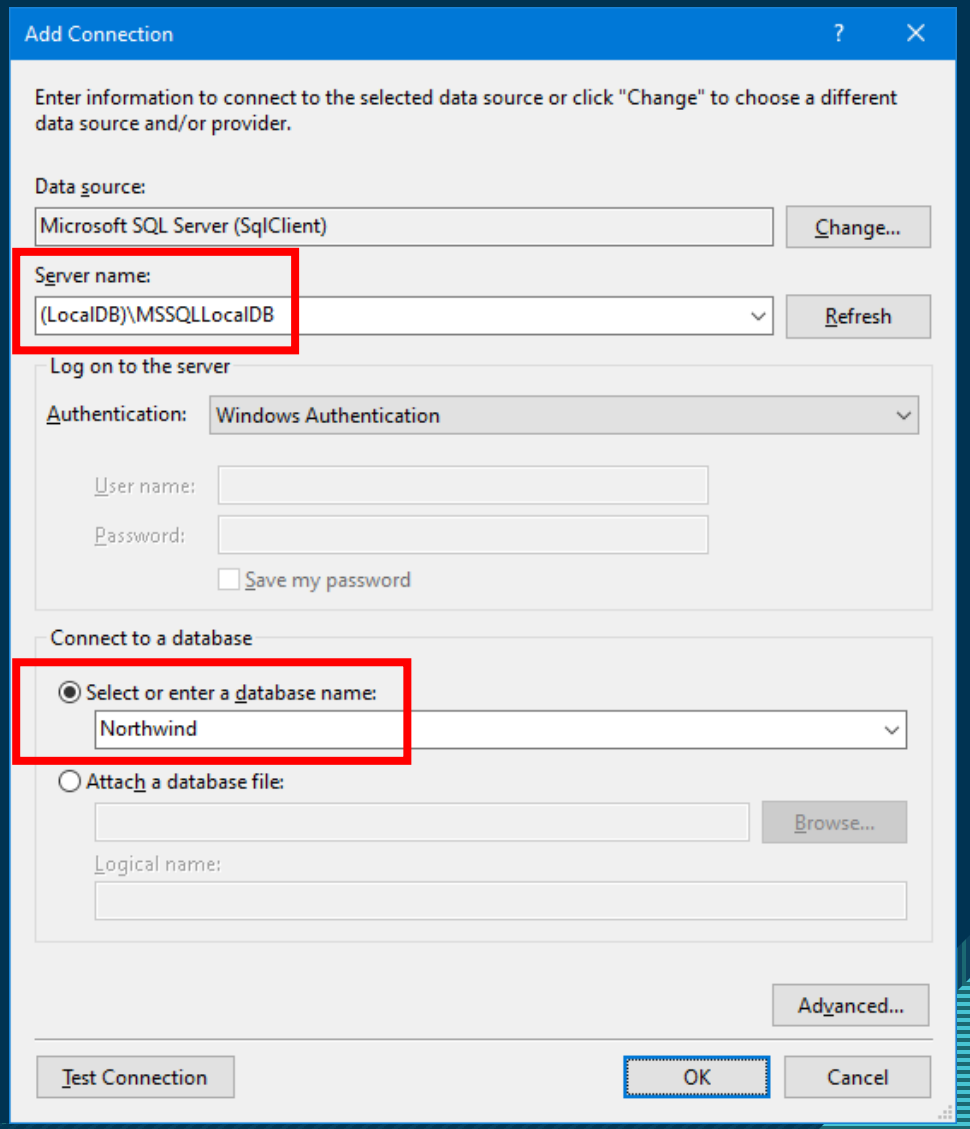
- In Choose Data Source dialog window.
- Select Microsoft SQL Server and click OK.



1. LocalDB in Visual Studio

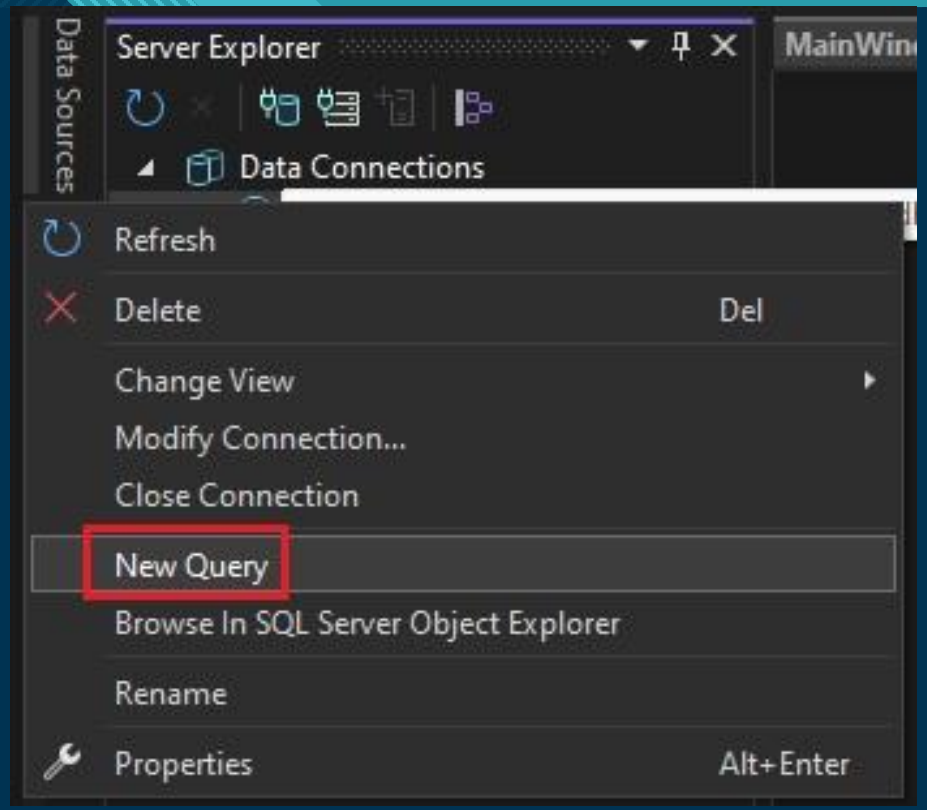
- Type the Server name: (LocalDB)\MSSQLLocalDB.
- Enter the database name: Northwind.
- On clicking OK, a warning message will be displayed.
- Just click Yes.





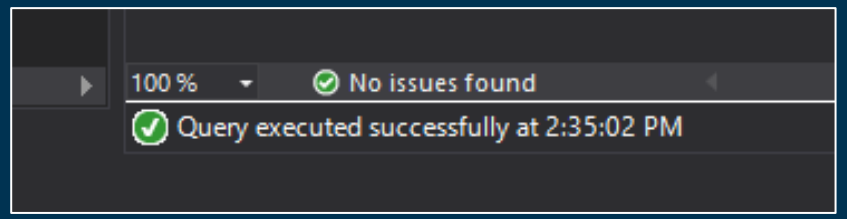
1. LocalDB in Visual Studio

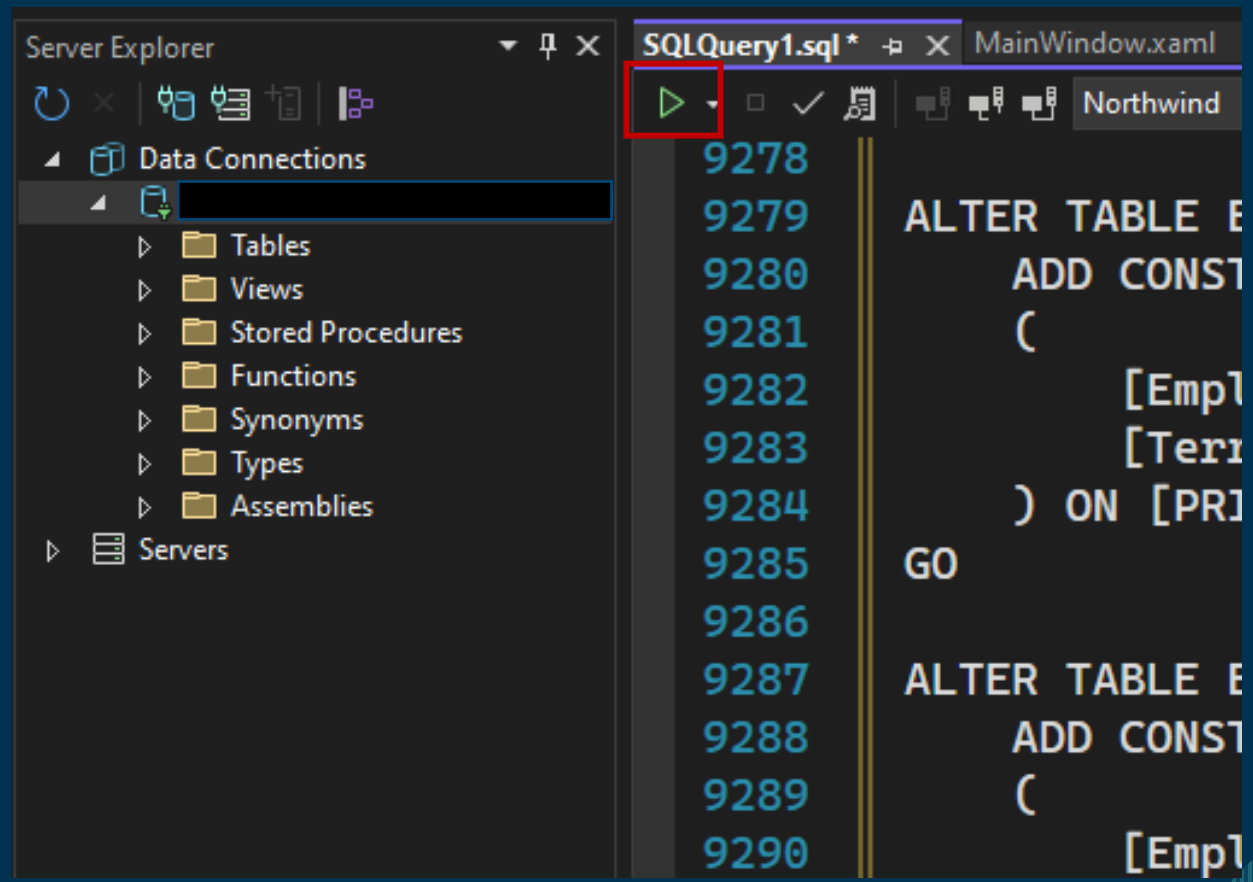
- This will create the database and the connection will be added in the Server Explorer.
- Right-click on the connection and select New Query.



1. LocalDB in Visual Studio

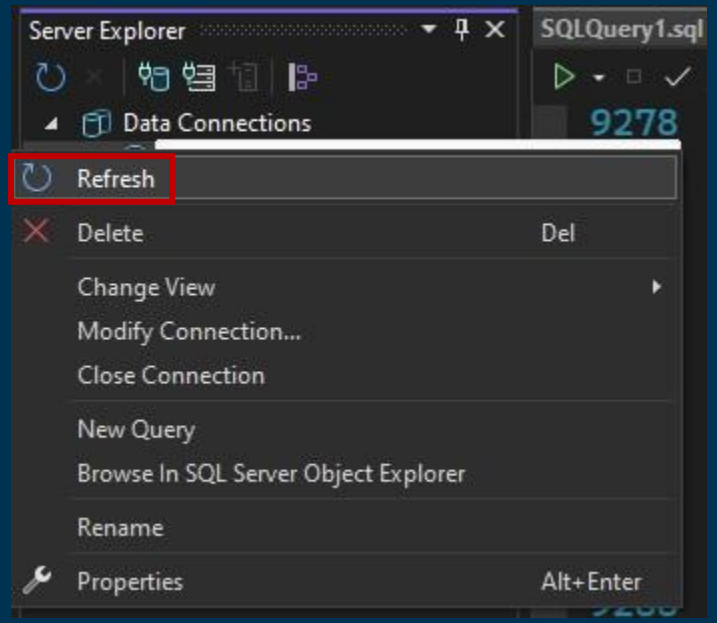
- Go to Northwind-Sample-Database-for-LocalDB.sql that you had opened in a text editor (Notepad) earlier.
- Select all and copy:
  - Ctrl + A and Ctrl + C
- In Visual Studio, paste the copied SQL query.
- Click the **Execute** button and wait for few seconds to finish the execution.

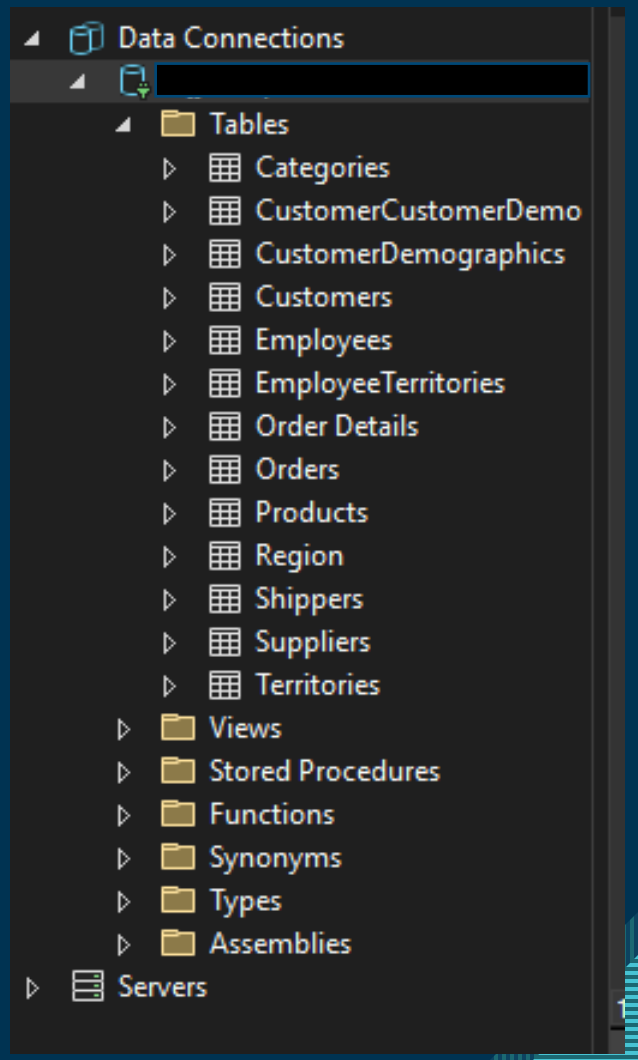




1. LocalDB in Visual Studio

- Right-click on the connection name and select Refresh.





- Expand the Tables node and notice that tables have been created.
- Right-click on any table and select Show Table Data to view the content of the table.

ADO.NET
Connected Models

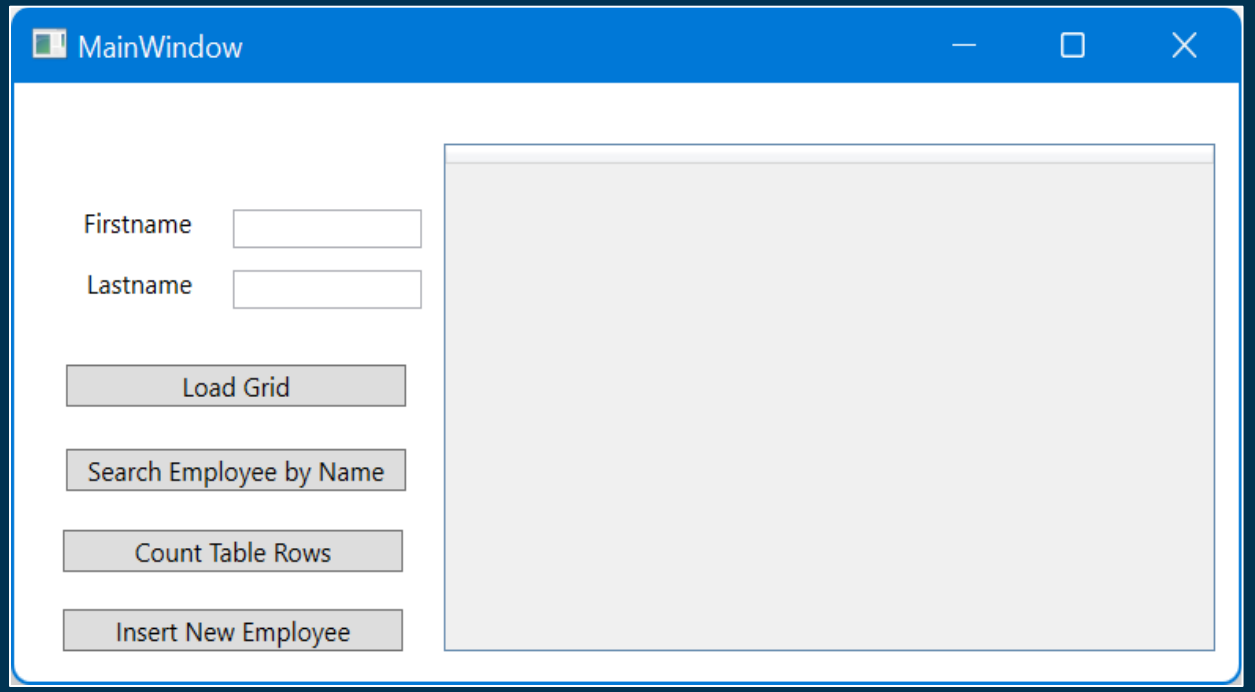
Connected Model

Steps used in the connected model:

1. Create the `SqlConnection` object.
   - You'll need the connection string to create it.
2. Create the `SqlCommand` object.
   - You need the SQL query in string format.
   - And `SqlConnection` object created in the previous step.
3. Open the connection.
4. Execute the `SqlCommand`.
5. Display the result.
6. Close the connection.

Create WPF App

- Create a new WPF app.
- Design a window that looks something like this.
- Give meaningful names to all the controls:
  - Firstname: `txtFirstname`
  - Lastname: `txtLastname`
  - DataGrid: `grdEmployees`
  - Load Grid button: `btnLoad`
  - Search Employees: `btnSearch`
  - Count Table Rows button: `btnCount`
  - Insert Employee button: `btnInsert`



Create Data Class for Connection String

- Add a class and name it something like Data.
- Add a private static field which stores the connection string.
- Add a public static property that returns the connection string.

```
public class Data
{
   private static string connStr =  @"Data Source=(LocalDB)\MSSQLLocalDB;
                                    Initial Catalog=Northwind;
                                    Integrated Security=True";

   public static string ConnectionString { get => connStr; }
}
```

Install NuGet Package

- We need to use the NuGet Package Manager to install a package.
- Right-click on project name in the Solution Explorer and select Manage NuGet Packages….

- Click on the Browse tab and search for the following package:
  - Microsoft.Data.SqlClient

- Install this package.
  - Accept any license agreement that might pop up.

Load Data

- Switch over to the Code View of the WPF app.
- Include the namespaces at the top:
  - `using Microsoft.Data.SqlClient;`
  - `using Microsoft.Data;`
- To populate the DataGrid, create a method `LoadGrid()`.

```
public partial class MainWindow : Window
{
   public MainWindow()
   {
      InitializeComponent();
   }

   private void LoadGrid()
   {

   }
}
```

SqlConnection Object

- Within the LoadGrid method, create the SqlConnection object.
  - Requires the connection string.

```
private void LoadGrid()
{
  SqlConnection conn = new SqlConnection(Data.ConnectionString);
}
```

SqlCommand Object

- Create the SqlCommand object.
  - Need the SQL query in string format.
  - Need the SqlConnection object created in the previous step.
- Then, open the connection.

```
private void LoadGrid()
{
        SqlConnection conn = new SqlConnection(Data.ConnectionString);

        string query = "Select EmployeeID, FirstName, LastName, City,
                        Country from Employees";
        SqlCommand cmd = new SqlCommand (query, conn);
        conn.Open();
}
```

Execute the SQL Command

- Execute the SQL command by calling the `SqlCommand`'s `ExecuteReader()` method.
- This method returns the instance of `SqlDataReader`.

```
private void LoadGrid()
{
        SqlConnection conn = new SqlConnection(Data.ConnectionString);

        string query = "Select EmployeeID, FirstName, LastName, City,
                        Country from Employees";
        SqlCommand cmd = new SqlCommand (query, conn);
        conn.Open();

        SqlDataReader reader = cmd.ExecuteReader();
}
```

Load the DataTable

- Create an instance of the DataTable.
- Load the DataTable with the result of the reader;

```
private void LoadGrid()
{
      SqlConnection conn = new SqlConnection(Data.ConnectionString);

      string query = "Select EmployeeID, FirstName, LastName, City,
                      Country from Employees";
      SqlCommand cmd = new SqlCommand (query, conn);
      conn.Open();

      SqlDataReader reader = cmd.ExecuteReader();

      DataTable tblEmployees = new DataTable();
      tblEmployees.Load(reader);
}
```

Populate the DataGrid

- Use the DataTable object as a source for the DataGrid to populate it.

```
private void LoadGrid()
{
      SqlConnection conn = new SqlConnection(Data.ConnectionString);

      string query = "Select EmployeeID, FirstName, LastName, City,
                      Country from Employees";
      SqlCommand cmd = new SqlCommand (query, conn);
      conn.Open();

      SqlDataReader reader = cmd.ExecuteReader();

      DataTable tblEmployees = new DataTable();
      tblEmployees.Load(reader);

      grdEmployees.ItemsSource = tblEmployees.DefaultView;
}
```

Close the Connection

- After populating the `DataGrid`, the last step is to close the connection.

```
conn.Close();
```

Call the LoadGrid Method

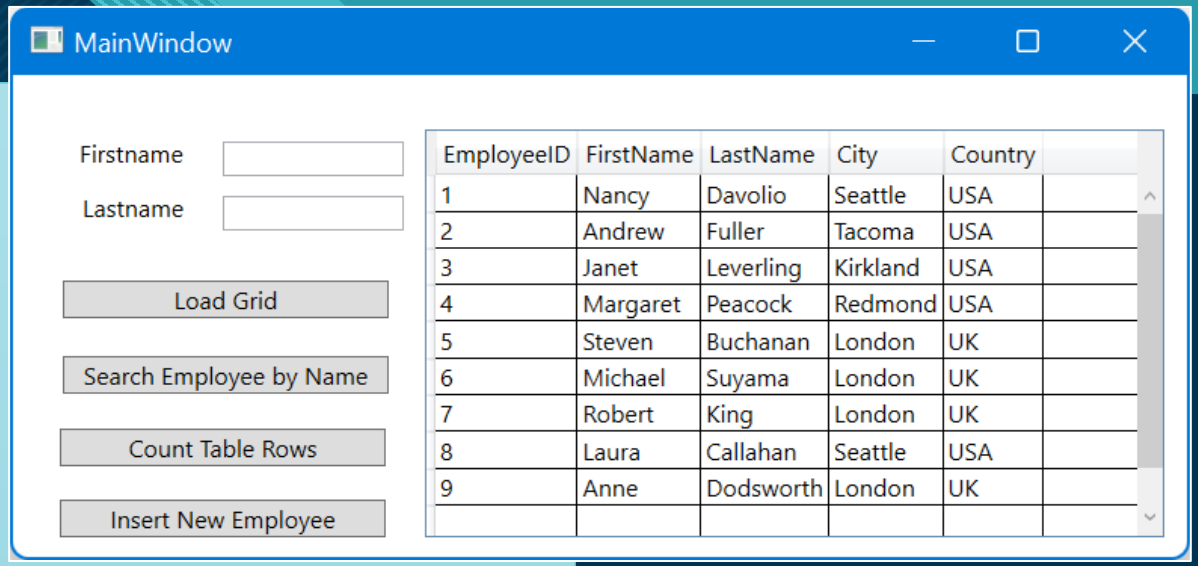
- Call the `LoadGrid` method in the `MainWindow`'s constructor.
- Also, call it in the Load Grid button's click event.

```
public MainWindow()
{
      InitializeComponent();
      LoadGrid();
}
```

```
private void btnLoad_Click(object sender, RoutedEventArgs e)
{
      LoadGrid();
}
```

Complete Code and Output

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        InitializeComponent();
        LoadGrid();
    }
    private void LoadGrid()
    {
        SqlConnection conn = new SqlConnection(Data.ConnectionString);
        DataTable tblEmployees = new DataTable();
        string query = "Select EmployeeID, FirstName, LastName,
                        City, Country from Employees";
        SqlCommand cmd = new SqlCommand(query, conn);
        conn.Open();
        SqlDataReader reader = cmd.ExecuteReader();
        tblEmployees.Load(reader);
        grdEmployees.ItemsSource = tblEmployees.DefaultView;
        conn.Close();
    }
}
```



The `using` Statement

- The connection can automatically be closed by making use of the `using` statement.
- With `using` statement, the `SqlConnection` will be closed without needing to explicitly call the `Close()` method.
- The `SqlConnection` class implements the `IDisposable` interface which has the `Dispose` method in addition to the `Close` method.
- The `using` statement ensures that `Dispose` is called even if an exception occurs within the `using` block.

```
using (SqlConnection conn = new SqlConnection("connection string"))
{
        conn.Open();
        // Execute SQL statement here on the connection you created
}
```

The using Statement

- The previous code can be modified to this (no `conn.Close()` statement is required).

```
private void LoadGrid()
{
    using (SqlConnection conn = new SqlConnection(Data.ConnectionString))
    {
        DataTable tblEmployees = new DataTable();

        string query = "Select EmployeeID, FirstName, LastName,
                        City, Country from Employees";

        SqlCommand cmd = new SqlCommand(query, conn);
        conn.Open();

        SqlDataReader reader = cmd.ExecuteReader();
        tblEmployees.Load(reader);

        grdEmployees.ItemsSource = tblEmployees.DefaultView;
    }
}
```

Command Parameters

- Commands often need parameters.
- For example, the following SQL statement requires a FirstName parameter.
- *Don’t be incited to use string concatenation to build up parameters.*
- Instead, always use the parameter features of ADO.NET.

```
string query = "SELECT EmployeeID, FirstName, LastName, City, Country
                FROM Employees WHERE FirstName = @FirstName";
```

- **Note:** Don’t do string concatenation:

```
string query = "SELECT EmployeeID, FirstName, LastName, City, Country
                FROM Employees WHERE FirstName = " + firstname;
```

Command Parameters

- If the SQL query contains parameter(s), then you need to add the parameter(s) to the `SqlCommand` object.
- To do this, there's a simple way to use the `Parameters` property of the `SqlCommand` and the `AddWithValue` method:

```
cmd.Parameters.AddWithValue("FirstName", "Anne");
```

- **Note:**
  - Don't be inclined to use string concatenation with SQL parameters.
  - This is often misused for SQL injection attacks.
  - Using `SqlParameter` objects prevents such attacks.

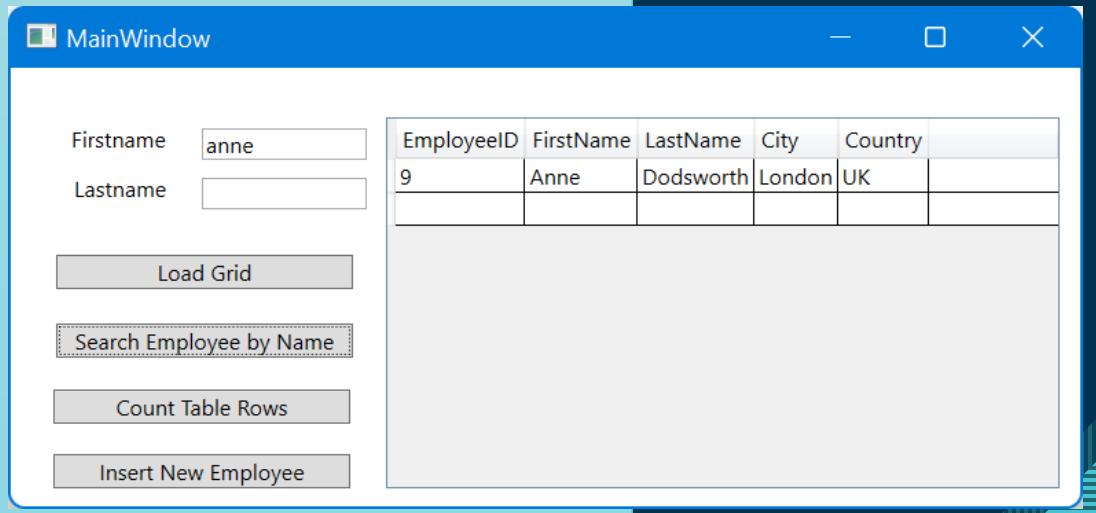
Command Parameters - Example

- Search an employee by their first name.
- Double-click on the button Search Employee by Name to generate its click event:

```
private void btnSearch_Click(object sender, RoutedEventArgs e)
{

}
```

Command Parameters - Example

```
private void btnSearch_Click(object sender, RoutedEventArgs e)
{
   string query = "Select EmployeeID, FirstName, LastName, City,
                   Country from Employees Where FirstName = @FirstName";

   using(SqlConnection conn = new SqlConnection(Data.ConnectionString))
   {
      SqlCommand cmd = new SqlCommand(query, conn);
      cmd.Parameters.AddWithValue("FirstName", txtFirstname.Text);
      conn.Open();

      SqlDataReader reader = cmd.ExecuteReader();

      DataTable tblEmployees = new DataTable();
      tblEmployees.Load(reader);



      grdEmployees.ItemsSource = tblEmployees.DefaultView;
   }
}
```



Executing `SqlCommand`

- The `SqlCommand` class provide methods for storing and executing SQL statements and Stored Procedures.
- The following are the various commands that are executed by the `SqlCommand` class:
- `ExecuteReader`:
  - Returns data to the client as rows.
  - This would typically be an SQL SELECT statement or a Stored Procedure that contains one or more select statements.
  - This method returns a `DataReader` object that can be used to fill a `DataTable` object or used directly for printing reports and so forth.
- `ExecuteScalar`:
  - This method only returns a single value.
  - This kind of query returns a count of rows or a calculated value.
- `ExecuteNonQuery`:
  - Executes a command that changes the data in the database, such as an UPDATE, DELETE, or INSERT statement, or a Stored Procedure that contains one or more of these statements.
  - This method returns an integer that is the number of rows affected by the query.

Executing SqlCommand

- In short, these methods can be summarized as below:

Method	SQL Statements	Description
`ExecuteReader`	SELECT	Used when SQL SELECT statement returns a row or multiple rows.
`ExecuteScalar`	SELECT	Used when SQL SELECT statement returns a single value, like Count (number of rows).
`ExecuteNonQuery`	INSERT, UPDATE, DELETE	Used when INSERT, UPDATE or DELETE statements are used.

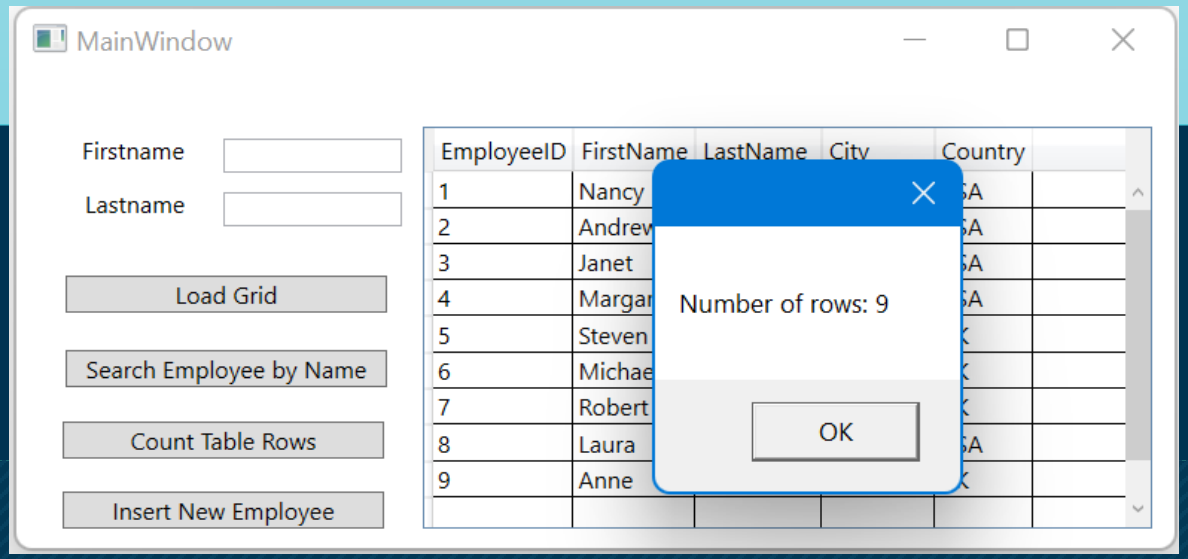
`ExecuteScalar` - Example

- We've already seen example of `ExecuteReader`.
- Now, let's look at `ExecuteScalar`.
- This method is used when the SQL SELECT statement returns a single value.
- The return type of this method is `object`.
  - That is because the returned value can be of any type.
  - So, you can easily cast it to the required type.

- Let's look at an example where we count the number of rows of a table.
- SQL Statement:
  - `SELECT Count(*) FROM Employees;`

ExecuteScalar - Example

```
private void btnCount_Click(object sender, RoutedEventArgs e)
{
   string query = "Select Count(*) from Employees";
   using (SqlConnection conn = new SqlConnection(Data.ConnectionString))
   {
      SqlCommand cmd = new SqlCommand(query, conn);
      conn.Open();

      int numOfRows = (int)cmd.ExecuteScalar();
      MessageBox.Show("Number of rows: " + numOfRows);
   }
}
```



`ExecuteNonQuery` - Example

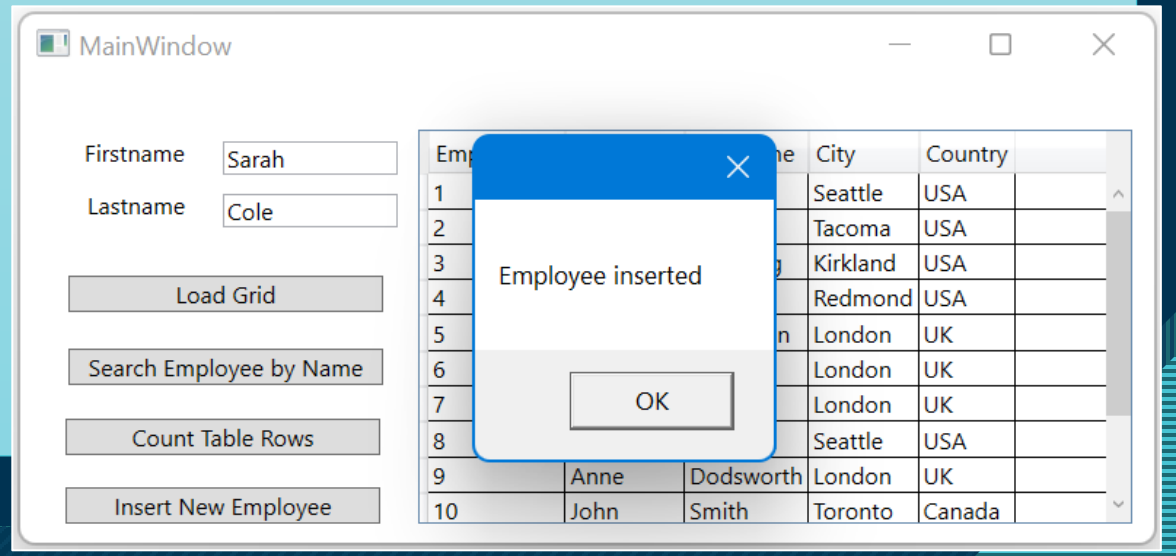
- This method is used when the SQL's INSERT, UPDATE or DELETE statements are used.
- The return type of this method is `int`.
  - It returns the number of rows affected.

- Let's look at an example where we insert a new employee.
- SQL Statement:
  - `Insert into Employees (LastName, FirstName) values (@LastName, @FirstName);`

- Remember to use query parameters instead of concatenation.

ExecuteNonQuery - Example

```
private void btnInsert_Click(object sender, RoutedEventArgs e)
{
    string query = "Insert into Employees (LastName, FirstName) values (@LastName, @FirstName)";
    using (SqlConnection conn = new SqlConnection(Data.ConnectionString))
    {
        SqlCommand cmd = new SqlCommand(query, conn);
        cmd.Parameters.AddWithValue("LastName", txtLastname.Text);
        cmd.Parameters.AddWithValue("FirstName", txtFirstname.Text);
        conn.Open();

        int result = cmd.ExecuteNonQuery();

        if (result == 1)
            MessageBox.Show("Employee inserted");
        else
            MessageBox.Show("Employee not inserted");
    }
}
```



Do It Yourself!

- Create an app where the user can perform the following actions:
  1. Get all Employees
  2. Search Employee by last name
  3. Update an Employee record
  4. Exit

- Use Employee table in Northwind database.
- Allow partial matches when searching by last name.
  - For example: If searching by last name user enters **an**, it should display all employees that contain **an** in their name.
  - Hint: Use SQL's LIKE operator.
- Use connected model.

Thank You

References

Some of the material has been taken from:

- Basics of ADO.NET:
  - https://www.c-sharpcorner.com/UploadFile/18fc30/understanding-the-basics-of-ado-net/