ADO.NET

What is ADO.NET:-

The .NET Framework includes its own data access technology,ADO.NET.

ADO. NET consists of managed classes that allow .NET applications to Connect to data sources (usually relational databases),execute commands, and manage disconnected data.

Connected Architecture

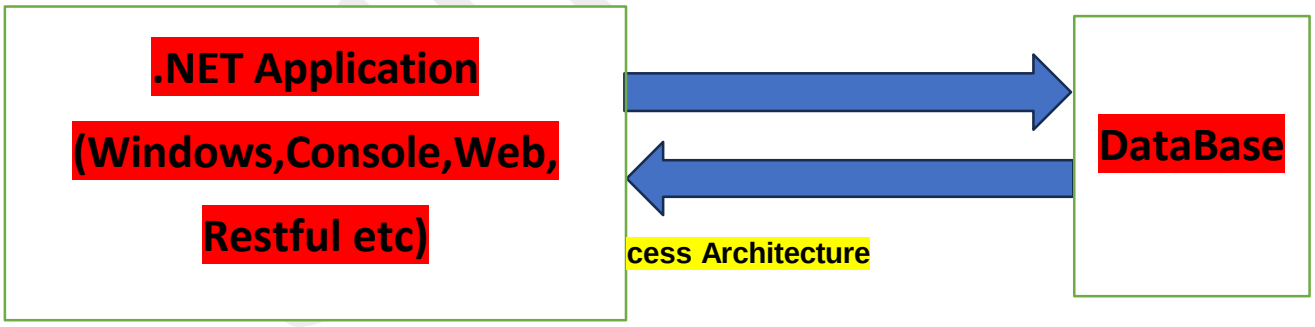
Connection:

Manages the connection to the database. In the case of Connection-Oriented Data Access Architecture, an open and active connection is always required between the .NET Application and the database.when we access data from the database, the Data Reader object requires an active and open connection to access the data. If the connection is closed, we cannot access the data from the database; in that case, we will get the runtime error.

Command:
Executes SQL queries, stored procedures, or commands on the database.The second important component is the command object. When we discuss databases such as SQL Server, Oracle, MySQL, then speak SQL, it is the command object that we use to create SQL queries. After you create your SQL queries, you can execute them over the connection using the command object.

DataReader:

Reads data from the database in a forward-only, read-only manner.The ADO.NET DataReader object reads the data from the database using Connected Oriented Architecture.It requires an active and open connection while reading the data from the database.



Select Query

```
        string connectionDetails = @"Data Source=(LocalDB)\MSSQLLocalDB;Initial
Catalog=kdac;Integrated Security=True";

   SqlConnection connection =
            new SqlConnection(connectionDetails);
        connection.Open();

        SqlCommand command =
            new SqlCommand("select * from Emp", connection);
```

```
SqlDataReader reader = command.ExecuteReader();

List<Emp> emps = new List<Emp>();

while (reader.Read())
{
    Emp emp = new Emp();
   emp.No = Convert.ToInt32(reader["No"]);
    emp.Name = reader["Name"].ToString();
    emp.Address = reader["address"].ToString();

    emps.Add(emp);
}
connection.Close();

foreach (Emp emp in emps)
{
    Console.WriteLine(emp.ToString());
}
```

```
SqlConnection connection=new SqlConnection(connectionDetails);
```

1. **SqlConnection:**
   This is a class provided by the .NET framework for working with SQL Server databases. It represents a connection to a SQL Server database. This class is part of the **System.Data.SqlClient** namespace.

2. **connection:** This is the variable name that hold the `SqlConnection` object.

3. **new:** This keyword is used to create a new instance of a class.

SqlCommand command = new SqlCommand("select * from Emp", connection);

1. **SqlCommand command :**
   This line declares a variable named `command` of type `SqlCommand`. `SqlCommand` is a class provided by the ADO.NET framework, which is used to execute SQL commands against a database.

2. **new SqlCommand("select * from Emp",connection**): SqlCommand class takes two parameters

   a. **select * from Emp -** This is the SQL command that you want to execute.select statement that retrives all columns from a table.
   b. **connection** – This is an instance of a `SqlConnection` class. It represents the connection to the database where you want to execute the SQL command.

SqlDataReader reader = command.ExecuteReader();

1. **SqlDataReader reader:**
   This line declares a variable named `reader` of type `SqlDataReader`. ` SqlDataReader ` is a class provided by the ADO.NET for reading a forward-only streams of rows from database.

2. **command.ExecuteReader():**
   This method is used to execute the SQL command represented by the **command** object and retrieve the resulting data as a data reader. When you call `ExecuteReader()`, it sends the SQL command to the database server for execution.

DisConnected Architecture

- DataAdapter: This object performs two tasks. First, you can use it to fill a DataSet (a disconnected collection of tables and relationships) with information extracted from a data source. Second, you can use it to apply changes to a datasource, according to the modifications you've made in a DataSet. The ADO.NET DataAdapter object acts as an interface between the .NET application and the database.
- DataSet: A DataSet is made up of a collection of tables, relationships, and constraints. In ADO.NET, DataTable objects are used to represent the tables in a **DataSet**.
  The ADO.NET DataSet is a memory-resident representation of data that provides a consistent relational programming model regardless of the source of the data it contains. A DataSet represents a complete set of data including the tables that contain, order, and constrain the data, as well as the relationships between the tables.

  Creating a DataSet:

  You can create an instance of a DataSet by calling the DataSet constructor. Optionally specify a name argument. If you do not specify a name for the DataSet, the name is set to "NewDataSet". You can also create a new DataSet based on an existing DataSet.

  The new DataSet can be an exact copy of the existing DataSet, a clone of the DataSet that copies the relational structure or schema but that does not contain any of the data from the existing DataSet; or a subset of the DataSet, containing only the modified rows from the existing DataSet using the GetChanges method.

  - **Data Table:**

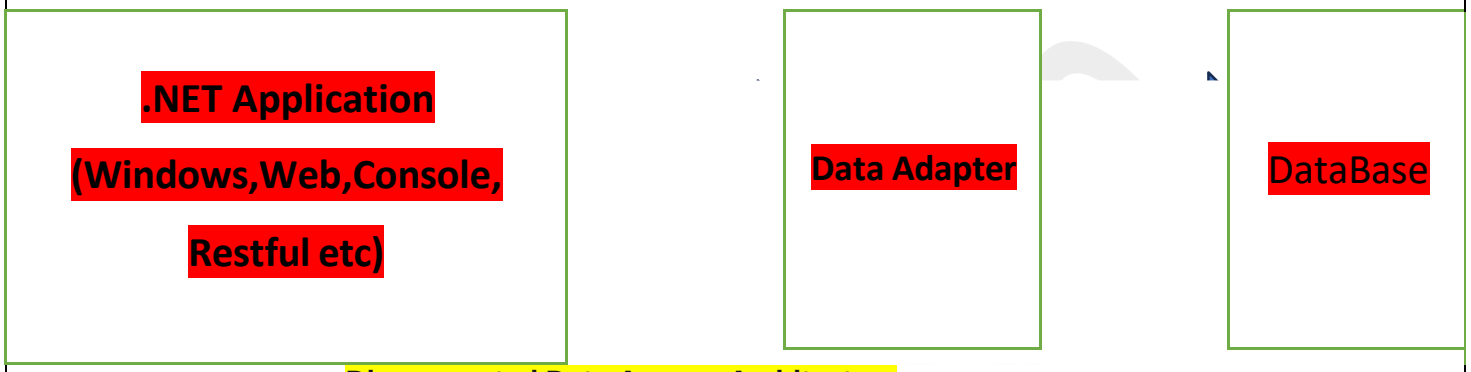
    The **DataTable** class is a member of the **System.Data** namespace within the .NET Framework class library. You can create and use a **DataTable** independently or as a member of a **DataSet**, and **DataTable** objects can also be used in conjunction with other .NET Framework objects, including the DataView. You access the collection of tables in a **DataSet** through the **Tables** property of the **DataSet** objects.

    Creating a DataTable:

    A DataTable, which represents one table of in-memory relational data, can be created and used independently, or can be used by other .NET Framework objects, most commonly as a member of a DataSet.

    You can create a DataTable object by using the appropriate DataTable constructor. You can add it to the DataSet by using the Add method to add it to the DataSet objects Tables collection.

When you first create a DataTable, it does not have a schema (that is, a structure). To define the schema of the table, you must create and add objects to the Columns collection of the table. You can also define a primary key column for the table, and create and add Constraint objects to the Constraints collection of the table. After you have defined the schema for a DataTable, you can add rows of data to the table by adding DataRow objects to the Rows collection of the table.

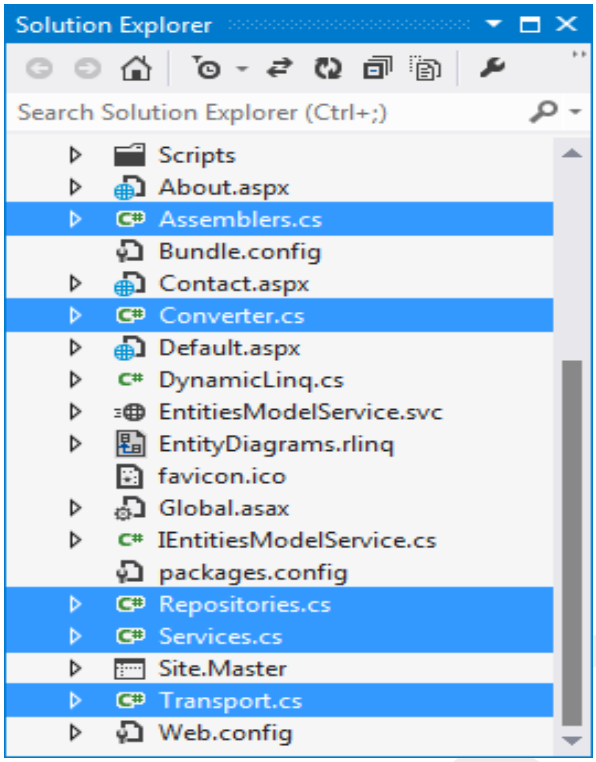


Disconnected Data Access Architecture

- **Writing DTO Layer:-**

Data Transfer Objects (DTO) are used to transfer data between the **Application Layer** and the **Presentation Layer** or other type of clients.

A DTO is nothing more than a container class that exposes properties but no methods. A data transfer object (DTO) holds all data that is required for the remote call. Your remote method will accept the DTO as a parameter and return a DTO to the client.

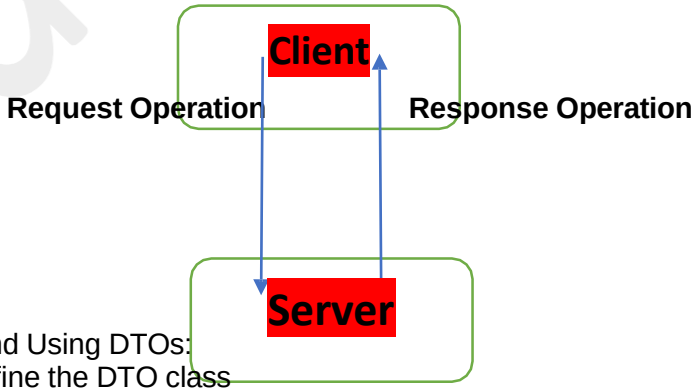


Why Use DTOs?

The use of DTOs is very common in web development. They provide solutions for many needs. Below are some of them:

- Separate the service layer from the database layer.
- Hide specific properties that clients don't need to receive.
- Manipulate nested objects to make them more convenient for clients.

DTOs provide an efficient way to separate domain objects from the presentation layer. This way, you can change the presentation layer without affecting the existing domain layers, and vice versa.



Creating and Using DTOs:
Step 1: Define the DTO class

Suppose we have a basic entity class representing employees.

```
public class Employee
{
    public int EmpId { get; set; }
    public string FirstName { get; set; }
```

```
    public string LastName { get; set; }
    public string Department { get; set; }

}
```

Now Lets create a DTO class for the Employee entity.

```
public class EmployeeDTO
{
    public int EmpId { get; set; }
    public string FullName { get; set; }
    public string Department { get; set; }
}
```

In this EmployeeDTO class, we're selecting specific fields (EmployeeId, FullName, and Department) that we want to transfer between different parts of our application. The FullName property is a computed property, as it's not present in the original Employee entity.

Step 2: Converting Entity To DTO:

```
public static class Employeemapper
{
    public static EmployeeDTO MapToDTO(Employee employee)
    {
        return new EmployeeDTO
        {
                  EmpId = employee.EmpId,
                    FullName = $"{employee.FirstName}
                {employee.LastName}",
                  Department = employee.Department
        };
    }

}
```

Step 3: Using DTOs in Our Application:

Suppose we have a service that retrives employee data from a database.

```
public class EmployeeService
{
  private List<Employee> _employees = new List<Employee>
  {
    new Employee { EmployeeId = 1, FirstName = "Amit", LastName =
"Patil", Department = "IT" },

new Employee { = 2, FirstName = "raj", LastName = "Sutar",
Department = "HR" },

new Employee { EmployeeId = 3, FirstName = "Bhaiya", =
"Magdum", Department = "Fir nce" },


  };
```

```
        public EmployeeDTO GetEmployeeById(int employeeId)
        {
    var employee = _employees.FirstOrDefault(e => e.EmployeeId == employeeId);

        if(employee == null)
        {
           return null;
        }

          return EmployeeMapper.MapToDTO(employee);
        }

        }
```

In this EmployeeService, we have a method GetEmployeeById that retrieves an employee by their ID and maps the retrieved Employee object to an EmployeeDTO using the EmployeeMapper class

- **Entity Framework:-**

  Entity Framework is an open source object–relational mapping framework for ADO.NET
  Entity Framework is an Object Relational Mapper (ORM) which is a type of tool that simplifies mapping between objects in your software to the tables

  commands, as well as taking query results and automatically materializing those results as your application objects.

  - **Writing POCO-:**

    POCO stands for "Plain Old CLR Object," and in the context of Entity Framework, POCO classes are simple, non-framework-specific classes that do not depend on any Entity Framework types.
    These classes are used to represent entities in the domain model, and they can be used with Entity Framework for data access without requiring any dependencies on the framework itself.
    POCO classes are often used to create a more maintainable and testable domain model, as they are not tied to a specific data access technology.

Creating POCO Classes:

1. **Define Your Classes:** You start by defining your entity classes as plain C# classes. For example,if you have a “Customer” entity,you would create a class named ‘Customer’.
2. **Add Properties:** Add properties to your classes to represents the attributes of your entities. Each property typically corresponds to a column in a database table.
3. **Decorate with Attributes:** You can use attributes such as [Key],[Required],[StringLength].

Mapping POCO Classes to Database Tables:

1. **DbContext:** You need to create **a 'DbContext'** class, which represents a session with the database and acts as a bridge between your domain classes and the database.
2. **DbSet<TEntity>:** Inside your 'DbContext' class, you define `DbSet<TEntity>` properties,where `TEntity` is your POCO class.

The Code Given Below Defines a POCO Class.

```
public class Customer
 {
    public int CustomerID {get; set;}
    public string Name { get; set; }
    public string Address { get; set; }
}

public class MyDbContext : DbContext {
 public DbSet<Customer> Customers { get; set; }
}
```

In this example, Customer is a POCO class representing a customer entity, and `MyDbContext` is a `DbContext` class containing a DbSet<Customer> property for accessing the "Customers" table in the database.

- **Using Attributes-:**

Attributes are like adjectives, which are used for metadata annotation that can be applied to a given type, assembly, module, method and so on. The .NET framework stipulates two types of attribute implementations, which are Predefined Attributes and Custom Attributes. Attributes are types derived from the System.Attribute class. This is an abstract class defining the required services of any attribute. The following is the syntax of an attribute;
[type: attributeName(parameter1,parameter2, ------- n)]

Role Of Attributes-

Attributes might be useful for documentation purposes. They fulfill many roles, including describing serialization, indicating conditional compilation, specifying import linkage and setting class blueprint and so on. Attributes allow information to be defined and applied to nearly any metadata table entry.

Prefined Attributes-

The System.Attribute base class library provides a number of attributes in various namespaces. The following table gives a snapshot of some predefined attributes.

Attributes	Description
[Serialization]	By marking this attributes, a class is able to persist its current state into stream.

[NonSerialization]	It specify that a given class or field should not persisted during the serialization process.

Here, assume that you have developed a Test class that can be persisted in a binary format using the [Serialization] attribute.

```
[Serializable]
 public class test
{
      public Test() { }
      string name;
      string country;

       [NonSerialized]
       int salary;
}
```

1. Table Attribute-

The Table attribute is applied to an entity to specify the name of the database table that the entity should map to. The following example specifies that the Book entity should map to a database table named tbl_Book:

```
[Table("tbl_book")]
public class Book
{
   public int BookId { get; set; }
   public string Title { get; set; }
   public Author Author { get; set; }

}
```

2. Column Attribute-

The Column attribute is applied to a property to specify the database column that the property should map to when the entity's property name and the database column name differ. The following example maps the Title property in the Book entity to a database column named Description:

```
public class Book
{
    public int BookId { get; set; }
    [Column("Description")]
    public string Title { get; set; }
    public Author Author { get; set; }
}
```

3. ForeignKey-

The ForeignKey attribute is used to specify which property is the foreign key in a relationship.

```
public class Author
```

```
{
   public int AuthorId {  get; set; }
   public string FirstName { get; set; }
   public string LastName { get; set; }
   [ForeignKey("AuthorFK")]
   public Icollection<Book> Books {  get; set ; }
}
```

4. DatabaseGenerated-

The DatabaseGenerated attribute specifies how values are generated for a property by the database. The attribute takes a DatabaseGeneratedOption enumeration value.

Computed-

The Computed option specifies that the property's value will be generated *by the database* when the value is first saved, and subsequently regenerated every time the value is updated.

```
public class Contact
 {
    public int Id { get; set; }
    public String FullName { get; set; }
    public String Email { get; set; }
   [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
   public DateTime LastAccessed { get; set; }

 }
```

Identity-

The Identity option specifies that the value will only be generated by the database when a value is first added to the database.

Database providers differ in the way that values are automatically generated. Some will generate values for selected data types such as Identity,rowversion.

```
public class Contact
 {
    public int Id { get; set; }
       public String FullName { get; set; }
       public String Email { get; set; }
      [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
      public DateTime Created  { get; set; } = DateTime.UtcNow;
}
```

5. MaxLength-

The MaxLength attribute is applied to a property to specify a maximum number of characters or bytes for the column that the property should map to.

The following example specifies that the Title column in the Books table is to have a maximum length of 150 characters**.**

```
public class Book
        {
          public int BookId { get; set; }
          [MaxLength(140)]
          public String Title { get; set; }
          public Author Author { get; set; }

           }
```

- **Entity Framework Packages-:**

Entity Framework core is shipped as NuGet packages.The packages needed by an application depends on:

- The type of database system being used (SQL Server,SQLite etc)
- The EF core features needed.

The usual process for installing packages is:

- Decide on a database provider and install the appropriate package.
- Also install Microsoft.EntityFrameworkCore and SQprovider.This helps ensure that consistent versions are being used,and also means that NuGet will let you know when new package versions are shipped.
- Optionally decide which kind of tooling you need and install the appropriate packages for that

Database providers:

EF Core supports different database systems through the use of “database providers ”. Each system has its own database provider,which is shipped as NuGet package. Applications should install one or more of these provider packages.
Common database providers are listed in the table below.

Database System	Package
SQL Server	Microsoft.EntityFrameworkCore.SqlServer
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL
MySQL	EntityFrameworkCore.MySql

- **Commands-:**

The command-line interface tools for Entity Framework core perform design-time development tasks. For Example they create migrations,apply migrations and generate code for a model based on an existing database. The commands are an extension to the cross-platform dotnet command, which is part of the .NET Core SDK.
When using Visual Studio,consider using the Package Manager Console tools instead of the CLI tools. Package Manager Console tools automatically.

- Works with the current project selected in the Package Manager Console without requiring that you manually switch directories.
- Opens files generated by a command after the command is completed.
- Provides tab completion of commands,parameters,project names,context types and migration names.

dotnet ef database update

Updates the database to the last migration or to a specified migration.

Argument	Description

<MIGRATION>	The target migration. Migrations may be identified by name or by ID. The number 0 is a special case that means *before the first migration* and causes all migrations to be reverted.

dotnet ef dbcontext info

Gets information about a DbContext type.
The common options are listed above.

1. dotnet ef dbcontext optimize
   - Generates a compiled version of the model used by the DbContext.
2. dotnet ef dbcontext scaffold
   - Generates code for a DbContext and entity types for a database. In order for this command to generate an entity type, the database table must have a primary key.

Arguments:
<CONNECTION>
<PROVIDER>

dotnet ef migrations add

Adds a new migration.

<NAME> Name Of Migrations.

dotnet ef migrations remove

Removes the last migration, rolling back the code changes that were done for the latest migration.

--force	Revert the latest migration, rolling back both code and database changes that were done for the latest migration.

Different Approaches
The Entity Framework provides three approaches to create an entity model and each one has their own pros and cons.

- Code First
- Database First
- Model First

- **Model First Approach-:**

- Model First is good when you are starting a new project where the database doesn't even exist yet.
- The model is stored in an EDMX file and can be viewed and edited in the Entity Framework Designer.
- In Model First, you define your model in an Entity Framework designer then generate SQL, which will create database schema to match your model and then you execute the SQL to create the schema in your database.
- The classes that you interact with in your application are automatically generated from the EDMX file.

General overview of how you can use the Model First approach with relations in Entity Framework:

1. **Create the Conceptual Model:**
   - Use Entity Framework Designer in Visual Studio or any other tool that supports Entity Framework modeling to create your conceptual model.
   - Define entities (classes) representing your domain objects. Each entity typically corresponds to a table in the database.
   - Define relationships between entities by specifying navigation properties. These navigation properties represent the associations between entities.

2. **Configure Relationship:**
   - In the conceptual model, you can define relationships between entities. These relationships can be one-to-one, one-to-many, or many-to-many.
   - Configure the cardinality and navigation properties for each relationship. For example, if you have a one-to-many relationship between Department and Employee, you would specify a collection navigation property in the Department entity to represent the employees associated with that department.

3. **Generate Database Schema:**
   - Once you have defined your conceptual model with relationships, can generate the database schema from it.
   - Entity Framework provides tools to generate SQL scripts or directly create the database based on your conceptual model.
   - When generating the database schema, Entity Framework will create

tables, columns, and foreign key constraints according to the relationships you defined in the conceptual model.

4. **Update and Synchronize:**
    - As your conceptual model evolves, you can make changes to it (e.g., adding new entities or modifying relationships).
    - Entity Framework provides mechanisms to update the database schema to reflect these changes.
5. **Use the Model in your Application:**
    - Once the database schema is generated **,** you can use Entity Framework to perform CRUD (Create, Read, Update, Delete) operations on your entities.
    - Entity Framework will handle the translation of your LINQ queries into SQL and the mapping of database results back to your entity objects.