UNIT-5
Working with Database

⊛ ADO.NET Basics:

ADO stands for Microsoft ActiveX Data Objects. The ADO.NET is one of the Microsoft's data access technology which is used to communicate between the .NET Application (Console, WCF, WPF, Windows, MVC, Web Form, etc.) and data sources such as SQL Server, Oracle, MySQL, XML, document, etc. It has classes and methods to retrieve and manipulate data. The following are a few of the .NET applications that use ADO.NET to connect to a database, execute commands and retrieve data from the database.

→ ASP.NET Web Applications
→ Console Applications
→ Windows Applications

Types of Connection Architectures:

1. Connected architecture: The application remains connected with the database throughout the processing.
2. Disconnected architecture: The application automatically connects/disconnects during the processing. The application uses temporary data on the application side called a DataSet.

⊛. Important Classes in ADO.NET:

1) Connection class: In ADO.NET, we use the connection classes to connect to the database. These connection classes also mange transactions and connection pooling.

2) 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 onwards.

↳ 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.

↳ ExecuteXMLReader: (SqlClient classes only) Obtains data from an SQL Server 2000 database using an XML stream. Returns an XML Reader object.

iii) DataReader Class: The DataReader is used to retrive data. It is used in conjunction with the Command class to execute an SQL Select statement and then access the returned rows.

iv) DataAdaptor Class: The DataAdaptor is used to connect DataSets to databases. The DataAdaptor is most useful when using data-bound controls in Windows Forms.

v) DataSet Class: The DataSet is essentially a collection of DataTable objects. In turn each object contains a ~~Relations collection that~~ collection of DataColumn and DataRow objects. The DataSet also contains a Relations collection that can be used to define relations among Data Table Objects.

⊛ Connect to a Database using ADO.NET: [Imp]

A connection string is required as a parameter to SQLConnection. This ConnectionString is a string variable (not case sensitive). This contains key and value pairs: Provider, Server, Database, User Id and Password as in the following:

Server="name of the server or IP Address of the server"
Database= "name of the database"
UserId="user name who has permission to work with database",
Password= "the password of UserId"

Example:

```
string constr="server=.; database=db1; user id=sa;
                password= mypassword";
```

⊛. How to connect, retrive and display data from a database:

→ Create a SqlConnection object using a connection string.
→ Handle exceptions
→ Open the connection
→ Create a SQLCommand. (like select *from student) and attach the existing connection to it.
→ Execute the command (use ExecuteReader).
→ Get the Result (use SqlDataReader).
→ Process the Result.
→ Display the Result.
→ Close the connection.

Example1: Showing connection and inserting data to database table.

```
string connStr="Data Source=AM; Initial Catalog=TestDb;
                User ID=sa; Password=12345";
SqlConnection conn = new SqlConnection(connStr);
string sql= "Insert into AddressBook values ('"+Ram+"', +
        "' Kathmandu +"', '"+9806470952+"')";
SqlCommand cmd =new SqlCommand(sql, conn);
conn.Open();
cmd.ExecuteNonQuery();
conn.Close();
```

Example2: Reading Data with SqlDataAdapter & DataSet:

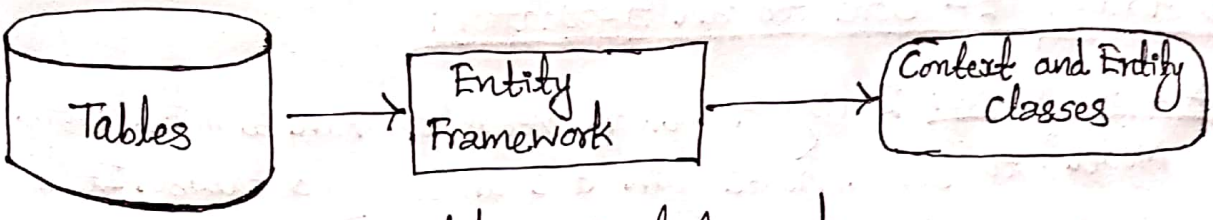
```
String connStr = "Data Source=.; Initial Catalog=TestDb; User Id=sa;
                  Password=123456";
SqlConnection conn = new SqlConnection(connStr);
string sql = "Select * from AddressBook";
SqlDataAdapter da = new SqlDataAdaptor (sql, conn);

DataSet ds = new DataSet();
da.Fill(ds);
GridView1.DataSource = ds;
GridView1.DataBind();
```

⊛. Entity Framework (EF) Core:

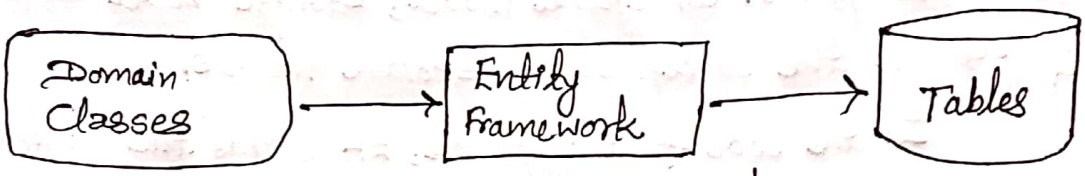
Entity Framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing and storing the data in the database. It is open-source, lightweight, extensible and cross-platform version of Entity Framework data access technology EF Core is intended to be used with .NET Core applications. However, it can also be used with standard .NET 4.5+ framework based applications.

⊛ EF Core Development Approaches:

EF Core supports two development approaches: Code-First and Database-First. EF Core mainly targets the code-first approach and provides some support for the database-first. In the code-first approach, EF Core API creates the database and tables using migration based on the conventions and configuration provided in domain classes. This approach is useful in Domain Driven Design (DDD). In the database-first approach, EF Core API creates the domain and context classes based on existing database using EF Core commands. This has limited support in EF Core as it does not support visual designer or wizard.



Database-First Approach
Generate Data Access Classes for Existing Database.

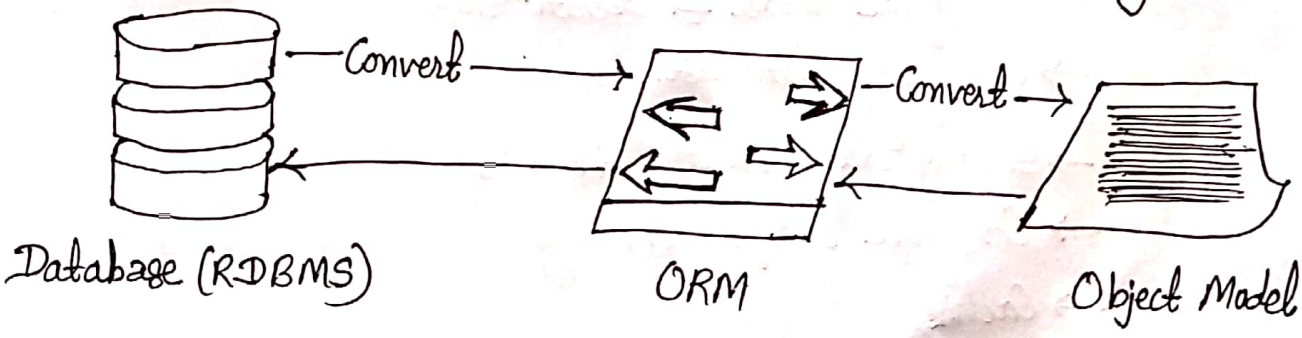


Code-First Approach
Create Database from the Domain Classes.

⊛ Object-Relational Mapper (ORM): [Imp]

Essential parts of an ASP.NET MVC application is the architectural design. It's the Model-View-Controller (MVC) pattern. A one of basic end point of project is the Database. We have to access the DB from the next layer (Controller). In that point, object relational mapper (ORM) will come to the battle.

An ORM is an application or system that support in the conversion of data within a relational database management system (RDBMS) and the object model that is necessary for use within object-oriented programming.



⊛. Adding EF Core to an application:

⊛Data Models: Entity Framework needs to have a model (Entity Data Model) to communicate with the underlying database. It builds a model based on the shape of our domain classes, the Data Annotations and Fluent API configurations. The EF model includes three parts: conceptual model, storage model and mapping between the conceptual and storage models.

In the code-first approach, EF builds the conceptual model based on domain classes (entity classes), the context class and configurations. EF Core builds the storage model and mappings based on the provider use. EF uses this model for CRUD (Create, Read, Update, Delete) operations to the underlying database.

⊛. Data Context: The DbContext class is an integral part of Entity Framework. An instance of DbContext represents a session with the database which can be used to query and save instances of entities to a database.

DbContext is a combination of the Unit of Work and Repository patterns. DbContext in EF Core allows us to perform following tasks:

- → Manage database connection.
- → Configure model & relationship.
- → Querying database.
- → Saving data to the database.
- → Configure change tracking
- → Caching
- → Transaction management.

Using Entity Framework

⊛ Querying and Saving data to database: (CRUD Operations):

Q. Create methods to insert, update, delete and read all data for the table Student having following fields StudentId (int), Name varchar (200), RollNo (int), Class varchar (50) using Entity Framework. [Model Question Imp.]

Solution:

```
namespace DatabaseOperations
{
    class EFCoreTest
    {
        private StudentContext _context = new StudentContext();
        public List<Student> GetAll()
        {
            return _context.tblStudent.ToList();
        }
        public void InsertStudent(Student emp)
        {
            _context.Add(emp);
            _context.SaveChanges();
        }
        public int EditStudent(Student emp)
        {
            if (emp.StudentId == -1)
            {
                return 0;
            }
            var stud = _context.tblStudent.Find(emp.StudentId);
            if (stud == null)
            {
                return 0;
            }
            int updatedCount = 0;
            try
            {
                _context.Update(stud);
                updatedCount = _context.SaveChanges();
            }
```

```
        catch (DbUpdateConcurrencyException)
        { return 0;
        }
        return updatedCount;
    }
    public int deleteStudent (int StudentId)
    {
        if (StudentId = -1)
        { return 0;
        }
        var stud = _context.tblStudent.FirstOrDefault (m => m.StudentId
                                                    = StudentId);
        if (stud == null)
        { return 0;
        }
        _context.tblStudent.Remove (stud);
        int updatedCount = _context.SaveChanges ();
        return updatedCount;
    }
    public void showStudents (List<Student> students)
    {
        foreach (Student student in students)
        {
            Console.WriteLine ("Student ID =" + student.StudentId);
            Console.WriteLine ("Student Name =" + student.Name);
            Console.WriteLine ("Student RollNo =" + student.RollNo);
            Console.WriteLine ("Student Class =" + student.Class);
        }
    }
}
Class Program
{
    static void Main (string[] args)
    {
        EFCoreTest test = new EFCoreTest ();
```

```
//Read All Students
List<Employee> employees = test.GetAll();
Console.WriteLine("Initial Records");
test.showStudents(employees);

//Insert Student
Employee stud = new Employee();
stud.Name = "Ram";
stud.RollNo = 31;
stud.Class = "Sixth Sem";
test.InsertStudent(stud);
Console.WriteLine("After Insert");
test.showStudents(employees);

//Update Student
stud.Name = "Updated Name";
test.EditStudent(stud);
Console.WriteLine("After Update");
test.showStudents(employees);

//Delete student
test.deleteStudent(stud.EmployeeId);
Console.WriteLine("After Delete");
test.showStudents(employees);

Console.ReadLine();
}
}
}
```