Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

INTRODUCTION TO DATABASES

DATABASE MANAGEMENT SYSTEMS (SQL SERVER, MYSQL, ETC.)

A Database Management System (DBMS) is a software system designed to manage databases and provide an interface for users and applications to interact with the data stored in the database. DBMSs help in storing, organizing, and retrieving large amounts of data in an efficient manner. There are several types of DBMSs available, and among the most widely used are **SQL Server** and **MySQL**, both of which are relational database management systems (RDBMS). Understanding how these DBMSs work is fundamental to building and managing applications that require persistent data storage.

SQL Server

SQL Server is a relational database management system developed by Microsoft. It is known for its scalability, security features, and integration with other Microsoft services. SQL Server supports T-SQL (Transact-SQL), an extension of SQL (Structured Query Language), which is used to query and manage data. SQL Server provides several features like data integrity, backup and recovery, high availability, and replication, making it ideal for enterprise-level applications. It also includes tools like SQL Server Management Studio (SSMS) for managing databases and querying data.

SQL Server can handle large-scale enterprise applications, including web applications, enterprise resource planning (ERP) systems, customer relationship management (CRM) systems, and more. Its ability to integrate with other Microsoft technologies, such as .NET, Power BI, and Azure, makes it a popular choice for organizations already using the Microsoft ecosystem.

MySQL

MySQL is an open-source relational database management system owned by Oracle Corporation. It is one of the most popular DBMSs for web applications due to its speed, reliability, and ease of use. MySQL supports SQL as its query language and is widely used with popular web development stacks such as LAMP (Linux, Apache, MySQL, PHP/Perl/Python) and MERN (MongoDB, Express, React, Node.js). MySQL is favored for its lightweight nature and ease of integration with platforms like WordPress, Joomla, and Drupal.

While SQL Server is typically used in enterprise environments, MySQL is favored for smaller-scale applications, especially in open-source and web development communities. It also supports a wide range of operating systems, including Linux, Windows, and macOS, making it highly versatile.

Key Differences Between SQL Server and MySQL

- **Licensing**: SQL Server is proprietary software, while MySQL is open-source.
- **Platform Support**: SQL Server is primarily used in Windows environments, although it supports Linux. MySQL is cross-platform, supporting various operating systems.
- **Scalability and Performance**: SQL Server is known for its advanced features and scalability, making it suitable for large enterprise applications. MySQL, while capable of handling large datasets, is often used for smaller applications but can be scaled through clustering and replication.

These differences can influence the choice of DBMS for a specific application depending on factors like budget, platform preference, and scale of operations.

Understanding Tables, Views, Stored Procedures

In relational databases, data is stored in tables, which are structured collections of rows and columns. Additionally, database systems support the use of views and stored procedures, which provide ways to manage and manipulate data more effectively. Understanding these core components is crucial for working with databases, as they form the foundation of most database operations.

Tables

A **table** in a database is a collection of data organized in rows and columns. Each row in a table represents a record, and each column represents a field within that record.

Tables are fundamental building blocks in a relational database, and they are where actual data is stored.

For example, a Books table might contain the following columns: BookID, Title, Author, Publisher, and Price. Each row in the table would represent a book, with its associated data.

Example Table: Books

BookID	Title	Author	Publisher	Price
1	"The Great Gatsby"	F. Scott	Scribner	10.99
2	"1984"	George Orwell	Secker & Warburg	8.99
3	"To Kill a Mockingbird"	Harper Lee	J.B. Lippincott & Co.	12.99

Tables must follow certain rules to maintain data integrity, such as ensuring that each record is unique (typically through a primary key) and preventing invalid data entries (through constraints like foreign keys, not null, etc.).

Views

A **view** in a database is a virtual table that is based on the result of a query. It is not a physical table; rather, it presents data from one or more tables in a specific way. Views simplify complex queries by encapsulating the query logic, and they can also be used for data security by restricting access to sensitive data. When you query a view, the database engine executes the underlying SQL query and returns the result as if it were a table.

For instance, a view can be created to only display certain columns from the Books table, such as just the Title and Price columns, excluding sensitive information like the Author or Publisher.

Example View: BookTitlesOnly

```
CREATE VIEW BookTitlesOnly AS

SELECT Title, Price

FROM Books;
```

You can now use the BookTitlesOnly view in a SELECT query just like a table:

```
SELECT * FROM BookTitlesOnly;
```

Views are useful for simplifying queries, reusing logic, and restricting data access.

Stored Procedures

A **stored procedure** is a precompiled collection of SQL statements that can be executed as a single unit. Stored procedures allow you to encapsulate complex logic and operations that can be reused multiple times. They can accept parameters, perform operations, and return results.

Stored procedures are especially useful for performing repetitive tasks or encapsulating business logic in the database. By using stored procedures, you can improve performance (since the SQL statements are precompiled) and enhance security (since users can be restricted to executing predefined procedures rather than arbitrary SQL queries).

Example Stored Procedure: AddNewBook

```
CREATE PROCEDURE AddNewBook
  @Title NVARCHAR(100),
  @Author NVARCHAR(100),
  @Publisher NVARCHAR(100),
  @Price DECIMAL(5, 2)
AS
BEGIN
  INSERT INTO Books (Title, Author, Publisher, Price)
  VALUES (@Title, @Author, @Publisher, @Price);
END;
```

This stored procedure inserts a new book into the Books table. You can call the stored procedure as follows:

```
EXEC AddNewBook 'Brave New World', 'Aldous Huxley', 'Harper & Row', 15.99;
```

Stored procedures improve code maintainability and security by abstracting complex logic from the application layer and into the database.

EXERCISES AND CASE STUDY

Exercise:

1. **Create a Table:** Design a Students table with the following columns: StudentID, FirstName, LastName, DOB, Email. Write the SQL statement to create this table.

2. **Create a View:** Create a view that displays the Title and Price of books from the Books table, excluding the Author and Publisher information.

3. **Create a Stored Procedure:** Write a stored procedure that updates the price of a book based on the BookID and new Price values.

Case Study:

Company XYZ is planning to build an online bookstore and needs to create a relational database to manage their inventory. The system needs to store information about books, authors, and publishers. Your task is to:

1. Design the necessary tables for books, authors, and publishers.

2. Create a view that combines data from these tables to display a list of books with their respective authors and publishers.

3. Write a stored procedure to add a new book to the database, taking input for the title, author, publisher, and price.

CONNECTING ASP.NET CORE TO DATABASES

ENTITY FRAMEWORK CORE

Entity Framework Core (EF Core) is a lightweight, extensible, and open-source version of Entity Framework that works with ASP.Net Core to facilitate interaction between an application and a database. EF Core is an Object-Relational Mapping (ORM) framework, which means it allows developers to interact with databases using .NET objects, thus abstracting the underlying SQL queries.

EF Core enables you to work with databases using strongly typed objects instead of writing raw SQL queries. The main advantage of using EF Core is its ability to automatically generate SQL queries and manage database connections, making database operations easier to handle, more secure, and more scalable.

Setting Up EF Core in ASP.Net Core
To use EF Core in your ASP.Net Core application, follow these steps:

1. **Install EF Core NuGet Packages**:
   You first need to install the EF Core NuGet packages for your project. If you are working with SQL Server, you can install the required package by running the following command in the **NuGet Package Manager Console**:

2. Install-Package Microsoft.EntityFrameworkCore.SqlServer

You might also need the tools for migrations and commands:

Install-Package Microsoft.EntityFrameworkCore.Tools

3. **Create a Database Context**:
   The database context class is the key class in EF Core. It acts as a bridge between your application and the database. This class inherits from DbContext and defines properties for each table in the database as DbSet<TEntity>.
   Example:

4. public class ApplicationDbContext : DbContext {

5. public DbSet<Customer> Customers { get; set; }

6. public DbSet<Order> Orders { get; set; }

```
7.
8.     public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
       options)
9.         : base(options) { }
10. }
```

Here, Customer and Order are entities that represent tables in your database, and EF Core will automatically create SQL queries to interact with those tables.

11. **Configure Database Connection**:
    In the Startup.cs file, you need to configure the database connection string and set up EF Core services in the ConfigureServices method:

```
12. public void ConfigureServices(IServiceCollection services) {
13.     services.AddDbContext<ApplicationDbContext>(options =>
14.
       options.UseSqlServer(Configuration.GetConnectionString("DefaultConnectio
       n")));
15. }
```

In your appsettings.json, add the connection string for your database:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Server=(localdb)\\mssqllocaldb;Database=ApplicationDb;Trusted_Connection=True;
"
  }
}
```

Entity Framework Core Querying

Once EF Core is set up, you can use LINQ (Language Integrated Query) to interact with your database. For example, to retrieve all customers from the Customers table:

```
var customers = await _context.Customers.ToListAsync();
```

You can also use more complex queries such as filtering, sorting, and joining tables using LINQ syntax. EF Core will translate these LINQ queries into appropriate SQL statements that interact with the database.

Advantages of EF Core:

- **Productivity**: EF Core allows developers to work with strongly typed objects, removing the need to write raw SQL.
- **Database Independence**: EF Core supports different database providers such as SQL Server, PostgreSQL, SQLite, MySQL, etc.
- **Migrations and Schema Management**: EF Core has built-in support for database migrations, making it easy to evolve the database schema as your application develops.

Database Migrations and Schema Generation

Database Migrations are a way to keep your database schema in sync with your application's data model over time. EF Core provides a migration system that can automatically generate and apply changes to your database schema as your models change. This process allows you to manage database changes in a structured and version-controlled manner.

Using Migrations in EF Core
Migrations allow you to update your database schema without manually writing SQL scripts. Here's how you can create and apply migrations:

1. **Add Migration**:
   After making changes to your models, run the following command to create a migration. This command compares your current data model with the database schema and generates the necessary SQL to bring the database up to date:

2. Add-Migration InitialCreate

This will create a new migration file in the Migrations folder, containing the SQL commands to create or update the database schema.

3. **Apply Migration**:
   Once the migration is added, you need to apply it to the database. You can do this using the Update-Database command:

4. Update-Database

This command will apply any pending migrations to the database, creating or modifying tables, columns, constraints, etc.

5. **View and Customize Migrations**:
   Each migration is stored in a separate file, and you can customize the SQL that will be generated. For example, you can add custom columns or modify constraints manually within the migration class:

```
public partial class AddEmailToCustomer : Migration {
    protected override void Up(MigrationBuilder migrationBuilder) {
        migrationBuilder.AddColumn<string>(
            name: "Email",
            table: "Customers",
            nullable: true);
    }
    protected override void Down(MigrationBuilder migrationBuilder) {
        migrationBuilder.DropColumn(
            name: "Email",
            table: "Customers");
    }
}
```

In this example, we add an Email column to the Customers table and provide a Down method to remove it if the migration is ever rolled back.

Handling Migrations in Development and Production
While migrations are useful for managing changes during development, special care should be taken when deploying to production. It's best practice to:

- **Test Migrations**: Always test your migrations in a development or staging environment before applying them to production.
- **Version Control**: Ensure that migration files are committed to version control to track changes and avoid conflicts across different environments.

- **Rollbacks**: Be prepared to rollback migrations if necessary. EF Core allows you to revert to previous versions of the schema using the Remove-Migration and Update-Database commands.

Schema Generation and Updating

EF Core allows you to generate the schema automatically by examining your model classes. As you update your data models (e.g., adding or removing properties), EF Core can generate the SQL required to reflect these changes in the database schema. This is especially useful in an agile development environment where the database schema evolves rapidly as the application grows.

Generating and Updating Database Schema

Whenever you update your model, you need to create a new migration and apply it:

1. Modify your models (e.g., add a new property to an entity class).
2. Run the Add-Migration command.
3. Apply the migration with Update-Database.

Handling Seed Data

In addition to creating tables, EF Core allows you to seed data into your database automatically. You can define seed data within the OnModelCreating method of your DbContext class.

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {

  modelBuilder.Entity<Customer>().HasData(

    new Customer { Id = 1, Name = "John Doe" },

    new Customer { Id = 2, Name = "Jane Smith" }

  );

}
```

This code will insert two customers into the Customer table when the database is created or updated.

EXERCISE

1. **Create a New Migration**: Add a new property to the Customer class (e.g., Email). Run the migration commands to generate and apply the changes to your database.

2. **Seed Data**: Implement a seeding process to add sample customers to the Customer table during the database initialization.

Case Study: Building a Simple E-Commerce Application

In this case study, you will build a **Simple E-Commerce Application** using ASP.Net Core and Entity Framework Core. The application should allow users to view products, add products to their cart, and make a purchase.

1. **Entities**: You will have Product, Category, and Order classes representing the database tables.
2. **Database Schema**: Use migrations to create and manage the database schema as you add new features to the application.
3. **Seed Data**: Implement seed data to populate the database with sample products and categories.
4. **Database Updates**: As new features are added (e.g., adding a discount or new payment methods), use EF Core migrations to evolve the database schema without disrupting the existing application.

CRUD Operations with ASP.Net and SQL Server

Creating, Reading, Updating, and Deleting Data in Databases

CRUD operations (Create, Read, Update, Delete) are the fundamental actions required to interact with a database. In the context of ASP.Net and SQL Server, these operations can be easily performed using the Entity Framework or ADO.Net, which are the primary data access technologies in ASP.Net applications. Mastering CRUD operations is essential for any developer working with databases, as it forms the core functionality for most data-driven applications.

Creating Data in SQL Server with ASP.Net

To create data in a database, you typically use an SQL INSERT statement to add new records to a table. In ASP.Net, you can execute SQL queries using ADO.Net or Entity Framework, both of which provide methods to interact with the database. Using ADO.Net, you can open a connection to SQL Server, execute an INSERT statement, and add a new record to a table.

Example of adding a new record using ADO.Net:

```
using (SqlConnection conn = new SqlConnection(connectionString))
{
   string query = "INSERT INTO Books (Title, Author, Price) VALUES (@Title, @Author, @Price)";
   SqlCommand cmd = new SqlCommand(query, conn);
   cmd.Parameters.AddWithValue("@Title", "New Book");
   cmd.Parameters.AddWithValue("@Author", "Author Name");
   cmd.Parameters.AddWithValue("@Price", 19.99);
```

```
    conn.Open();
    cmd.ExecuteNonQuery();
}
```

In Entity Framework, creating data is done by adding an object to the DbSet collection, and then calling SaveChanges() to commit the data to the database.

Example of creating a new book using Entity Framework:

```
using (var context = new BookStoreContext())
{
    var newBook = new Book
    {
        Title = "New Book",
        Author = "Author Name",
        Price = 19.99
    };

    context.Books.Add(newBook);
    context.SaveChanges();
}
```

Both approaches allow you to create new records in the database, but Entity Framework offers higher-level abstractions, reducing the need to write raw SQL queries and improving maintainability.

Reading Data from SQL Server with ASP.Net

Reading data from a database is achieved using SQL SELECT statements, and in ASP.Net, you can retrieve this data using ADO.Net or Entity Framework. When querying data, you often need to display it on a web page or process it further.

Example of reading data using ADO.Net:

```
using (SqlConnection conn = new SqlConnection(connectionString))
{
   string query = "SELECT * FROM Books";
   SqlCommand cmd = new SqlCommand(query, conn);

   conn.Open();
   SqlDataReader reader = cmd.ExecuteReader();

   while (reader.Read())
   {
      Console.WriteLine(reader["Title"].ToString());
   }
}
```

Using Entity Framework, you can retrieve data more efficiently by querying the DbSet directly:

```
using (var context = new BookStoreContext())
{
   var books = context.Books.ToList();

   foreach (var book in books)
   {
      Console.WriteLine(book.Title);
   }
}
```

In both cases, you retrieve data from SQL Server and use it in your application. The Entity Framework allows for more complex querying through LINQ (Language Integrated Query), which provides a strongly typed, object-oriented way of interacting with data.

UPDATING DATA IN SQL SERVER WITH ASP.NET

Updating existing data is performed using the SQL UPDATE statement. In ASP.Net, whether using ADO.Net or Entity Framework, you can modify the data and then commit the changes to the database.

Example of updating data using ADO.Net:

```
using (SqlConnection conn = new SqlConnection(connectionString))
{
   string query = "UPDATE Books SET Price = @Price WHERE Title = @Title";
   SqlCommand cmd = new SqlCommand(query, conn);
   cmd.Parameters.AddWithValue("@Price", 29.99);
   cmd.Parameters.AddWithValue("@Title", "Existing Book");

   conn.Open();
   cmd.ExecuteNonQuery();
}
```

Example using Entity Framework:

```
using (var context = new BookStoreContext())
{
   var book = context.Books.FirstOrDefault(b => b.Title == "Existing Book");
   if (book != null)
   {
      book.Price = 29.99;
```

```
        context.SaveChanges();
    }
}
```

With Entity Framework, after retrieving the object, you modify its properties directly and call SaveChanges() to persist the changes back to the database.

DELETING DATA IN SQL SERVER WITH ASP.NET

Deleting data is performed using the SQL DELETE statement. Both ADO.Net and Entity Framework allow you to remove data from a database.

Example of deleting data using ADO.Net:

```
using (SqlConnection conn = new SqlConnection(connectionString))
{
    string query = "DELETE FROM Books WHERE Title = @Title";
    SqlCommand cmd = new SqlCommand(query, conn);
    cmd.Parameters.AddWithValue("@Title", "Book to Delete");

    conn.Open();
    cmd.ExecuteNonQuery();
}
```

In Entity Framework, deleting data involves finding the entity you want to remove and calling Remove():

```
using (var context = new BookStoreContext())
{
    var book = context.Books.FirstOrDefault(b => b.Title == "Book to Delete");
    if (book != null)
    {
```

```
        context.Books.Remove(book);

        context.SaveChanges();

    }

}
```

Deleting data from the database is a critical operation that should be performed with caution, as it permanently removes records from the table.

LINQ Queries and Data Binding

LINQ (Language Integrated Query) is a powerful feature in C# that allows you to write SQL-like queries directly in C# code. It provides a convenient way to query collections, databases, XML, and other data sources in a type-safe manner. LINQ queries are often used in ASP.Net applications to interact with data retrieved from databases, and they offer a more readable and maintainable alternative to raw SQL queries.

LINQ Queries in ASP.Net

LINQ queries can be executed on collections, arrays, or objects that implement IEnumerable. In the context of databases, LINQ queries are typically used with Entity Framework, which maps database tables to C# objects. LINQ queries are written using the LINQ syntax, which can be either query syntax or method syntax.

Example of LINQ Query (Method Syntax):

```
using (var context = new BookStoreContext())

{

    var books = context.Books

                .Where(b => b.Price > 20)

                .OrderBy(b => b.Title)

                .ToList();
```

```
    foreach (var book in books)
    {
        Console.WriteLine(book.Title);
    }
}
```

Example of LINQ Query (Query Syntax):

```
using (var context = new BookStoreContext())
{
    var books = from b in context.Books
                where b.Price > 20
                orderby b.Title
                select b;

    foreach (var book in books)
    {
        Console.WriteLine(book.Title);
    }
}
```

LINQ queries provide a powerful and flexible way to interact with data, offering features like filtering (Where), ordering (OrderBy), grouping, and joining collections, which makes data manipulation much easier.

DATA BINDING IN ASP.NET

Data binding is the process of connecting a data source (e.g., a database, collection, or object) to a UI component, such as a grid, list, or form. In ASP.Net, data binding allows you to display data dynamically and provide a seamless interaction between the user interface and the underlying data model.

For example, when working with a GridView or ListView, you can bind data to these controls using LINQ queries.

Example of Data Binding with GridView:

```
protected void Page_Load(object sender, EventArgs e)
{
  using (var context = new BookStoreContext())
  {
    var books = context.Books.ToList();
    GridView1.DataSource = books;
    GridView1.DataBind();
  }
}
```

In this example, the GridView control is bound to a list of books retrieved from the database using LINQ. Whenever the data changes, you can call DataBind() again to refresh the data displayed in the control.

EXERCISES AND CASE STUDY

Exercise:

1. **Create CRUD Operations for Students:**
    - Create a Student table with columns: StudentID, Name, Age, and Grade.
    - Implement Create, Read, Update, and Delete operations for the Student table using both ADO.Net and Entity Framework.
    - Display the list of students in a GridView and enable functionality to add, edit, and delete students.
2. **LINQ Query Exercise:**

- Write a LINQ query to display a list of books whose price is between $10 and $30. Sort the books by Author and display the result in a table.

Case Study:

Company ABC is developing an inventory management system that needs to manage a list of products. The system must allow users to add, view, update, and delete products. You are tasked with building the CRUD functionality for the Product table using ASP.Net and SQL Server. You will also need to implement LINQ queries to filter and sort products based on certain criteria, such as price and category.

Your task:

- Design the Product table with appropriate columns.
- Implement CRUD operations for products.
- Use LINQ to filter products based on their price range and category.
- Bind the data to a GridView or ListView control for display.

Assignment Solution: Build a User Management System with CRUD Functionality Using SQL Server and ASP.Net Core

In this assignment, we will build a basic **User Management System** using **ASP.Net Core** and **SQL Server**. The system will include **CRUD (Create, Read, Update, Delete)** operations to manage users and their information. The application will interact with a **SQL Server database** using **Entity Framework Core** (EF Core).

Step-by-Step Guide

Prerequisites

Ensure you have the following installed:

1. **Visual Studio** with the **ASP.Net and Web Development** workload.
2. **SQL Server** or **SQL Server Express** installed on your machine. If you don't have it installed, you can download **SQL Server Express** from the Microsoft website.
3. **.NET SDK** installed on your machine.
4. **SQL Server Management Studio (SSMS)** for managing your SQL Server database.

Step 1: Set Up the Database in SQL Server

1. **Create a new Database**:
    - Open **SQL Server Management Studio (SSMS)** and connect to your SQL Server instance.
    - Create a new database called UserManagementDB by running the following SQL query:
2. CREATE DATABASE UserManagementDB;
3. GO

4. **Create a User Table**:
    - In the UserManagementDB database, create a Users table with the following SQL query:

```
USE UserManagementDB;
GO

CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY(1,1),
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Email NVARCHAR(100) UNIQUE,
    DateOfBirth DATE
);
GO
```

16. **Insert Sample Data** (optional for testing):

```
INSERT INTO Users (FirstName, LastName, Email, DateOfBirth)
VALUES ('John', 'Doe', 'john.doe@example.com', '1985-05-15'),
       ('Jane', 'Smith', 'jane.smith@example.com', '1990-08-25');
GO
```

Step 2: Set Up ASP.Net Core Project

1. **Create a New ASP.Net Core Web Application**:
    - Open **Visual Studio** and select **Create a New Project**.
    - Choose **ASP.NET Core Web Application**, and click **Next**.
    - Name your project UserManagementSystem and choose the location to save it.

- Select **Web Application (Model-View-Controller)** and click **Create**.
- Make sure to select **.NET 6** or later as the framework.

2. **Install Entity Framework Core Packages**:
   - Open the **NuGet Package Manager Console** and install the required EF Core packages:

3. Install-Package Microsoft.EntityFrameworkCore.SqlServer
4. Install-Package Microsoft.EntityFrameworkCore.Tools
5. **Configure Connection String**:
   - Open the appsettings.json file and add the connection string for your SQL Server database:

6. {
7. "ConnectionStrings": {
8. "DefaultConnection": "Server=localhost;Database=UserManagementDB;Trusted_Connection=True;"
9. },
10. ...
11. }

Step 3: Set Up the Entity Framework Core Model

1. **Create a User Model**:
   - In your project, create a new folder named **Models** and add a new class User.cs:

2. public class User {
3. public int Id { get; set; }
4. public string FirstName { get; set; }
5. public string LastName { get; set; }

```
6.      public string Email { get; set; }
7.      public DateTime DateOfBirth { get; set; }
8.  }
```

9. **Create the DbContext**:
    - In the **Data** folder (or create one), add a new class ApplicationDbContext.cs to set up the database context:

```
10. using Microsoft.EntityFrameworkCore;
11.
12. public class ApplicationDbContext : DbContext {
13.     public ApplicationDbContext(DbContextOptions<ApplicationDbContext>
    options) : base(options) { }
14.
15.     public DbSet<User> Users { get; set; }
16. }
```

17. **Configure DbContext in Startup**:
    - Open Startup.cs (or Program.cs in .NET 6) and configure the ApplicationDbContext in the ConfigureServices method:

```
18. public void ConfigureServices(IServiceCollection services) {
19.     services.AddDbContext<ApplicationDbContext>(options =>
20.
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnectio
    n")));
21.
22.     services.AddControllersWithViews();
23. }
```

STEP 4: CREATE THE CRUD OPERATIONS

1. **Create a UserController**:
    - In the **Controllers** folder, create a new controller UserController.cs for handling the CRUD operations.

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using UserManagementSystem.Models;

public class UserController : Controller {
    private readonly ApplicationDbContext _context;

    public UserController(ApplicationDbContext context) {
        _context = context;
    }

    // Create User
    [HttpGet]
    public IActionResult Create() {
        return View();
    }

    [HttpPost]
    public async Task<IActionResult> Create(User user) {
        if (ModelState.IsValid) {
            _context.Add(user);
            await _context.SaveChangesAsync();
```

```
24.         return RedirectToAction(nameof(Index));
25.     }
26.     return View(user);
27. }
28.
29. // Read Users
30. public async Task<IActionResult> Index() {
31.     var users = await _context.Users.ToListAsync();
32.     return View(users);
33. }
34.
35. // Edit User
36. [HttpGet]
37. public async Task<IActionResult> Edit(int? id) {
38.     if (id == null) {
39.         return NotFound();
40.     }
41.
42.     var user = await _context.Users.FindAsync(id);
43.     if (user == null) {
44.         return NotFound();
45.     }
46.     return View(user);
47. }
```

```
48.
49.  [HttpPost]
50.  public async Task<IActionResult> Edit(int id, User user) {
51.    if (id != user.Id) {
52.      return NotFound();
53.    }
54.
55.    if (ModelState.IsValid) {
56.      try {
57.        _context.Update(user);
58.        await _context.SaveChangesAsync();
59.      } catch (DbUpdateConcurrencyException) {
60.        if (!_context.Users.Any(u => u.Id == user.Id)) {
61.          return NotFound();
62.        }
63.        throw;
64.      }
65.      return RedirectToAction(nameof(Index));
66.    }
67.    return View(user);
68.  }
69.
70.  // Delete User
71.  [HttpGet]
```

```
72.  public async Task<IActionResult> Delete(int? id) {
73.      if (id == null) {
74.          return NotFound();
75.      }
76.
77.      var user = await _context.Users
78.          .FirstOrDefaultAsync(m => m.Id == id);
79.      if (user == null) {
80.          return NotFound();
81.      }
82.
83.      return View(user);
84.  }
85.
86.  [HttpPost, ActionName("Delete")]
87.  public async Task<IActionResult> DeleteConfirmed(int id) {
88.      var user = await _context.Users.FindAsync(id);
89.      _context.Users.Remove(user);
90.      await _context.SaveChangesAsync();
91.      return RedirectToAction(nameof(Index));
92.  }
93. }
```

94. **Create Views for CRUD Operations**:

    - **Create.cshtml**:

```
@model UserManagementSystem.Models.User

<h2>Create User</h2>
<form method="post">
    <div>
        <label>First Name</label>
        <input type="text" name="FirstName" value="@Model.FirstName" />
    </div>
    <div>
        <label>Last Name</label>
        <input type="text" name="LastName" value="@Model.LastName" />
    </div>
    <div>
        <label>Email</label>
        <input type="email" name="Email" value="@Model.Email" />
    </div>
    <div>
        <label>Date of Birth</label>
        <input type="date" name="DateOfBirth" value="@Model.DateOfBirth.ToString("yyyy-MM-dd")" />
    </div>
    <button type="submit">Create</button>
</form>
```

- **Index.cshtml**:

```
@model IEnumerable<UserManagementSystem.Models.User>

<h2>User List</h2>
<table>
  <thead>
    <tr>
      <th>First Name</th>
      <th>Last Name</th>
      <th>Email</th>
      <th>Date of Birth</th>
      <th>Actions</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var user in Model) {
      <tr>
        <td>@user.FirstName</td>
        <td>@user.LastName</td>
        <td>@user.Email</td>
        <td>@user.DateOfBirth.ToString("yyyy-MM-dd")</td>
        <td>
          <a href="@Url.Action("Edit", "User", new { id = user.Id })">Edit</a>
          |
```

```
140.                     <a href="@Url.Action("Delete", "User", new { id = user.Id
    })">Delete</a>
141.                 </td>
142.             </tr>
143.         }
144.     </tbody>
145. </table>
```

- **Edit.cshtml** and **Delete.cshtml**: Similar to Create.cshtml, these views will display forms for editing and deleting users.

Step 5: Running the Application

1. **Build the Application**:
   - In Visual Studio, press **Ctrl + Shift + B** to build the solution.
2. **Run the Application**:
   - Press **F5** or click **Start** to run the application.
   - Navigate to /User/Index to see the list of users.
   - Use the navigation to **Create**, **Edit**, or **Delete** users.

Conclusion

This guide walks you through building a basic **User Management System** with **CRUD functionality** using **ASP.Net Core** and **SQL Server**. By following the steps above, you created a database, built models, set up Entity Framework Core, and implemented the CRUD operations in your application. You now have a working system for managing users within a SQL Server database.