Implementing CRUD Operations in EF Core

Objective: by the end of this activity, you will be able to perform basic CRUD (Create, Read, Update, Delete) operations using Entity Framework Core (EF Core) with MySQL Server in a .NET console application on a Linux Ubuntu system.

Step 1: Prepare for the Application

You'll create a new .NET console application and set up EF Core to manage a MySQL database for products.

Instructions:

1. **Set Up the Project in VS Code:**
    a. Open the project in Visual Studio Code: dotnet new console -n CRUDWithMySQL cd CRUDWithMySQL
    b. Open a terminal in your Ubuntu system and create a new .NET console application: code .
2. **Install Required EF Core Packages:**
    a. Install the MySQL EF Core provider and tools package: dotnet add package Pomelo.EntityFrameworkCore.MySql dotnet add package Microsoft.EntityFrameworkCore.Tools --version 8.0.2 dotnet new tool-manifest dotnet tool install dotnet-ef
3. **Add a Models Folder and Class Files:**
    a. Inside your project directory, create a folder named Models.
    b. Inside Models, add two files:
        i. Product.cs for the data model.
        ii. ApplicationDbContext.cs for the EF Core context.
4. **Create a New Database:**
    a. Open the MySQL CLI: mysql -u root -p
    b. Enter the password: password when prompted.
    c. Create a new database: CREATE DATABASE ProductDB;

Step 2: Defining the Data Model

Define a Product entity for the database.

Instructions:

1. In the Models/Product.cs file, define a class Product with:
    - Id as an integer (primary key).
    - Name as a string.
    - Price as a decimal.

Step 3: Setting up the Database Context

Set up the EF Core context to manage the Product entity.

Instructions:

1. In Models/ApplicationDbContext.cs, create a class ApplicationDbContext that inherits from DbContext.
2. Add a DbSet<Product> property to manage the Products table.
3. Configure the MySQL connection string in the OnConfiguring method. Use the following format: optionsBuilder.UseMySql("Server=localhost;Database=ProductDB;User=root;Password=password;", new MySqlServerVersion(new Version(8, 0, 26)));

Step 4: Initializing the Database

Create and apply a migration to generate the database schema.

Instructions:

1. Add and apply migrations: dotnet ef migrations add InitialCreate dotnet ef database update
2. Confirm the database schema has been updated in the ProductDB database.

Step 5: Implementing CRUD Operations

Create:

1. **Write code to:**
    - Create a new Product object.

    - Add it to the Products DbSet using the Add method.
    - Save changes to the database.

Read:

- Query the Products DbSet to:
    - Retrieve all products using the ToList method.
    - Retrieve a single product by its Id using the Find method.

Update:

- Modify an existing product:
    - Retrieve it using the Find method.
    - Change one of its properties (e.g., Name or Price).
    - Save changes using SaveChanges.

Delete:

- Remove a product:
    - Retrieve it using the Find method.
    - Use the Remove method.
    - Save changes to the database.

To test the program, run dotnet run from the terminal.

Product.cs:

```
namespace CRUDWithMySQL.Models;

public class Product
{
    public int Id { get; set; }
    public string Name { get; set; } = "";
    public decimal Price { get; set; }
}
```

ApplicationDbContext.cs:

```
using Microsoft.EntityFrameworkCore;
using CRUDWithMySQL.Models;

namespace CRUDWithMySQL.Data;

public class ApplicationDbContext : DbContext
{
    public DbSet<Product> Products { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder
optionsBuilder)
    {
        var connectionString =

"Server=localhost;Database=ProductDB;User=efuser;Password=efpassword;";

        optionsBuilder.UseMySql(
            connectionString,
            new MySqlServerVersion(new Version(8, 0, 36))
        );
    }
}
```

Program.cs:

```
using System.Globalization;
using CRUDWithMySQL.Data;
using CRUDWithMySQL.Models;

var mdl = new CultureInfo("ro-MD");

using (var context = new ApplicationDbContext())
{
    // === CREATE ===
    var newProduct = new Product { Name = "Laptop", Price = 1200.50m };
    context.Products.Add(newProduct);
    context.SaveChanges();
    Console.WriteLine($"☑ Product created: {newProduct.Name}");

    // === READ ===
    var products = context.Products.ToList();
    Console.WriteLine("\n📋 All products:");
```

```
    foreach (var p in products)
        Console.WriteLine($"{p.Id}: {p.Name} - {p.Price.ToString("C",
mdl)}");

    // === READ ===
    var product = context.Products.Find(newProduct.Id);
    if (product != null)
        Console.WriteLine($"\n🔍 Found product: {product.Name} -
{product.Price.ToString("C", mdl)}");

    // === UPDATE ===
    if (product != null)
    {
        product.Price = 999.99m;
        context.SaveChanges();
        Console.WriteLine($"\n🖊 Updated product: {product.Name} -
{product.Price.ToString("C", mdl)}");
    }

    // === DELETE ===
    if (product != null)
    {
        context.Products.Remove(product);
        context.SaveChanges();
        Console.WriteLine($"\n🗑 Deleted product: {product.Name}");
    }

    Console.WriteLine("\n📋 Final list:");
    foreach (var p in context.Products.ToList())
        Console.WriteLine($"{p.Id}: {p.Name} - {p.Price.ToString("C",
mdl)}");
}
```