**1. MVC Coding**

**What is MVC Coding?**

MVC (Model-View-Controller) is a software architectural pattern that separates an application into three main components:

* **Model**: Represents the data and the business logic.
* **View**: Represents the user interface and the presentation of data.
* **Controller**: Handles user input, interacts with the model, and renders the appropriate view.

**How Did I Solve This?**

In this project, I used ASP.NET Core MVC to implement the MVC pattern. The ChildrenController handles the logic for CRUD operations on the Children entity, interacting with the ChildrenContext (Model) and rendering views (View).

**Code Snippet - ChildrenController.cs:**

using DeveloperAssignment.Models;

using Microsoft.AspNetCore.Mvc;

namespace DeveloperAssignment.Controllers

{

public class ChildrenController : Controller

{

private readonly ChildrenContext \_context;

public ChildrenController(ChildrenContext context)

{

\_context = context;

}

// List all children

public IActionResult Index()

{

var children = \_context.Children.ToList();

return View(children);

}

// Show details of a specific child

public IActionResult Details(int id)

{

var child = \_context.Children.FirstOrDefault(c => c.Id == id);

if (child == null)

{

return NotFound();

}

return View(child);

}

// Show create form

public IActionResult Create()

{

return View();

}

// Handle create form submission

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Create(Child child)

{

if (ModelState.IsValid)

{

\_context.Children.Add(child);

\_context.SaveChanges();

return RedirectToAction(nameof(Index));

}

return View(child);

}

// Show edit form

public IActionResult Edit(int id)

{

var child = \_context.Children.FirstOrDefault(c => c.Id == id);

if (child == null)

{

return NotFound();

}

return View(child);

}

// Handle edit form submission

[HttpPost]

[ValidateAntiForgeryToken]

public IActionResult Edit(int id, Child child)

{

if (id != child.Id)

{

return NotFound();

}

if (ModelState.IsValid)

{

\_context.Update(child);

\_context.SaveChanges();

return RedirectToAction(nameof(Index));

}

return View(child);

}

// Show delete confirmation

public IActionResult Delete(int id)

{

var child = \_context.Children.FirstOrDefault(c => c.Id == id);

if (child == null)

{

return NotFound();

}

return View(child);

}

// Handle delete confirmation

[HttpPost, ActionName("Delete")]

[ValidateAntiForgeryToken]

public IActionResult DeleteConfirmed(int id)

{

var child = \_context.Children.FirstOrDefault(c => c.Id == id);

if (child == null)

{

return NotFound();

}

\_context.Children.Remove(child);

\_context.SaveChanges();

return RedirectToAction(nameof(Index));

}

}

}

#### ****Why Did I Do It Like This?****

Using the MVC pattern ensures a clean separation of concerns. The model manages data and business logic, the view handles the presentation, and the controller manages the flow of the application. This structure improves maintainability, scalability, and testability.

### ****2. EF Coding****

#### ****What is EF Coding?****

Entity Framework (EF) is an ORM (Object-Relational Mapping) framework that allows developers to interact with databases using .NET objects. It abstracts the database layer, making it easier to perform CRUD operations and manage database schemas.

#### ****How Did I Solve This?****

I used Entity Framework Core to interact with the SQL Server database. The ChildrenContext class represents the database context, and DbSet properties are used to perform CRUD operations on the Children entity.

**Code Snippet - ChildrenContext.cs:**

using Microsoft.EntityFrameworkCore;

using System.Collections.Generic;

namespace DeveloperAssignment.Models

{

public class ChildrenContext : DbContext

{

public DbSet<Child> Children { get; set; }

public ChildrenContext(DbContextOptions<ChildrenContext> options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.Entity<Child>().HasData(

new Child { Id = 1, Name = "John Doe", Age = 10 },

new Child { Id = 2, Name = "Jane Smith", Age = 8 }

);

}

}

}

#### ****Why Did I Do It Like This?****

Entity Framework Core simplifies database management by allowing developers to interact with the database using .NET objects. It reduces the amount of boilerplate code and provides built-in features like migrations and seeding, which help maintain the database schema and data integrity.

### ****3. SQL Query****

#### ****What is SQL Query?****

SQL (Structured Query Language) is used to manage and manipulate relational databases. It is the standard language for querying data, inserting new records, updating existing records, and deleting records.

#### ****How Did I Solve This?****

In cases where complex queries were required, I used raw SQL within Entity Framework Core. For example, to compare data between two databases, I used SQL EXCEPT and UNION operations.

**Code Snippet - SQL Query in DatabaseController.cs:**

public IActionResult CompareTables()

{

// First query against DatabaseA

string queryA = @"

SELECT \* FROM Children

EXCEPT

SELECT \* FROM DatabaseB.dbo.Children;";

var differencesA = \_contextA.Children.FromSqlRaw(queryA).ToList();

// Second query against DatabaseB

string queryB = @"

SELECT \* FROM Children

EXCEPT

SELECT \* FROM Children;";

var differencesB = \_contextB.Children.FromSqlRaw(queryB).ToList();

// Combine the results

var differences = differencesA.Concat(differencesB).ToList();

return View(differences);

}

#### ****Why Did I Do It Like This?****

While Entity Framework Core provides powerful querying capabilities through LINQ, certain complex operations, like set-based operations, are more efficiently executed using raw SQL. This approach leverages the strengths of both EF Core and SQL, ensuring optimal performance and flexibility.

### ****4. Unit Test / Debug****

#### ****What is Unit Test / Debug?****

Unit testing involves verifying that individual components of the software function correctly. Debugging is the process of identifying and fixing issues in the code.

#### ****How Did I Solve This?****

I implemented unit tests for key components of the application using xUnit. Debugging was done using Visual Studio's integrated debugger, which allowed me to set breakpoints, step through the code, and inspect variables.

**Code Snippet - Unit Test in CsvParserTests.cs:**

using DeveloperAssignment.Utilities;

using Microsoft.VisualStudio.TestTools.UnitTesting;

namespace DeveloperAssignment.Tests

{

[TestClass]

public class CsvParserTests

{

[TestMethod]

public void RemoveQuote\_ShouldRemoveQuotes()

{

Assert.AreEqual("hello", "\"hello\"".RemoveQuote());

Assert.AreEqual("world", "world\"".RemoveQuote());

Assert.AreEqual("test", "\"test".RemoveQuote());

Assert.AreEqual("", "".RemoveQuote());

Assert.AreEqual("example", "example".RemoveQuote());

}

}

}

**Code Snippet - Debugging Example in ChildrenController.cs:**

public IActionResult Create(Child child)

{

if (ModelState.IsValid)

{

\_context.Children.Add(child);

\_context.SaveChanges();

return RedirectToAction(nameof(Index));

}

return View(child);

}

#### ****Why Did I Do It Like This?****

Unit testing and debugging are essential for ensuring the correctness and reliability of the code. Writing unit tests helps catch errors early in the development process, while debugging allows for real-time analysis and problem-solving.

### ****5. Error Handling****

#### ****What is Error Handling?****

Error handling is the practice of anticipating and managing errors in software to prevent crashes and ensure a smooth user experience. It involves catching exceptions, providing meaningful error messages, and logging issues for further analysis.

#### ****How Did I Solve This?****

Error handling was implemented using try-catch blocks, custom error pages, and logging. In the database context, I used EnableRetryOnFailure to handle transient errors.

**Code Snippet - Error Handling in program.cs:**

builder.Services.AddDbContext<ChildrenContext>(options =>

options.UseSqlServer(builder.Configuration.GetConnectionString("ChildrenDB"),

sqlServerOptionsAction: sqlOptions =>

{

sqlOptions.EnableRetryOnFailure(

maxRetryCount: 5,

maxRetryDelay: TimeSpan.FromSeconds(30),

errorNumbersToAdd: null);

}));

**Code Snippet - Try-Catch Block in ChildrenController.cs:**

public IActionResult Edit(int id, Child child)

{

if (id != child.Id)

{

return NotFound();

}

if (ModelState.IsValid)

{

try

{

\_context.Update(child);

\_context.SaveChanges();

return RedirectToAction(nameof(Index));

}

catch (DbUpdateConcurrencyException)

{

ModelState.AddModelError("", "Unable to save changes due to concurrency conflict.");

}

}

return View(child);

}

#### ****Why Did I Do It Like This?****

Implementing robust error handling ensures that the application can gracefully recover from unexpected situations. By catching exceptions and providing user-friendly error messages, I improved the user experience and reduced the likelihood of application crashes.