**Repository Pattern in ASP.NET Core - Ultimate Guide**

In this extensive guide, we will go through everything you will need to know about Repository Pattern in ASP.NET Core, Generic Repository Patterns, Unit of Work and related topics. We will build a project right from scratch where we implement a clean architecture to access data. The source code of this implemenation is over at my [Github](https://github.com/iammukeshm/RepositoryPattern.WebApi).

**What’s a Repository Pattern?**

A Repository pattern is a design pattern that mediates data from and to the Domain and Data Access Layers ( like Entity Framework Core / Dapper). **Repositories are classes** that hide the logics required to store or retreive data. Thus, our application will not care about what kind of ORM we are using, as everything related to the ORM is handled within a repository layer. This allows you to have a cleaner seperation of concerns. Repository pattern is one of the heavily used Design Patterns to build cleaner solutions.

**Benefits of Repository Pattern**

**Reduces Duplicate Queries**

Imagine having to write lines of code to just fetch some data from your datastore. Now what if this set of queries are going to be used in multiple places in the application. Not very ideal to write this code over and over again, right? Here is the added advantage of Repository Classes. You could write your data access code within the Repository and call it from multiple Controllers / Libraries. Get the point?

**De-couples the application from the Data Access Layer**

There are quite a lot of ORMs available for ASP.NET Core. Currently the most popular one is Entity Framework Core. But that change in the upcoming years. To keep in pace with the evolving technologies and to keep our Solutions upto date, it is highly crucial to build applications that can switch over to a new DataAccessTechnology with minimal impact on our application’s code base.

There can be also cases where you need to use multiple ORMs in a single solution. Probably **Dapper** to fetch the data and EFCore to write the data. This is solely for performance optimizations.

Repository pattern helps us to achieve this by creating an Abstration over the DataAccess Layer. Now, you no longer have to depend on EFCore or any other ORM for your application. EFCore becomes one of your options rather than your only option to access data.

*The Architecture should be independent of the Frameworks.*

* *Uncle Bob ( Robert Cecil Martin )*

Building an Enterprise level ASP.NET Core Application would really need Repository Patterns to keep the codebase future proof for atleast the next 20-25 years (After which, probably the robots would take over :P ).

**Is Repository Pattern Dead?**

This is one of the most debated topics within the .NET Core Community. Microsoft has built the **Entity Framework Core using the Repository Pattern and Unit of Work Patterns.** So, why do we need to add another layer of abstraction over the Entity Framework Core, which is yet another abstration of Data Access. The answer to this is also given by Microsoft.

Read more here - <https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-implemenation-entity-framework-core#using-a-custom-repository-versus-using-ef-dbcontext-directly>

Microsoft themselves recommend using Repository Patterns in complex scenarios to reduce the coupling and provide better Testability of your solutions. In cases where you want the simplest possible code, you would want to avoid the Repository Pattern.

Custom Repository vs DBContext . Source : Microsoft

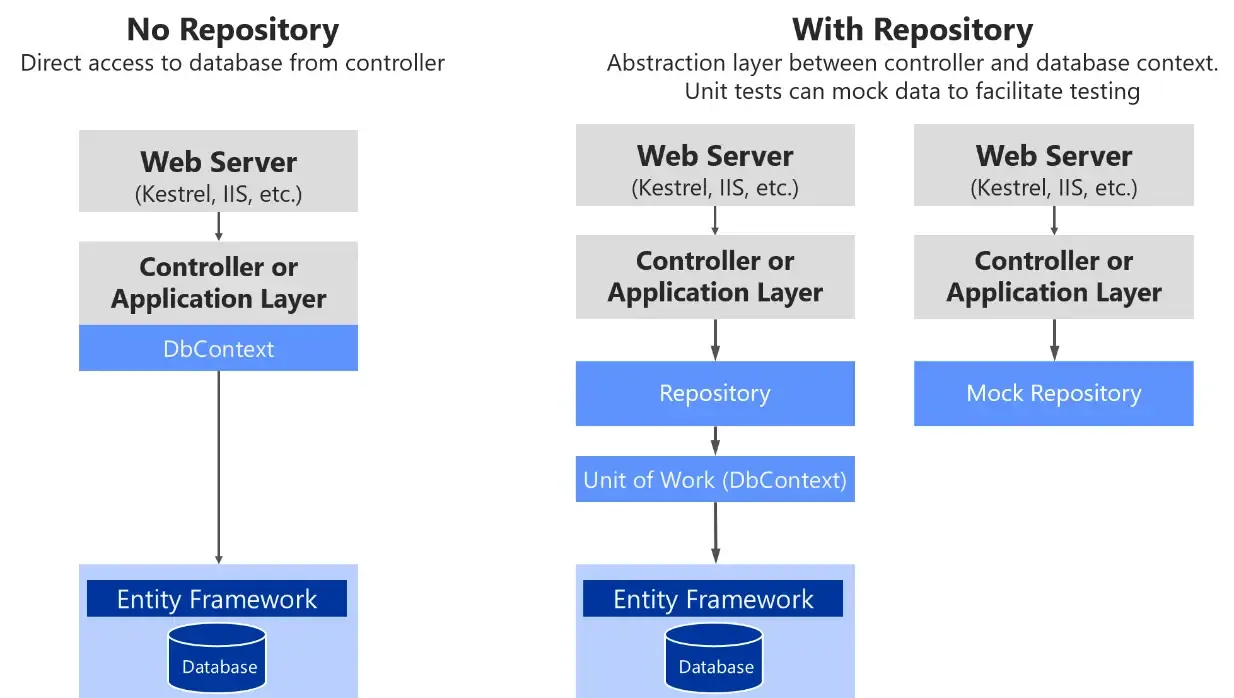
Adding the Repository has it’s own benefits too. But i strongly advice to not use Design Patterns everywhere. Try to use it only whenever the scenario demands the usage of a Design Pattern. That being stated, Repository pattern is something that can benefit you in the long run.

Implementing Repository Pattern in ASP.NET Core 3.1

Let’s implement Repository Pattern in an ASP.NET Core WebApi Project. What seperates this guide from the others is that we will also be working with a Clean Architecture in mind to demonstrate the real-life implementation. This means that we will be working with multiple Layers and Projects and also go through the basics of Dependency Inversion Principle.

*PS, I am using Visual Studio 2019 Community as my default IDE. Read*[*this article*](https://codewithmukesh.com/blog/install-visual-studio-2019-community/)*to install this awesome IDE on to your machine.*

Let’s start by creating a new Solution. Here I am naming my Solution as RepositoryPattern.WebApi and the first project as WebApi (ASP.NET Core).



Simalary, let’s add 2 more .NET Core Class Library Projects within the solution. We will call it DataAccess.EFCore and Domain. Here are the features and purposes of each project.

* Domain - Holds the Entities and Interfaces. It does not depend on any other project in the solution.
* DataAccess,EFCore - Since we will be using Entity Framework Core - Code First Apporach to interact with our Database, let’s build a project that solely represents everything related to EFCore. The aim is that, later down the road one can easily build another Data Access layer like DataAccess.Dapper or so. And our application would still support it. Here is where Dependency Inversion comes to play.
* WebApi - This is like the presentation layer of the entire solution. It depends on both the projects.

Here is how our Solution would look like now.

Setting up the Entities and EFCore

Now, let’s add the Required Entities to the Domain Project. Create a new Folder in the Domain Project named Entities.

Create 2 very simple classes - Developer and Project under the Entities Folder.

public class Developer

{

public int Id { get; set; }

public string Name { get; set; }

public int Followers { get; set; }

}

public class Project

{

public int Id { get; set; }

public string Name { get; set; }

}

Next , we will setup and configure Entity Framework Core. Install these Required Packages in the DataAccess.EFCore Project. Here is where we would have our DbContect Class and the actual Implementations of the Repositories.

Install-Package Microsoft.EntityFrameworkCore Install-Package Microsoft.EntityFrameworkCore.SqlServer

Add a reference to the Domain Project (where we have defined our entities) and create a new Class in the DataAccess.EFCore Project and Name it ApplicationContext.cs.

public class ApplicationContext : DbContext

{

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

{

}

public DbSet<Developer> Developers { get; set; }

public DbSet<Project> Projects { get; set; }

}

Once our Data Access Layer is done, let’s move to the WebApi Project to register EFCore within the ASP.NET Core Application. We will also update the database in this step to accomadate the Developer and Project Table.

Firstly, Install this package on the WebApi Project. This allows you to run EF Core commands on the CLI.

Install-Package Microsoft.EntityFrameworkCore.Tools

Next, Navigate to Startup.cs and add this line to Register the ApplicationContext class that we created. Note that you will have to add a refernce of the DataAccess.EFCore Project to the WebApi Project.

services.AddDbContext<ApplicationContext>(options =>

options.UseSqlServer(

Configuration.GetConnectionString("DefaultConnection"),

b => b.MigrationsAssembly(typeof(ApplicationContext).Assembly.FullName)));

After that, open up the appsettings.json file in the Api Project and add the connection string.

"ConnectionStrings": {

"DefaultConnection": "<your connection string>",

},

Finally, Let’s update the database. Open your Package Manager Console on Visual Studio and run the following commands.

add-migration Initial

update-database

Make sure that you have set your Startup Project as WebApi and the Default Project as DataAccess.EFCore. Here is a screenshot.

Ps, This is a very basic setup of Entity Framework Core. I have written a detailed Guide on [Entity Framework Core - Code First Apporach](https://codewithmukesh.com/blog/entity-framework-core-in-aspnet-core/). Give it a look to learn more.

Let’s Keep Repositories Away for a Moment.

Now that we have configured our EFCore Layer, let’s talk a bit about the traditional way to get data from this layer. Traditionally, you would directly call the dbContext object to read and write data. This is fine. But is it really ideal for the long run? When you use the dbContext directly, what you are doing is that, the Entity Framework Core is being tightly coupled within your application. So, Tommorow when there is something newer and better than EFCore, you would find it really annoying to implement the new tech and make the corresponding changes. Right?

One more disadvantage of directly using the dbContext directly is that you would be exposing the DbContext, which is totally insecure.

This is the reason to use Repository Pattern in ASP.NET Core Applications.

Practical Use-Case of Repositories

While Performing CRUD Operations with Entity Framework Core, you might have noticed that the basic essence of the code keeps the same. Yet we write it multiple times over and over. The CRUD Operations include Create, Read, Update, and Delete. So, why not have a class / interface setup where you can generalize each of these operations.

Building a Generic Repository

First up, let’s add a new folder Interfaces in our Domain Project. Why? Because, we will be inverting the dependencies, so that, you can define the interface in the Domain Project, but the implementation can be outside the Domain Project. In this case, the implementations will go the DataAccess.EFCore Project. Thus, your domain layer will not depends on anything, rather, the other layers tend to depend on the Domain Layer’s interface. This is a simple explanation of Dependency Inversion Principle. Pretty Cool, yeah?

Add a new interface, Interfaces/IGenericRepository.cs

public interface IGenericRepository<T> where T : class

{

T GetById(int id);

IEnumerable<T> GetAll();

IEnumerable<T> Find(Expression<Func<T, bool>> expression);

void Add(T entity);

void AddRange(IEnumerable<T> entities);

void Remove(T entity);

void RemoveRange(IEnumerable<T> entities);

}

This will be a Generic Interface, that can be used for both the Developer and Project Classes. Here T is the specific class.

The set of functions depends on your preference. Ideally, we require 7 functions that cover most of the data handling part.

1. Get’s the entity By Id.
2. Get’s all the Record.
3. Finds a set of record that matches the passed expression.
4. Adds a new record to the context
5. Add a list of records
6. Removes a record from the context
7. Removes a list of records.

Now, Let’s implement this Interfaces. Create a new class in the DataAccess.EFCore Project and name it Repositories/GenericRepository

public class GenericRepository<T> : IGenericRepository<T> where T : class

{

protected readonly ApplicationContext \_context;

public GenericRepository(ApplicationContext context)

{

\_context = context;

}

public void Add(T entity)

{

\_context.Set<T>().Add(entity);

}

public void AddRange(IEnumerable<T> entities)

{

\_context.Set<T>().AddRange(entities);

}

public IEnumerable<T> Find(Expression<Func<T, bool>> expression)

{

return \_context.Set<T>().Where(expression);

}

public IEnumerable<T> GetAll()

{

return \_context.Set<T>().ToList();

}

public T GetById(int id)

{

return \_context.Set<T>().Find(id);

}

public void Remove(T entity)

{

\_context.Set<T>().Remove(entity);

}

public void RemoveRange(IEnumerable<T> entities)

{

\_context.Set<T>().RemoveRange(entities);

}

}

This class will implement the IGenericRepository Interface. We will also inject the ApplicationContext here. This way we are hiding all the actions related to the dbContext object within Repository Classes. Also note that, for the ADD and Remove Functions, we just do the operation on the dbContext object. But we are not yet commiting/updating/saving the changes to the database whatsover. This is not something to be done in a Repository Class. We would need Unit of Work Pattern for these cases where you commit data to the database. We will discuss about Unit of Work in a later section.

Understood why we used a Generic Repository instead of a IDevloperRepository?? When there are large number of entites in our application, we would need seperate repositories for each entities. But we do not want to implement all of the above 7 Functions in each and every Repository Class, right? Thus we made a generic repository that holds the most commonly used implementaions.

Now what happens if we need the records of Most Popular Developers from our Database? We do not have a function for it in our Generic Class, do we ? This is where we can see the advantage of building a Generic Repository.

Inheriting and Extending the Generic Repository

In the Domain Project, under the Interfaces, add a new interface named IDeveloperRepository.

public interface IDeveloperRepository : IGenericRepository<Developer>

{

IEnumerable<Developer> GetPopularDevelopers(int count);

}

Here we are inheriting all the Functions of the Generic Repository, as well as adding a new Funciton ‘GetPopularDevelopers’. Get it?

Let’s implement the IDeveloperRepostory. Go to the DataAccess Project and under Repositories folder, add a new class, DeveloperRepository.

public class DeveloperRepository : GenericRepository<Developer>, IDeveloperRepository

{

public DeveloperRepository(ApplicationContext context):base(context)

{

}

public IEnumerable<Developer> GetPopularDevelopers(int count)

{

return \_context.Developers.OrderByDescending(d => d.Followers).Take(count).ToList();

}

}

You can notice that we have not implemented all the 7 functions here, as it is is already implemented in our Generic Repository. Saves a lot of lines, yeah?

Similary, let’s create interface and implementation for ProjectRepository.

public interface IProjectRepository : IGenericRepository<Project>

{

}

Now, the implementation.

public class ProjectRepository : GenericRepository<Project>, IProjectRepository

{

public ProjectRepository(ApplicationContext context): base(context)

{

}

}

You can see that the interface and implementations are quite blank. So why create new class and interface for Project? This can also attribute to a good practice while developing applications. We also anticipate that in future, there can be functions that are spcific to the Project Entity. To support them later on, we provide with interfaces and classes. Future Proofing, yeah?

Finally, let’s register these Interfaces to the respective implementaions in the Startup.cs of the WebApi Project. Navigate to Startup.cs/ConfigureServices Method and add these lines.

#region Repositories

services.AddTransient(typeof(IGenericRepository<>), typeof(GenericRepository<>));

services.AddTransient<IDeveloperRepository, DeveloperRepository>();

services.AddTransient<IProjectRepository, ProjectRepository>();

#endregion

Unit Of Work Pattern

Unit of Work Pattern is a design pattern with which you can expose various respostiories in our application. It has very similar properties of dbContext, just that Unit of Work is not coupled to any framework like dbContext to Entity Framework Core.

Till now, we have built a couple of repositories. We can easily inject these repositories to the constructor of the Services classes and access data. This is quite easy when you have just 2 or 3 repository objects involved. What happens when there are quite more than 3 repositories. It would not be practical to keep adding new injections every now and then. Inorder to wrap all the Repositories to a Single Object, we use Unit Of Work.

Unit of Work is responsible for exposing the available Repositories and to Commit Changes to the DataSource ensuring a full transaction, without loss of data.

The other major advantage is that, multiple repository objects will have different instances of dbcontext within them. This can lead to data leaks in complex cases.

Let’s say that you have a requirement to insert a new Developer and a new Project within the same transaction. What happens when the new Developer get’s inserted but the Project Repository fails for some reason. In real-world scenarios, this is quite fatal. We will need to ensure that both the repositories work well, before commiting any change to the database. This is exactly why we decided to not include SaveChanges in any of the repostories. Clear?

Rather, the SaveChanges will be available in the UnitOfWork Class. You will get a better idea once you see the impelemntation.

Let’s get started with the IUnitOfWork. Create a new Interface in the domain Project, Interfaces/IUnitOfWork

public interface IUnitOfWork : IDisposable

{

IDeveloperRepository Developers { get; }

IProjectRepository Projects { get; }

int Complete();

}

You can see that we are listing the interfaces of the required Repositories within the UOW Interface. It’s also a Disposable Element. Finally we have a ‘Complete’ Function which will save the changes to the database.

Let’s implement this interface. Create the implementation at the DataAccess Project. Add a new class in the UnitOfWork/UnitOfWork.cs

public class UnitOfWork : IUnitOfWork

{

private readonly ApplicationContext \_context;

public UnitOfWork(ApplicationContext context)

{

\_context = context;

Developers = new DeveloperRepository(\_context);

Projects = new ProjectRepository(\_context);

}

public IDeveloperRepository Developers { get; private set; }

public IProjectRepository Projects { get; private set; }

public int Complete()

{

return \_context.SaveChanges();

}

public void Dispose()

{

\_context.Dispose();

}

}

Note that, Here we are injecting a private AppplicationContext. Let’s wire up or controllers with these Repositories. Ideally you would want to have a service layer between the Repository and Controllers. But, to keep things fairly simple, we will avoid the service layer now.

Before that, let’s not forget to register the IUnitofWork Interface in our Application. Navigate to Startup.cs/ConfigureServices Method and add this line.

services.AddTransient<IUnitOfWork, UnitOfWork>();

Wiring up with an API Controller

Add a new Empty API Controller in the WebAPI Project under the Controllers folder.

[Route("api/[controller]")]

[ApiController]

public class DeveloperController : ControllerBase

{

private readonly IUnitOfWork \_unitOfWork;

public DeveloperController(IUnitOfWork unitOfWork)

{

\_unitOfWork = unitOfWork;

}

}

Here are injecting only the IUnitOfWork object. This way, you can completely avoid writing lines and lines of injections to your controllers.

Now, let’s say we need two endpoints for this controller. A POST and a GET Method.

* Get all the Popular Developers.
* Insert a new Develoeper an a new Project.

We’ll start working on the methods now.

public IActionResult GetPopularDevelopers([FromQuery]int count)

{

var popularDevelopers = \_unitOfWork.Developers.GetPopularDevelopers(count);

return Ok(popularDevelopers);

}

Line #3 - Using the \_unitofWork object, we are able to acces the custom method ‘GetPopularDeveloper’ we created. This returns a set of developers sorted by the descending order of the follower count.  
Line #4 Returns a 200 Status Ok with the developers.

[HttpPost]

public IActionResult AddDeveloperAndProject()

{

var developer = new Developer

{

Followers = 35,

Name = "Mukesh Murugan"

};

var project = new Project

{

Name = "codewithmukesh"

};

\_unitOfWork.Developers.Add(developer);

\_unitOfWork.Projects.Add(project);

\_unitOfWork.Complete();

return Ok();

}

Line 4-12 is where I build sample entity objects just for demonstration purpose.  
Line 13 - Add the developer object to the uow context.  
Line 14 - Add the project object to the uow context.  
Line 15 - Finally commits the changes to the database.

What would happen if we didnt have an UnitOfWork Abstraction?

Let’s say Line 13 saves the developer to the database, but for some reason, Line 14 fails to store the project. This can be quite fatal for applications due to data inconsistency. By introducing a unit of work, we are able to store both the developer and project in one go, in a single transaction.

Testing with PostMan

Adding a new Developer and Project. Open up Postman and send a POST request to <https://localhost:xxxx/api/developer>

We get a 200 Ok Status Code, which means that the unit of work trasaction is completed. Now let’s fetch the data and check.

As expected, we are able to retreive the inserted data. That’s it for this extensive guide on Repository Pattern in ASP.NET Core Applications or C# in general.

Liked this article? I have added another article that discusses around the Generic Repository Pattern, it’s limitation and how we can use Specification Pattern to overcome it. Specification Pattern in ASP.NET Core is one of the coolest design patterns to have in your application. Read the entire article here - [Specification Pattern in ASP.NET Core – Enhancing Generic Repository Pattern](https://codewithmukesh.com/blog/specification-pattern-in-aspnet-core/)

Summary

We learnt all about Repository Pattern in ASP.NET Core Application, Generic Repositories, Unit Of Work , a cleaner way to access data with layered projects, and other Use Case Scenarios. This covers nearly everything that you would need to know to become a Pro at Repository Pattern in ASP.NET Core. Have any suggestions or question? Feel free to leave them in the comments section below. Here is the [source code](https://github.com/iammukeshm/RepositoryPattern.WebApi) of the implementation for your reference. Thanks and Happy Coding! :D

**Repository Pattern With ASP.NET MVC And Entity Framework**

Repository Pattern is used to create an abstraction layer between the data access layer and the business logic layer of an application. The repository directly communicates with the data access layer [DAL] and gets the data and provides it to the business logic layer [BAL]. The main advantage of using a repository pattern is to isolate the data access logic and business logic so that if you make changes in any of this logic it cannot directly on other logic.

Today, I am going to explain how to use repository patterns in ASP.NET MVC with EntityFramework.

**Step 1**

* Start Visual Studio 2013 or 2012.
* Create a new project -> Web -> Visual Studio 2012.
* Select ASP.NET MVC 4 Web Application.
* Provide the Name and Location for the project and click Next.
* Choose Basic template as the project template and click OK.

**Step 2.**Create an EmployeeContext.cs and Employee.cs entity class inside the Model folder for the database factory. You can create a database and table manually or if you use my approach it will create a database and table automatically when you run the application the first time because I have used here code first approach.

Employee.cs

using System;

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

using System.ComponentModel.DataAnnotations.Schema;

using System.Linq;

using System.Web;

namespace RepositoryWithMVC.Models

{

[Table("Employee")]

public class Employee

{

[Key]

public int EmployeeId

{

get;

set;

}

[Display(Name = "Employee Name")]

[Required(ErrorMessage = "Name is required")]

public string EmployeeName

{

get;

set;

}

[Display(Name = "Address")]

[Required(ErrorMessage = "Address is required")]

public string Address

{

get;

set;

}

[Required(ErrorMessage = "Email Id is required")]

public string EmailId

{

get;

set;

}

}

}

C#

EmployeeContext.cs

using System;

using System.Collections.Generic;

using System.Data.Entity;

using System.Linq;

using System.Web;

namespace RepositoryWithMVC.Models

{

public class EmployeeContext : DbContext

{

public EmployeeContext() : base("DefaultConnection") {}

public DbSet<Employee> Employees

{

get;

set;

}

}

}

C#

Web.Config

<connectionStrings>

<add name="DefaultConnection"

connectionString="data source=(local); database=Demodb; user id=sa; password=xyz;"

providerName="System.Data.SqlClient" />

</connectionStrings>

XML

**Step 3.**Create a folder with the name “Repository” inside your project. Add an interface and a class respectively IEmployeeRepository.cs and EmployeeRepository.cs.

IEmployeeRepository.cs

using RepositoryWithMVC.Models;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace RepositoryWithMVC.Repository

{

public interface IEmployeeRepository : IDisposable

{

IEnumerable<Employee> GetAllEmployee();

Employee GetEmployeeById(int studentId);

int AddEmployee(Employee employeeEntity);

int UpdateEmployee(Employee employeeEntity);

void DeleteEmployee(int employeeId);

}

}

C#

EmployeeRepository.cs

using RepositoryWithMVC.Models;

using System;

using System.Collections.Generic;

using System.Data;

using System.Linq;

using System.Web;

namespace RepositoryWithMVC.Repository

{

public class EmployeeRepository : IEmployeeRepository

{

private readonly EmployeeContext \_context;

public EmployeeRepository(EmployeeContext context)

{

\_context = context;

}

public IEnumerable<Employee> GetAllEmployee()

{

return \_context.Employees.ToList();

}

public Employee GetEmployeeById(int studentId)

{

return \_context.Employees.Find(studentId);

}

public int AddEmployee(Employee employeeEntity)

{

int result = -1;

if (employeeEntity != null)

{

\_context.Employees.Add(employeeEntity);

\_context.SaveChanges();

result = employeeEntity.EmployeeId;

}

return result;

}

public int UpdateEmployee(Employee employeeEntity)

{

int result = -1;

if (employeeEntity != null)

{

\_context.Entry(employeeEntity).State = EntityState.Modified;

\_context.SaveChanges();

result = employeeEntity.EmployeeId;

}

return result;

}

public void DeleteEmployee(int employeeId)

{

Employee employeeEntity = \_context.Employees.Find(employeeId);

\_context.Employees.Remove(employeeEntity);

\_context.SaveChanges();

}

private bool disposed = false;

protected virtual void Dispose(bool disposing)

{

if (!this.disposed)

{

if (disposing)

{

\_context.Dispose();

}

}

this.disposed = true;

}

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

}

}

C#

**Step 4.** Add an EmployeeController that directly interacts with the Repository.

using RepositoryWithMVC.Models;

using RepositoryWithMVC.Repository;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using System.Web.Mvc;

namespace RepositoryWithMVC.Controllers

{

public class EmployeeController : Controller

{

private IEmployeeRepository \_employeeRepository;

public EmployeeController()

{

\_employeeRepository = new EmployeeRepository(new Models.EmployeeContext());

}

public EmployeeController(IEmployeeRepository employeeRepository)

{

\_employeeRepository = employeeRepository;

}

public ActionResult Index()

{

var model = \_employeeRepository.GetAllEmployee();

return View(model);

}

public ActionResult AddEmployee()

{

if (TempData["Failed"] != null)

{

ViewBag.Failed = "Add Employee Failed";

}

return View();

}

[HttpPost]

public ActionResult AddEmployee(Employee model)

{

if (ModelState.IsValid)

{

int result = \_employeeRepository.AddEmployee(model);

if (result > 0)

{

return RedirectToAction("Index", "Employee");

}

else

{

TempData["Failed"] = "Failed";

return RedirectToAction("AddEmployee", "Employee");

}

}

return View();

}

public ActionResult EditEmployee(int employeeId)

{

if (TempData["Failed"] != null)

{

ViewBag.Failed = "Edit Employee Failed";

}

Employee model = \_employeeRepository.GetEmployeeById(employeeId);

return View(model);

}

[HttpPost]

public ActionResult EditEmployee(Employee model)

{

if (ModelState.IsValid)

{

int result = \_employeeRepository.UpdateEmployee(model);

if (result > 0)

{

return RedirectToAction("Index", "Employee");

}

else

{

return RedirectToAction("Index", "Employee");

}

}

return View();

}

public ActionResult DeleteEmployee(int employeeId)

{

Employee model = \_employeeRepository.GetEmployeeById(employeeId);

return View(model);

}

[HttpPost]

public ActionResult DeleteEmployee(Employee model)

{

if (TempData["Failed"] != null)

{

ViewBag.Failed = "Delete Employee Failed";

}

\_employeeRepository.DeleteEmployee(model.EmployeeId);

return RedirectToAction("Index", "Employee");

}

}

}

C#

**Step 5.** Create a View for the Controller action method like Index, EditEmployee, DeleteEmployee, etc.

Index.cshtml

@model IEnumerable<RepositoryWithMVC.Models.Employee>

@{

ViewBag.Title = "Index";

}

<div align="center">

<h3>Employee Management</h3>

<span>

<a href="@Url.Action("AddEmployee", "Employee")">Add Employee</a>

</span>

<br />

<br />

<table cellpadding="5" border="1">

<tr style="background-color:#808080; color:white;">

<td>Employee Id</td>

<td>Name</td>

<td>Address</td>

<td>Email Id</td>

<td>Action</td>

</tr>

@foreach (var emp in Model)

{

<tr>

<td>@emp.EmployeeId</td>

<td>@emp.EmployeeName</td>

<td>@emp.Address</td>

<td>@emp.EmailId</td>

<td>

<a href="@Url.Action("EditEmployee", "Employee", new { EmployeeId = emp.EmployeeId })">Edit</a>

<a href="@Url.Action("DeleteEmployee", "Employee", new { EmployeeId = emp.EmployeeId })">Delete</a>

</td>

</tr>

}

</table>

</div>

C#

AddEmployee.cshtml

@model RepositoryWithMVC.Models.Employee

@{

ViewBag.Title = "AddEmployee";

}

<div align="center">

<h3>Employee Management</h3>

<br />

<b>Add New Employee</b>

<br />

<br />

@using (Html.BeginForm("AddEmployee", "Employee", FormMethod.Post))

{

<table>

<tr>

<td colspan="2">

@if (ViewBag.Failed != null)

{

<span style="color:red;">@ViewBag.Failed</span>

}

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.EmployeeName)

</td>

<td>

@Html.TextBoxFor(e => e.EmployeeName)

<br />

@Html.ValidationMessageFor(e => e.EmployeeName, null, new { style = "color:red;" })

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.Address)

</td>

<td>

@Html.TextBoxFor(e => e.Address)

<br />

@Html.ValidationMessageFor(e => e.Address, null, new { style = "color:red;" })

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.EmailId)

</td>

<td>

@Html.TextBoxFor(e => e.EmailId)

<br />

@Html.ValidationMessageFor(e => e.EmailId, null, new { style = "color:red;" })

</td>

</tr>

<tr>

<td colspan="2" align="center">

<br />

<input type="submit" value="Submit" />

</td>

</tr>

</table>

}

</div>

C#

EditEmployee.cshtml

@model RepositoryWithMVC.Models.Employee

@{

ViewBag.Title = "Edit Employee";

}

<div align="center">

<h3>Employee Management</h3>

<br />

<b>Edit Employee</b>

<br />

<br />

@using (Html.BeginForm("EditEmployee", "Employee", FormMethod.Post))

{

@Html.HiddenFor(e => e.EmployeeId)

<table>

<tr>

<td colspan="2">

@if (ViewBag.Failed != null)

{

<span style="color:red;">@ViewBag.Failed</span>

}

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.EmployeeName)

</td>

<td>

@Html.TextBoxFor(e => e.EmployeeName)

<br />

@Html.ValidationMessageFor(e => e.EmployeeName, null, new { style = "color:red;" })

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.Address)

</td>

<td>

@Html.TextBoxFor(e => e.Address)

<br />

@Html.ValidationMessageFor(e => e.Address, null, new { style = "color:red;" })

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.EmailId)

</td>

<td>

@Html.TextBoxFor(e => e.EmailId)

<br />

@Html.ValidationMessageFor(e => e.EmailId, null, new { style = "color:red;" })

</td>

</tr>

<tr>

<td colspan="2" align="center">

<br />

<input type="submit" value="Update" />

</td>

</tr>

</table>

}

</div>

C#

DeleteEmployee.cshtml

@model RepositoryWithMVC.Models.Employee

@{

ViewBag.Title = "Delete Employee";

}

<div align="center">

<h3>Employee Management</h3>

<br />

@using (Html.BeginForm("DeleteEmployee", "Employee", FormMethod.Post))

{

@Html.HiddenFor(e => e.EmployeeId)

<table border="1" cellpadding="10">

<tr>

<td colspan="2" align="center">

<b>Delete Employee</b>

@if (ViewBag.Failed != null)

{

<span style="color:red;">@ViewBag.Failed</span>

}

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.EmployeeName)

</td>

<td>

@Html.TextBoxFor(e => e.EmployeeName, new { @readonly = "readonly" })

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.Address)

</td>

<td>

@Html.TextBoxFor(e => e.Address, new { @readonly = "readonly" })

</td>

</tr>

<tr>

<td>

@Html.LabelFor(e => e.EmailId)

</td>

<td>

@Html.TextBoxFor(e => e.EmailId, new { @readonly = "readonly" })

</td>

</tr>

<tr>

<td colspan="2" align="center">

<br />

<input type="submit" value="Delete" />

</td>

</tr>

</table>

}

</div>

C#

**Note.** Change in RouteConfig.cs controller name from home to employee in default.

Thanks for reading this article, hope you enjoyed it.