Repository Design Pattern in ASP.NET Core MVC using EF Core

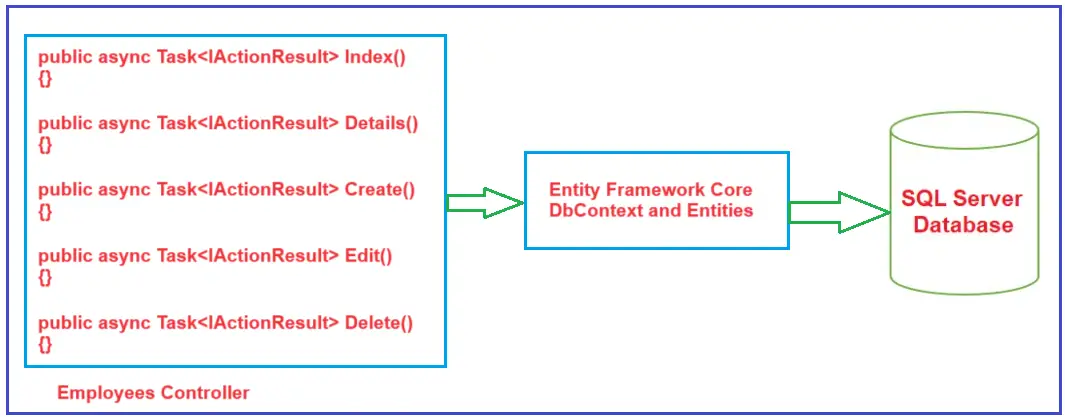
In this article, I will discuss implementing the Repository Design Pattern in ASP.NET Core MVC Application with Entity Framework Core. Please read our previous article discussing how to Perform [**CRUD Operations in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/asp-net-core-mvc-using-entity-framework-core/) Web Applications using Entity Framework Core. We will work with the same application we created in our previous article.

Implementing the Repository Design Pattern in an ASP.NET Core MVC application using Entity Framework Core is a great way to abstract the data access layer code from the rest of your application code, which makes your code more maintainable and testable. Before implementing the repository design pattern, let’s first understand why we need it.

**How Does a Modern Data-Driven Application Access Data from a Database?**

Nowadays, most data-driven applications need to access data residing in one or more other data sources, such as databases. The easiest approach is to write all the data access-related code in the main application.

For example, if you have an ASP.NET Core MVC Controller, let’s say EmployeesController. Then, the EmployeesController class may have many action methods that can perform the typical CRUD (Create, Read, Update, and Delete) operations against the underlying database. Let’s assume we use Entity Framework Core for all these database-related operations. In that case, your application would do something as shown in the below diagram.



As you can see in the above diagram, the action methods of the Employees Controller directly interact with the Entity Framework Core DbContext class and execute the queries to retrieve the data from the database. They also perform the INSERT, UPDATE, and DELETE operations using the DbContext and DbSet properties. The Entity Framework Core, in turn, talks with the underlying SQL Server database. Our previous article used this approach to perform the database CRUD operations using EF Core in ASP.NET Core MVC Application.

**Drawbacks of the Above Implementation:**

As we saw in our previous article, the above implementation works as expected. However, the drawback is that the database access code (i.e., creating the DbContext object, writing the queries, manipulating the data, persisting the changes to the database, etc.) is embedded directly inside the controller action methods. This design or implementation can cause **Code Duplication**, and further, we need to change the controller even if we make a small change in the data access logic.

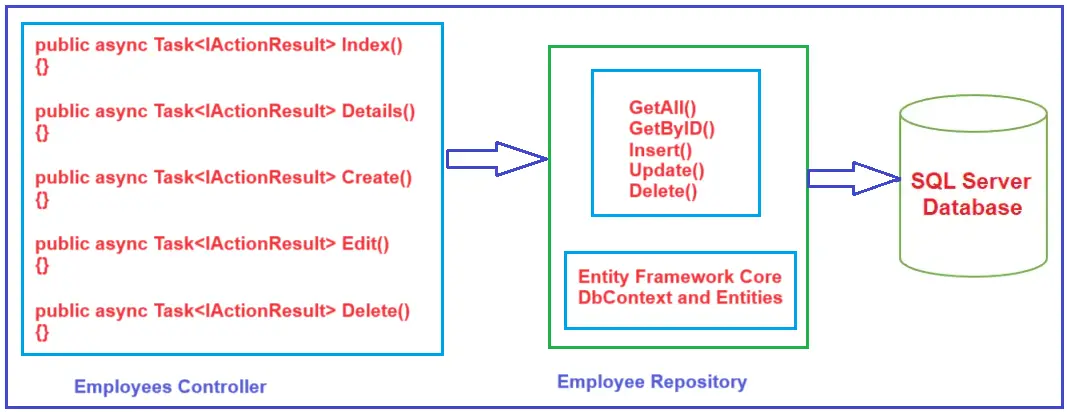
For example, if the application modifies employee information from two controllers, each controller will repeat the same data access code. Future modifications must also be done at two places, i.e., the two controllers where we write the same data access code.

The second problem is the separation of concerns. The controller’s primary responsibility is not to handle the business and data access logic. The repository design pattern can overcome these problems.

**What is the Repository Design Pattern?**

The Repository Design Pattern is a software design pattern that facilitates data management by providing a clean separation between the business logic and the data access layers in an application. This pattern is widely used in applications that require data persistence, meaning storing data in a database. So, it acts as a middleman or middle layer between the rest of the application and the data access logic.

The advantage of doing so is that if you need to make any changes, you must do it in one place. The previous diagram will change to the following diagram with the Repository Design Pattern.



As shown in the above diagram, the Employees Controller won’t directly talk with the Entity Framework Core DbContext class. That means no queries or other data access codes are written in the Employees Controller’s action methods.

The Employee Repository wraps all these operations (i.e., basic CRUD operations). The Employee Repository uses the Entity Framework Core DbContext and DbSet properties to perform the CRUD operations. As you can see, now the Employee repository has methods such as GetAll(), GetByID(), Insert(), Update(), and Delete(). These methods will perform the Typical CRUD operations against the underlying database. The Employees Controller uses the Employee Repository methods to perform the required database operations.

Advertisements

**Key Features of Repository Design Pattern**

* **Abstraction:** The repository design pattern acts as a middle layer that abstracts the details of the data access logic from the application’s business logic.
* **Decoupling:** The repository pattern decouples the data access code from the rest of the application, allowing the business logic to be independent of the underlying database technology.
* **Consistency:** Repositories can help maintain consistency in data manipulation logic since all data access logic is centralized in one place. This avoids code duplication and makes implementing and changing data handling easier.
* **Testing:** With repositories, it becomes easier to unit test the business logic without worrying about the integration with the database because repositories can be mocked or stubbed out during testing.
* **Aggregation of Data Operations:** The repository pattern typically includes standard methods for data operations such as add, remove, update, and find. This standardization supports more robust and consistent development.

**What is a Repository?**

A repository is a class defined for an entity with all the possible database operations. For example, a repository for an employee will have the basic CRUD operations and any other possible operations related to the Employee entity. Similarly, a repository for a department will include the CRUD operations related to the Department entity.

**How to Implement Repository Design Pattern in ASP.NET Core MVC using EF Core?**

We are going to work with the same application that we created in our previous article, where we created the following Employee and Department Entities:

**Employee.cs**

**using** *System.ComponentModel.DataAnnotations;*

**namespace** *CRUDinCoreMVC.Models*

**{**

**public** **class** Employee

**{**

**public** **int** EmployeeId **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **string** Email **{** **get**; **set**; **}**

**public** **string** Position **{** **get**; **set**; **}**

**[**Display**(**Name ="Department Name"**)]**

**public** **int** DepartmentId **{** **get**; **set**; **}**

**public** Department? Department **{** **get**; **set**; **}**

**}**

**}**

**Department.cs**

**namespace** *CRUDinCoreMVC.Models*

**{**

**public** **class** Department

**{**

**public** **int** DepartmentId **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** List**<**Employee**>** Employees **{** **get**; **set**; **}**

**}**

**}**

We have also created the following **DbContext** class:

**using** *Microsoft.EntityFrameworkCore;*

**namespace** *CRUDinCoreMVC.Models*

**{**

**public** **class** EFCoreDbContext : DbContext

**{**

//Constructor calling the Base DbContext Class Constructor

**public** EFCoreDbContext**(**DbContextOptions**<**EFCoreDbContext**>** options**)** : **base(**options**)**

**{**

**}**

//OnConfiguring() method is used to select and configure the data source

**protected** **override** **void** OnConfiguring**(**DbContextOptionsBuilder optionsBuilder**)**

**{**

//We will store the connection string in AppSettings.json file instead of hard coding here

**}**

**protected** **override** **void** OnModelCreating**(**ModelBuilder modelBuilder**)**

**{**

**}**

//Adding Domain Classes as DbSet Properties

**public** DbSet**<**Employee**>** Employees **{** **get**; **set**; **}**

**public** DbSet**<**Department**>** Departments **{** **get**; **set**; **}**

**}**

**}**

We have then added the database connection string in the **appsettings.json** file as follows:

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

**}**

**}**,

"AllowedHosts": "\*",

"ConnectionStrings": **{**

"EFCoreDBConnection": "Server=LAPTOP-6P5NK25R\\SQLSERVER2022DEV;Database=EFCoreMVCDB;Trusted\_Connection=True;TrustServerCertificate=True;"

**}**

**}**

Then we registered the connection string and DbContext class to use the connection string in the program class, and we have set the Employees Controller Index action method as the default route as follows:

**using** *CRUDinCoreMVC.Models;*

**using** *Microsoft.EntityFrameworkCore;*

**namespace** *CRUDinCoreMVC*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

//Configure the ConnectionString and DbContext class

builder.Services.AddDbContext**<**EFCoreDbContext**>(**options =**>**

**{**

options.UseSqlServer**(**builder.Configuration.GetConnectionString**(**"EFCoreDBConnection"**))**;

**})**;

// Add services to the container.

builder.Services.AddControllersWithViews**()**;

var app = builder.Build**()**;

// Configure the HTTP request pipeline.

**if** **(**!app.Environment.IsDevelopment**())**

**{**

app.UseExceptionHandler**(**"/Home/Error"**)**;

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts**()**;

**}**

app.UseHttpsRedirection**()**;

app.UseStaticFiles**()**;

app.UseRouting**()**;

app.UseAuthorization**()**;

app.MapControllerRoute**(**

name: "default",

pattern: "{controller=Employees}/{action=Index}/{id?}"**)**;

app.Run**()**;

**}**

**}**

**}**

Then we created the Employees and Departments controller using MVC Controller with views, using Entity Framework template. Then, we modified some of the actions and views of the Employees Controller.

**Create Repository Interfaces**

A repository interface typically includes methods covering basic CRUD operations (Create, Read, Update, Delete). That means these interfaces declare the operations you can perform on the entities. A repository typically does at least five operations as follows:

* Selecting all records from a table
* Selecting a single record based on its primary key
* Insert a new record into the database
* Update an existing record in the database
* Delete an existing record in the database

However, this list is not fixed. You may have more or fewer operations in the repository as per your business requirements. For our example, these five operations are needed from the Employee repository. To implement this, first, we will create an Interface (e.g., IEmployeeRepository) with these five methods, and then we will implement this interface in a class (e.g., **EmployeeRepositpry**).

First, add a folder with the name **Repository** to your project. To do so, right-click on the **Project => Add => New Folder** and then rename the folder name as **Repository**. Add an Interface within this Repository folder named **IEmployeeRepository.cs**, and copy and paste the code below. The following code is self-explained; please go through the comment lines.

**using** *CRUDinCoreMVC.Models;*

**namespace** *CRUDinCoreMVC.Repository*

**{**

**public** **interface** IEmployeeRepository

**{**

//This method returns all the Employee entities as an enumerable collection

Task**<**IEnumerable**<**Employee**>>** GetAllAsync**()**;

//This method accepts an integer parameter representing an Employee ID and

//returns a single Employee entity matching that Employee ID.

Task**<**Employee?**>** GetByIdAsync**(int** EmployeeID**)**;

//This method accepts an Employee object as the parameter and

//adds that Employee object to the Employees DbSet.

//mark the entity state as Added

Task InsertAsync**(**Employee employee**)**;

//This method accepts an Employee object as a parameter and

//marks that Employee object as a modified Employee in the DbSet.

Task UpdateAsync**(**Employee employee**)**;

//This method accepts an EmployeeID as a parameter and

//removes that Employee entity from the Employees DbSet.

//Mark the Entity state as Deleted

Task DeleteAsync**(int** employeeId**)**;

//This method Saves changes to the EFCoreDb database.

Task SaveAsync**()**;

**}**

**}**

**Adding EmployeeRepository Class**

Now, we need to add the **EmployeeRepository** class by implementing the **IEmployeeRepository** interface and providing implementations for the interface methods. To do so, add a class file within the **Repository** folder named **EmployeeRepository.cs** and copy and paste the code below. The following example code is self-explained.

**using** *CRUDinCoreMVC.Models;*

**using** *Microsoft.EntityFrameworkCore;*

**namespace** *CRUDinCoreMVC.Repository*

**{**

**public** **class** EmployeeRepository : IEmployeeRepository

**{**

//The following variable is going to hold the EmployeeDBContext instance

**private** **readonly** EFCoreDbContext \_context;

//Initializing the EmployeeDBContext instance which it received as an argument

//MVC Framework DI Container will provide the EFCoreDbContext instance

**public** EmployeeRepository**(**EFCoreDbContext context**)**

**{**

\_context = context;

**}**

//Returns all employees from the database.

**public** **async** Task**<**IEnumerable**<**Employee**>>** GetAllAsync**()**

**{**

**return** **await** \_context.Employees.Include**(**e =**>** e.Department**)**.ToListAsync**()**;

**}**

//Retrieves a single employee by their ID.

**public** **async** Task**<**Employee?**>** GetByIdAsync**(int** EmployeeID**)**

**{**

var employee = **await** \_context.Employees

.Include**(**e =**>** e.Department**)**

.FirstOrDefaultAsync**(**m =**>** m.EmployeeId == EmployeeID**)**;

**return** employee;

**}**

//Adds a new employee to the database.

**public** **async** Task InsertAsync**(**Employee employee**)**

**{**

**await** \_context.Employees.AddAsync**(**employee**)**;

**}**

//Updates an existing employee's details.

**public** **async** Task UpdateAsync**(**Employee employee**)**

**{**

\_context.Employees.Update**(**employee**)**;

**}**

//Deletes an employee from the database

**public** **async** Task DeleteAsync**(int** employeeId**)**

**{**

var employee = **await** \_context.Employees.FindAsync**(**employeeId**)**;

**if** **(**employee != **null)**

**{**

\_context.Employees.Remove**(**employee**)**;

**}**

**}**

//InsertAsync, UpdateAsync, and DeleteAsync methods,

//remember to call SaveAsync to commit the changes to the database.

**public** **async** Task SaveAsync**()**

**{**

**await** \_context.SaveChangesAsync**()**;

**}**

**}**

**}**

**Using Employee Repository inside Employees Controller:**

Next, modify the **EmployeesController** to Use **Employee Repository** instead of DbContext and DbSet Properties to perform the CRUD operations on the Employee entity as follows:

**using** *System;*

**using** *System.Collections.Generic;*

**using** *System.Linq;*

**using** *System.Threading.Tasks;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.AspNetCore.Mvc.Rendering;*

**using** *Microsoft.EntityFrameworkCore;*

**using** *CRUDinCoreMVC.Models;*

**using** *CRUDinCoreMVC.Repository;*

**namespace** *CRUDinCoreMVC.Controllers*

**{**

**public** **class** EmployeesController : Controller

**{**

//Other Than Employee Entity

**private** **readonly** EFCoreDbContext \_context;

//For Employee Entity

**private** **readonly** IEmployeeRepository \_employeeRepository;

**public** EmployeesController**(**IEmployeeRepository employeeRepository, EFCoreDbContext context**)**

**{**

\_employeeRepository = employeeRepository;

\_context = context;

**}**

// GET: Employees

**public** **async** Task**<**IActionResult**>** Index**()**

**{**

var employees = **await** \_employeeRepository.GetAllAsync**()**;

**return** View**(**employees**)**;

**}**

// GET: Employees/Details/5

**public** **async** Task**<**IActionResult**>** Details**(int**? id**)**

**{**

**if** **(**id == **null)**

**{**

**return** NotFound**()**;

**}**

var employee = **await** \_employeeRepository.GetByIdAsync**(**Convert.ToInt32**(**id**))**;

**if** **(**employee == **null)**

**{**

**return** NotFound**()**;

**}**

**return** View**(**employee**)**;

**}**

// GET: Employees/Create

**public** IActionResult Create**()**

**{**

ViewData**[**"DepartmentId"**]** = new SelectList**(**\_context.Departments, "DepartmentId", "Name"**)**;

**return** View**()**;

**}**

// POST: Employees/Create

// To protect from overposting attacks, enable the specific properties you want to bind to.

// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** **async** Task**<**IActionResult**>** Create**([**Bind**(**"EmployeeId,Name,Email,Position,DepartmentId"**)]** Employee employee**)**

**{**

**if** **(**ModelState.IsValid**)**

**{**

**await** \_employeeRepository.InsertAsync**(**employee**)**;

//Call SaveAsync to Insert the data into the database

**await** \_employeeRepository.SaveAsync**()**;

**return** RedirectToAction**(nameof(**Index**))**;

**}**

ViewData**[**"DepartmentId"**]** = new SelectList**(**\_context.Departments, "DepartmentId", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

// GET: Employees/Edit/5

**public** **async** Task**<**IActionResult**>** Edit**(int**? id**)**

**{**

**if** **(**id == **null** **)**

**{**

**return** NotFound**()**;

**}**

var employee = **await** \_employeeRepository.GetByIdAsync**(**Convert.ToInt32**(**id**))**;

**if** **(**employee == **null)**

**{**

**return** NotFound**()**;

**}**

ViewData**[**"DepartmentId"**]** = new SelectList**(**\_context.Departments, "DepartmentId", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

// POST: Employees/Edit/5

// To protect from overposting attacks, enable the specific properties you want to bind to.

// For more details, see http://go.microsoft.com/fwlink/?LinkId=317598.

**[**HttpPost**]**

**[**ValidateAntiForgeryToken**]**

**public** **async** Task**<**IActionResult**>** Edit**(int** id, **[**Bind**(**"EmployeeId,Name,Email,Position,DepartmentId"**)]** Employee employee**)**

**{**

**if** **(**id != employee.EmployeeId**)**

**{**

**return** NotFound**()**;

**}**

**if** **(**ModelState.IsValid**)**

**{**

**try**

**{**

**await** \_employeeRepository.UpdateAsync**(**employee**)**;

**await** \_employeeRepository.SaveAsync**()**;

**}**

**catch** **(**DbUpdateConcurrencyException**)**

**{**

var emp = **await** \_employeeRepository.GetByIdAsync**(**employee.EmployeeId**)**;

**if** **(**emp == **null)**

**{**

**return** NotFound**()**;

**}**

**else**

**{**

**throw**;

**}**

**}**

**return** RedirectToAction**(nameof(**Index**))**;

**}**

ViewData**[**"DepartmentId"**]** = new SelectList**(**\_context.Departments, "DepartmentId", "Name", employee.DepartmentId**)**;

**return** View**(**employee**)**;

**}**

// GET: Employees/Delete/5

**public** **async** Task**<**IActionResult**>** Delete**(int**? id**)**

**{**

**if** **(**id == **null)**

**{**

**return** NotFound**()**;

**}**

var employee = **await** \_employeeRepository.GetByIdAsync**(**Convert.ToInt32**(**id**))**;

**if** **(**employee == **null)**

**{**

**return** NotFound**()**;

**}**

**return** View**(**employee**)**;

**}**

// POST: Employees/Delete/5

**[**HttpPost, ActionName**(**"Delete"**)]**

**[**ValidateAntiForgeryToken**]**

**public** **async** Task**<**IActionResult**>** DeleteConfirmed**(int** id**)**

**{**

var employee = **await** \_employeeRepository.GetByIdAsync**(**id**)**;

**if** **(**employee != **null)**

**{**

**await** \_employeeRepository.DeleteAsync**(**id**)**;

**await** \_employeeRepository.SaveAsync**()**;

**}**

**return** RedirectToAction**(nameof(**Index**))**;

**}**

**}**

**}**

**Registering the EmployeeRepository:**

Next, we need to register the **IEmployeeRepositiry** and **EmployeeRepositiry** to the built-in dependency injection container. So, add the following line to your Program class:

**builder.Services.AddScoped<IEmployeeRepository, EmployeeRepository>();**

Now, run the application and perform the database CRUD Operations using Employee Controller action methods and Views, and it should work as expected.

**Note:** We have implemented the Repository Pattern for the Employee Entity, and you need to implement the Repository for the Department Entity similarly.

**When Should We Use Repository Design Pattern in ASP.NET Core MVC?**

The Repository Design Pattern is commonly used in ASP.NET Core MVC to abstract the data layer, making your application easier to manage and test. Here are several scenarios where implementing the Repository Design Pattern can be beneficial:

* **Decoupling Application from Persistence Frameworks:** If you anticipate that your application may need to switch databases or ORM (Object-Relational Mapping) frameworks, using a repository can isolate the rest of your application from these changes.
* **Unit Testing:** The Repository Pattern allows you to mock data access logic, making it easier to write unit tests for your business logic without setting up a database.
* **Complex Data Operations:** When your application performs complex queries, a repository can encapsulate these complexities. This keeps your controllers slim and focused on handling request and response logic.
* **Code Reusability:** Using repositories can avoid duplicating data access logic across your application. Repositories can be reused across different parts of your application, promoting DRY (Don’t Repeat Yourself) principles.
* **Cleaner Code Base:** Implementing repositories helps maintain a cleaner codebase by segregating business logic from data access logic. This separation of concerns makes the system easier to navigate and maintain.

**When Shoud Not We Use Repository Design Pattern?**

* **Simple CRUD Operations:** If your application primarily performs simple create, read, update, and delete (CRUD) operations and doesn’t involve complex business logic or data access requirements, implementing a repository might be overkill. Direct data access techniques might suffice.
* **Small Applications:** Adding a repository layer can introduce unnecessary complexity and overhead for smaller, less complex applications. In these cases, the application might benefit from a simpler and more direct approach to data access.
* **Applications with Minimal Data Interaction:** If the application interacts minimally with data stores or uses data straightforwardly without complex transactions or queries, the repository pattern might add an unnecessary abstraction layer.

In the next article, I will discuss How to Implement the [**Generic Repository Pattern in ASP.NET Core MVC**](https://dotnettutorials.net/lesson/generic-repository-pattern-in-asp-net-core-mvc/) with Entity Framework Core. In this article, I explain the **Repository Design Pattern in ASP.NET Core MVC** with EF Core. I hope you enjoy this Repository Design Pattern in ASP.NET Core MVC with Entity Framework Core article.