Repository Pattern Implementation Guidelines in c#

Back to: Dot Net Design Patterns With Real-Time Examples

Repository Pattern Implementation Guidelines in c#

In this article, I will discuss how to use both generic and non-generic repository in ASP.NET MVC application using Entity Framework. That means we discuss the Repository Pattern Implementation Guidelines in C#.

Please read the below article before proceeding to this article.

In Part – 1, I discussed the basics of Repository Design pattern

In Part – 2, I discussed Non-Generic (Basic) Repository Pattern in ASP.NET MVC using Entity Framework

In Part – 3, I discussed generic Repository pattern in ASP.NET MVC application using Entity Framework.

In Part – 4, I discussed the Implementation Guidelines of using Repository Design Pattern

As we already discussed the repository pattern is used to create an abstraction layer between the data access layer and business logic layer to perform the CRUD operations against the underlying database.

The repository can be implemented in two ways.

1. Non-Generic Repository (Discussed in Part – 2,)
2. Generic Repository (Discussed in Part – 3)

Generic Repository Pattern

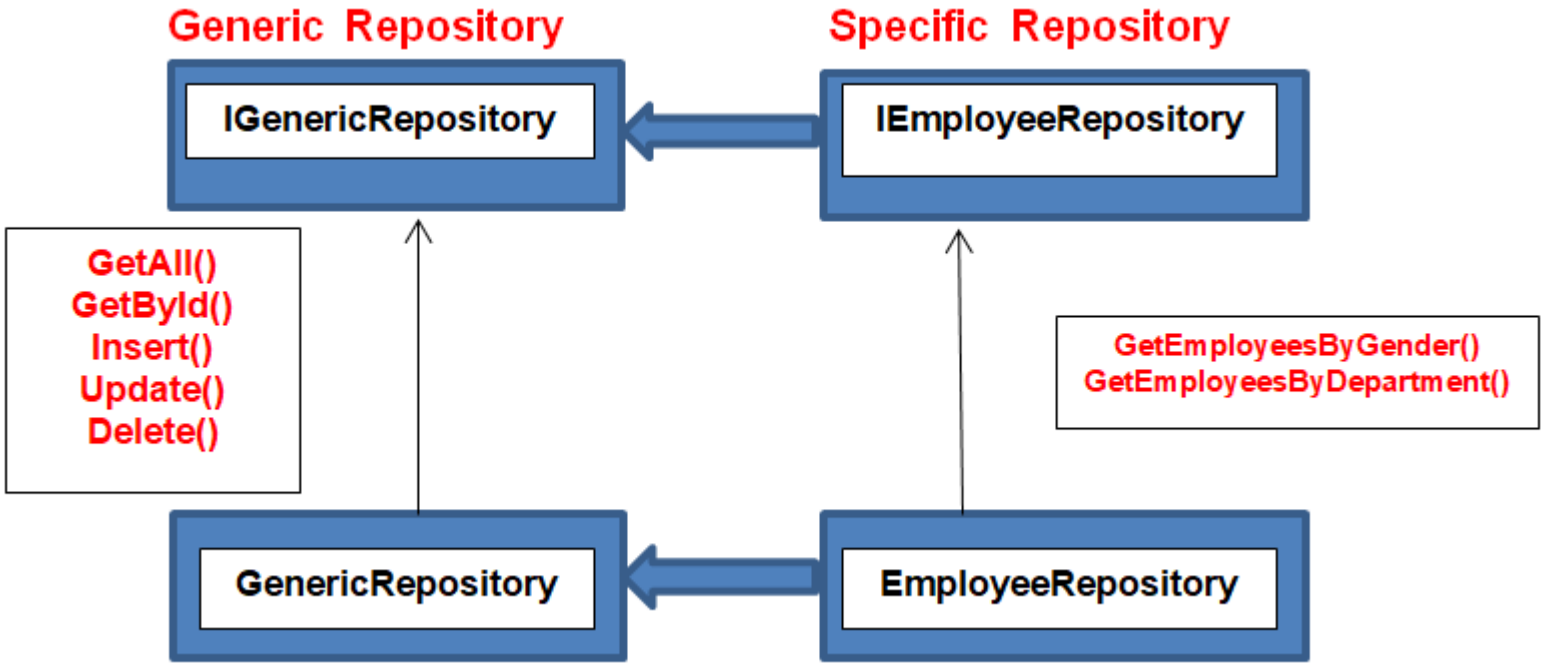
A generic repository is used to define common database operations (like Create, Retrieve Update, Delete etc.) for all the database entities in a single class.

Non-Generic Repository Pattern (Specific Repository)

A non-generic repository is used to define all database operation related to an entity within a separate class. For example, if you have two entities let's say, Employee and Customer, then each entity will have its own implementation repository.

Repository Pattern Implementation Guidelines

If you will use one of the above implementations, then with the generic repository, you cannot use specific operation for an entity and in case of non-generic implementation, you have to write code for common CRUD operations for each entity. So better way is, just create a generic repository for commonly used CRUD operation and for specific operation create a non-generic repository and inherit from the generic repository. The below diagram explains the above things.



Let's understand this with one example.

We are going to work with the same example that we started in Part – 2, and Continue in Part – 3. So please read Part – 2, and Part – 3 before proceeding to this article.

Modify the IGenericRepository.cs file as shown below.

```
namespace RepositoryUsingEFinMVC.GenericRepository
{
    public interface IGenericRepository<T> where T : class
    {
        IEnumerable<T> GetAll();
        T GetById(object id);
        void Insert(T obj);
        void Update(T obj);
        void Delete(object id);
```

```
        void Save();
    }
}
```

Modify the GenericRepository.cs file as shown below

```
namespace RepositoryUsingEFinMVC.GenericRepository
{
    public class GenericRepository<T> : IGenericRepository<T> where T : class
    {
        public EmployeeDBContext _context = null;
        public DbSet<T> table = null;

        public GenericRepository()
        {
            this._context = new EmployeeDBContext();
            table = _context.Set<T>();
        }

        public GenericRepository(EmployeeDBContext _context)
        {
            this._context = _context;
            table = _context.Set<T>();
        }

        public IEnumerable<T> GetAll()
        {
            return table.ToList();
        }

        public T GetById(object id)
        {
            return table.Find(id);
        }
```

```
        public void Insert(T obj)
        {
            table.Add(obj);
        }

        public void Update(T obj)
        {
            table.Attach(obj);
            _context.Entry(obj).State = EntityState.Modified;
        }

        public void Delete(object id)
        {
            T existing = table.Find(id);
            table.Remove(existing);
        }

        public void Save()
        {
            _context.SaveChanges();
        }
    }
}
```

Note: The above code is the implementation of Generic Repository where we implement the code for common CRUD operation for each entity.

Now we need to provide a specific implementation for each entity. Let's we need two extra operations for Employee entity such as get Employees By gender and get employees by department. As these two operations are specific to Employee entity there is no point to add these two operations in the Generic Repository.

So we need to create a non-generic repository called EmployeeRepository which will also inherit from GenericRepository and within this repository, we need to provide the two specific operations as shown below.

Modify the IEmployeeRepository.cs file as shown below.

```
using RepositoryUsingEFinMVC.DAL;
using RepositoryUsingEFinMVC.GenericRepository;
using System.Collections.Generic;
namespace RepositoryUsingEFinMVC.Repository
{
    public interface IEmployeeRepository : IGenericRepository<Employee>
    {
        IEnumerable<Employee> GetEmployeesByGender(string Gender);
        IEnumerable<Employee> GetEmployeesByDepartment(string Dept);
    }
}
```

Modify the EmployeeRepository.cs file as shown below

```
namespace RepositoryUsingEFinMVC.Repository
{
    public class EmployeeRepository : GenericRepository<Employee>, IEmployeeRepository
    {
        public IEnumerable<Employee> GetEmployeesByGender(string Gender)
        {
            return _context.Employees.Where(emp => emp.Gender == Gender).ToList();
        }

        public IEnumerable<Employee> GetEmployeesByDepartment(string Dept)
        {
            return _context.Employees.Where(emp => emp.Dept == Dept).ToList();
        }
    }
}
```

Now we need to use both these generic and non-generic repository in Employee Controller.

Modify the Employee Controller as shown below.

```
using RepositoryUsingEFinMVC.Repository;
using System.Web.Mvc;
using RepositoryUsingEFinMVC.DAL;
using RepositoryUsingEFinMVC.GenericRepository;

namespace RepositoryUsingEFinMVC.Controllers
{
    public class EmployeeController : Controller
    {
        private IGenericRepository<Employee> repository = null;
        private IEmployeeRepository employee_repository = null;

        public EmployeeController()
        {
            this.employee_repository = new EmployeeRepository();
            this.repository = new GenericRepository<Employee>();
        }

        public EmployeeController(EmployeeRepository repository)
        {
            this.employee_repository = repository;
        }
        public EmployeeController(IGenericRepository<Employee> repository)
        {
            this.repository = repository;
        }


        [HttpGet]
        public ActionResult Index()
        {
            //you can not access the below two mwthods using generic repository
```

```
        //var model = repository.GetEmployeesByDepartment("IT");
        var model = employee_repository.GetEmployeesByGender("Male");
        return View(model);
    }

    [HttpGet]
    public ActionResult AddEmployee()
    {
        return View();
    }

    [HttpPost]
    public ActionResult AddEmployee(Employee model)
    {
        if (ModelState.IsValid)
        {
            repository.Insert(model);
            repository.Save();
            return RedirectToAction("Index", "Employee");
        }
        return View();
    }

    [HttpGet]
    public ActionResult EditEmployee(int EmployeeId)
    {
        Employee model = repository.GetById(EmployeeId);
        return View(model);
    }

    [HttpPost]
    public ActionResult EditEmployee(Employee model)
    {
        if (ModelState.IsValid)
        {
```

```
                repository.Update(model);
                repository.Save();
                return RedirectToAction("Index", "Employee");
            }
            else
            {
                return View(model);
            }
        }

        [HttpGet]
        public ActionResult DeleteEmployee(int EmployeeId)
        {
            Employee model = repository.GetById(EmployeeId);
            return View(model);
        }

        [HttpPost]
        public ActionResult Delete(int EmployeeID)
        {
            repository.Delete(EmployeeID);
            repository.Save();
            return RedirectToAction("Index", "Employee");
        }
    }
}
```

Note: Using specific repository we can access all the operations but using generic repository we can access only the operations which are defined in the generic repository.

Now run the application and see everything is working as expected.

In the next article, I will discuss how to use Unit of Work using both generic and non-generic repository in ASP.NET MVC application.

SUMMARY

In this article, I try to explain **Repository Pattern Implementation Guidelines** in ASP.NET MVC application using Entity Framework step by step with a simple example. I hope this article will help you with your need. I would like to have your feedback. Please post your feedback, question, or comments about this article.

← Previous Lesson
Generic Repository Pattern in C#

Next Lesson →
Unit of Work using Repository Design Pattern

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment

Name*

Email*

Website

Post Comment

SINGLETON DESIGN PATTERN

- Singleton Design Pattern in C#
- Why Singleton Class sealed in C#
- Thread-safe Singleton Design Pattern in C#
- Lazy Loading and Eager loading in Singleton Design Pattern
- Singleton VS Static class in C#
- Singleton Design Pattern Real Time Example in C#

DEPENDENCY INJECTION DESIGN PATTERN

- Dependency Injection in C#
- Property and Method Dependency Injection in C#
- Dependency Injection using Unity Container in MVC

REPOSITORY DESIGN PATTERN

- Repository Design Pattern in C#
- How to implement Repository Design Pattern in C#
- Generic Repository Pattern in C#
- **Repository Pattern Implementation Guidelines in c#**
- Unit of Work using Repository Design Pattern

Factory Design Pattern

- Factory Design Pattern in C#
- Factory Method Design Pattern in C#
- Abstract Factory Design Pattern in C#

INVERSION OF CONTROL

- Introduction to Inversion of Control
- Inversion of Control Using Factory Pattern in C#
- Inversion of Control Using Dependency Inversion Principle
- Inversion of Control Using Dependency Injection Design pattern
- Inversion of Control Containers in C#