**Repository and Unit of Work Patterns in .NET Core**

**What are the Repository and Unit of Work Patterns?**

According to the official MS Docs ([Designing the infrastructure persistence layer | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design)), repositories are classes or components that encapsulate the logic required to access data sources. They include methods for common operations, providing better decoupling and maintainability.

The Unit of Work pattern is used to aggregate multiple operations into a single transaction. With this we ensure that either all operations succeed or fail as a single unit.

Note that you can use the Repository pattern without using the Unit of Work pattern.

**Implementing the Repository and Unit of Work Patterns**

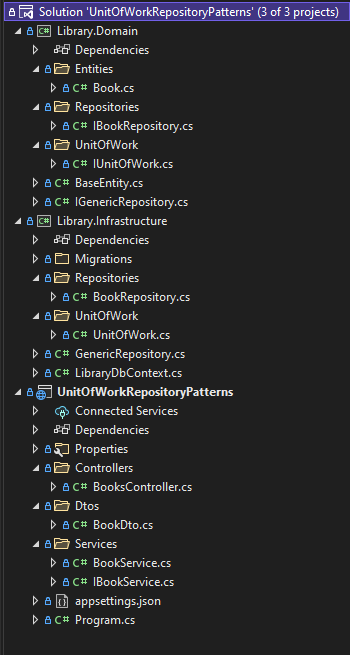
The project uses Entity Framework Core as our O/RM. I suggest you read the official MS Docs [Overview of Entity Framework Core - EF Core | Microsoft Docs](https://docs.microsoft.com/en-us/ef/core/) for more information.

***One important thing to highlight here:***

According to the official MS Docs ([DbContext Class (System.Data.Entity) | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/api/system.data.entity.dbcontext?view=entity-framework-6.2.0)), the **DbContext**class is a combination of the Unit of Work and Repository patterns, where the **DbContext**is an abstraction of the Unit of Work pattern and a **DbSet**is an abstraction of the Repository pattern.

So, you don’t need to use any of these patterns in your code, if you wish. But it is a good approach to create an additional layer of abstraction over them if you don’t want your project to be tightly coupled to Entity Framework Core.

**Project Structure**



**Creating the Domain Entity**

For the simplicity of this example, we will use only one entity which will be a representation of a Book.

 public class Book : BaseEntity

    {

        public string Title { get; set; }

        public int NmPages { get; set; }

        public string Genre { get; set; }

    }

Notice the Base class named **BaseEntity**, from which our entity will inherit. If you decide to add more entities to your example, the same will apply for them.

 public abstract class BaseEntity

    {

        public Guid Id { get; set; }

    }

Last thing to do is see the simple implementation of the **LibraryDbContext.cs**class, where we define the DbSet for the Book entity:

 public class LibraryDbContext : DbContext

    {

        public LibraryDbContext(DbContextOptions<LibraryDbContext> options) : base(options) { }

        public DbSet<Book> Books { get; set; }

    }

**Creating the Generic repository**

Next, we’ll create a generic repository which will have common operations for each entity like Add, Update, Remove and so on. This repository will be implemented by any future entity specific repository we create.

The interface **IGenericRepository.cs** is defined as follows:

public interface IGenericRepository<T> where T : class

    {

        T Get(Expression<Func<T, bool>> expression);

        IEnumerable<T> GetAll();

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

        void Add(T entity);

        void AddRange(IEnumerable<T> entities);

        void Remove(T entity);

        void RemoveRange(IEnumerable<T> entities);

        void Update(T entity);

        void UpdateRange(IEnumerable<T> entities);

        Task<T> GetAsync(Expression<Func<T, bool>> expression, CancellationToken cancellationToken = default);

        Task<IEnumerable<T>> GetAllAsync(CancellationToken cancellationToken = default);

        Task<IEnumerable<T>> GetAllAsync(Expression<Func<T, bool>> expression, CancellationToken cancellationToken = default);

        Task AddAsync(T entity, CancellationToken cancellationToken = default);

        Task AddRangeAsync(IEnumerable<T> entities, CancellationToken cancellationToken = default);

    }

And following is the implementation of the interface in **GenericRepository.cs**file:

using Library.Domain;

using Microsoft.EntityFrameworkCore;

using System.Linq.Expressions;

namespace Library.Infrastructure

{

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

    {

        protected readonly LibraryDbContext \_dbContext;

        private readonly DbSet<T> \_entitiySet;

        public GenericRepository(LibraryDbContext dbContext)

        {

            \_dbContext = dbContext;

            \_entitiySet = \_dbContext.Set<T>();

        }

        public void Add(T entity)

            => \_dbContext.Add(entity);

        public async Task AddAsync(T entity, CancellationToken cancellationToken = default)

            => await \_dbContext.AddAsync(entity, cancellationToken);

        public void AddRange(IEnumerable<T> entities)

            => \_dbContext.AddRange(entities);

        public async Task AddRangeAsync(IEnumerable<T> entities, CancellationToken cancellationToken = default)

            => await \_dbContext.AddRangeAsync(entities, cancellationToken);

        public T Get(Expression<Func<T, bool>> expression)

            => \_entitiySet.FirstOrDefault(expression);

        public IEnumerable<T> GetAll()

            => \_entitiySet.AsEnumerable();

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

            => \_entitiySet.Where(expression).AsEnumerable();

        public async Task<IEnumerable<T>> GetAllAsync(CancellationToken cancellationToken = default)

            => await \_entitiySet.ToListAsync(cancellationToken);

        public async Task<IEnumerable<T>> GetAllAsync(Expression<Func<T, bool>> expression, CancellationToken cancellationToken = default)

            => await \_entitiySet.Where(expression).ToListAsync(cancellationToken);

        public async Task<T> GetAsync(Expression<Func<T, bool>> expression, CancellationToken cancellationToken = default)

            => await \_entitiySet.FirstOrDefaultAsync(expression, cancellationToken);

        public void Remove(T entity)

            => \_dbContext.Remove(entity);

        public void RemoveRange(IEnumerable<T> entities)

            => \_dbContext.RemoveRange(entities);

        public void Update(T entity)

            => \_dbContext.Update(entity);

        public void UpdateRange(IEnumerable<T> entities)

            => \_dbContext.UpdateRange(entities);

    }

}

**Creating the Book Repository**

Now, we want to create a repository for the Book entity we defined earlier. We create an interface named **IBookRepository.cs**with the following code:

 public interface IBookRepository : IGenericRepository<Book>

    {

    }

And now implement this interface in the **BookRepository.cs**file:

 public class BookRepository : GenericRepository<Book>, IBookRepository

    {

        public BookRepository(LibraryDbContext dbContext) : base(dbContext)

        {

        }

    }

***Perfect***. Until now, we’ve finished implementing the repository pattern in our application. Next, we can move on and see how to use this repository as part of the Unit of Work pattern.

**Creating the Unit of Work**

First, we’ll define an interface **IUnitOfWork.cs** with the following code:

public interface IUnitOfWork

    {

        IBookRepository BookRepository { get; }

        void Commit();

        void Rollback();

        Task CommitAsync();

        Task RollbackAsync();

    }

And we’ll implement it in the **UnitOfWork.cs**file:

public class UnitOfWork : IUnitOfWork

    {

        private readonly LibraryDbContext \_dbContext;

        private IBookRepository \_bookRepository;

        public UnitOfWork(LibraryDbContext dbContext)

        {

            \_dbContext = dbContext;

        }

        public IBookRepository BookRepository

        {

            get { return \_bookRepository = \_bookRepository ?? new BookRepository(\_dbContext); }

        }

        public void Commit()

            => \_dbContext.SaveChanges();

        public async Task CommitAsync()

            => await \_dbContext.SaveChangesAsync();

        public void Rollback()

            => \_dbContext.Dispose();

        public async Task RollbackAsync()

            => await \_dbContext.DisposeAsync();

    }

You can see that we have our Book repository as a **field**in the Unit of Work and also methods for the operations commit and rollback.

For each new repository we create, we will need to add it to the Unit of Work.

Let’s not forget to register our service in **Program.cs** file:

builder.Services.AddScoped<IUnitOfWork, UnitOfWork>();

Now the implementation of our Unit of Work pattern is complete.

**Creating the Book Service**

Lastly, we create an **IBookService.cs**interface that has methods for adding a book and getting a list of all the books.

public interface IBookService

    {

        public Task<IEnumerable<Book>> GetAll();

        public Task AddBook(BookDto book);

    }

Let’s add its implementation in the **BookService.cs**file:

public class BookService : IBookService

    {

        public IUnitOfWork \_unitOfWork;

        public BookService(IUnitOfWork unitOfWork)

        {

            \_unitOfWork = unitOfWork;

        }

        public async Task AddBook(BookDto bookDto)

        {

            var book = new Book

            {

                Genre = bookDto.Genre,

                NmPages = bookDto.NmPages,

                Title = bookDto.Title,

            };

            \_unitOfWork.BookRepository.Add(book);

            await \_unitOfWork.CommitAsync();

        }

        public async Task<IEnumerable<Book>> GetAll()

            => await \_unitOfWork.BookRepository.GetAllAsync();

    }

Don’t not forget to register our service in **Program.cs**file:

builder.Services.AddTransient(typeof(IBookService), typeof(BookService));

We use **Dependency Injection**to inject the unit of work in the book service. Next, in each method of the service, we access the book repository through the unit of work. In the case where we’re adding a new book, we need call **CommitAsync()**to save all the changes to the database at once.

**Creating the Books Controller**

To be able to test this, let’s add a **BooksController.cs**file and expose a few endpoints.

[ApiController]

    [Route("[controller]")]

    public class BooksController : ControllerBase

    {

        public IBookService \_bookService { get; set; }

        public BooksController(IBookService bookService)

        {

            \_bookService = bookService;

        }

        [HttpGet(Name = "Books")]

        public async Task<IEnumerable<Book>> GetAll()

          => await \_bookService.GetAll();

        [HttpPost]

        public async Task AddBook([FromBody] BookDto book)

        {

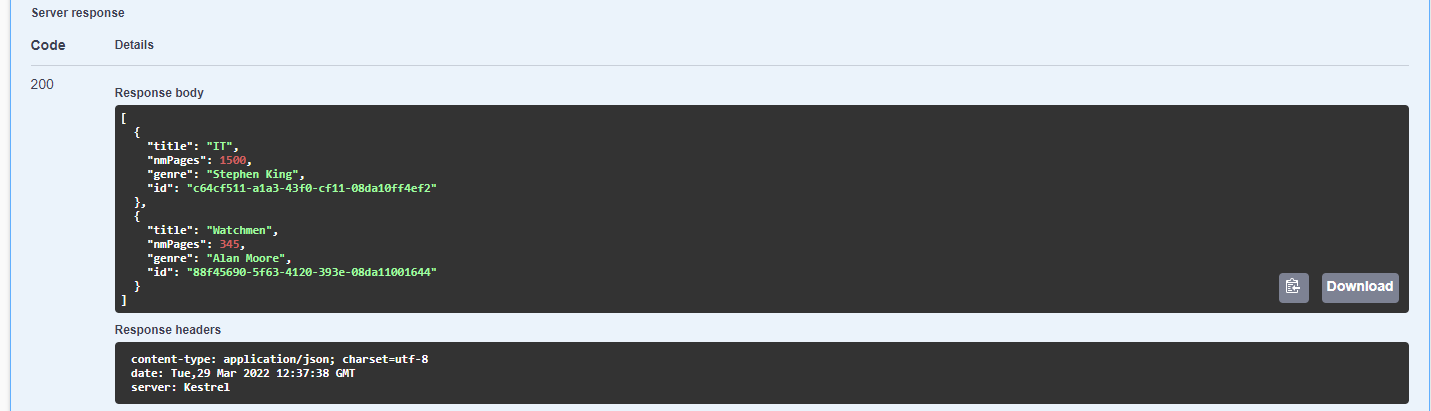
            await \_bookService.AddBook(book);

        }

    }

**Testing the Example**

If we make an **HTTP GET**request to the **/Books** endpoint, we get the following result:



***Perfect.*** We successfully made use of both the Repository and Unit of Work patterns in our application.