THE GENERIC REPOSITORY WITH C# AND ENTITY FRAMEWORK

written by **Kens Learning Curve** April 26, 2023

Source: Generic Repository in C# - Combine generics and repository (kenslearningcurve.com)

A short while ago I wrote about the Repository Pattern in C#, which decouples the data from the logic and has simple methods representing the CRUD. The article shows how we can create two repositories for movies and genres. But assume you have 15 entities… Do you need 15 repositories? Could be a bit overdone, right? And that is where the generic repository in C# comes to help.
In another previous article, I talked about generics in C#. Let's combine the generics and the repository in one.

Table Of Contents

Previously on…

In the previous article about the repository pattern in C#, I created two repositories: *MovieRepository* and *GenreRepository*. If you look carefully, you will see they are a lot alike. Okay, some methods are not used in the GenreRepository, but still. The only difference is the entity. The *MovieRepository* is solely built for the entity Movie, whereas the *GenreRepository* is totally committed to the Genre entity. This isn't a bad thing, but if you have 15 repositories you might have trouble maintaining them all.

One Repository To Rule Them All

A generic repository in C# has one purpose: To handle the data for all entities in the project. In other words: Create one repository instead of 2, or 15, and don't add new repositories as new entities are created.

To do this we start with an interface that will be generic. After that, we create the implementation of that interface. Last but not least, we are going to inject the generic repository into the classes where needed.

Interface

As with most stuff, it all starts with an interface. The interface for the generic repository in C# will have a few methods: Get, Get by ID, Create, Delete, Update, and SaveChanges. Yes, SaveChanges too, since I don't want to commit my changes to the database after each create, update, or delete.

```
public interface IRepository<TEntity> where TEntity : class
{
  IQueryable<TEntity> GetAll();
  TEntity Get(int id);
  TEntity Create(TEntity entity);
  void Delete(TEntity entity);
  void Update(TEntity entity);
  void SaveChanges();
}
```

I call the interface **IRepository**. A simple, yet powerful name. It tells me exactly what it does. This interface is generic by the generic data type TEntity. But what is the TEntity? It's a class because I put down the "where TEntity: class".
The only problem that could occur is with **Get(int id)**. TEntity is a generic data type, which means it doesn't contain any information. So how does it know where the ID is stored? We will solve this later on.
As shown in the code above, I can now use the generic data type in my methods. You can use it as a parameter or as a return type.

Implementation

The implementation of the **IRepository** is similar to the **MovieRepository**, except the repository has no clue which entities will be used. I create a new class and call it Repository. Then I make it generic and derive the IRepository.
Again, we need to inject the DataContext, which is connected to the dbContext. Below is the code for the implementation. I skipped the **Get(int id)** for now.

```
public class Repository<TEntity> : IRepository<TEntity> where TEntity : class
{
  private readonly DataContext context;
  public Repository(DataContext context)
  {
    this.context = context;
  }
  public TEntity Create(TEntity entity)
  {
    context.Add(entity);
    return entity;
  }
  public void Delete(TEntity entity)
```

```
    {
        context.Set<TEntity>().Remove(entity);
    }
    public TEntity Get(int id)
    {
        // Will be done later
        throw new NotImplementedException();
    }
    public IQueryable<TEntity> GetAll()
    {
        return context.Set<TEntity>();
    }
    public void SaveChanges()
    {
        context.SaveChanges();
    }
    public void Update(TEntity entity)
    {
        context.Entry(entity).State = Microsoft.EntityFrameworkCore.EntityState.Modified;
    }
}
```

I think most of the logic is self-explanatory. But look at the **Delete** and Create. Here I use **context.Set<TEntity>()**. This is equivalent to **context.Movies**, except now I don't know what entity it is. By using the **Set** Entity Framework will try to connect the type of TEntity (Movie or Genre) to a configured entity that is set in the **DataContext** class with **dbSet()**.

Where Is The ID?

The only method that is a bit different is the **Get(int id)**. TEntity is a generic data type and has no clue what properties the actual data type has, or hasn't. Each entity has different properties. A movie has a release date and not a name. A genre has a name, but not a title. But all entities have an ID.
So how can we find a particular entity by ID? Well, by giving the entities an interface. This interface will have one property: int Id { get; set; }

I created a new interface and called it **IEntity**. I place this interface in the domain project since it is connected to the entities, which are there too.

```
public interface IEntity
{
    int Id { get; set; }
}
```

Next, we need to connect that interface to the entities like this:

```
public class Movie : IEntity
{
   public int Id { get; set; }
   [Required]
   public string Title { get; set; }
   public string ReleaseDate { get; set; }
   [Required]
   public int GenreId { get; set; }
   public Genre Genre { get; set; }
}
public class Genre: IEntity
{
   public int Id { get; set; }
   public string Name { get; set; }
}
```

All that is left to do is tell the generic repository in C# that the TEntity has the interface **IEntity**. If the repository knows the interface it also knows the properties. With this information, we can create the body for the **Get(int id)**:

```
public class Repository<TEntity> : IRepository<TEntity> where TEntity : class, IEntity
{
   ....
   public TEntity Get(int id)
   {
      return context.Set<TEntity>().SingleOrDefault(entity => entity.Id == id);
   }
   ...
}
```

Using The Generic Repository In C#

Now that we have created the generic repository in C#, we can use it. First, I would delete all other repositories and interfaces (MovieRepository, IMovieRepository, GenreRepository, and IGenreRepository). This way you will get errors and know where to insert the generic repository.

The first spot that gives an error is *GenreService.cs*. You only have to change this:

```
private readonly IGenreRepository genreRepository;
public GenreService(IGenreRepository genreRepository)
{
   this.genreRepository=genreRepository;
}
```

To this:

```
private readonly IRepository<Genre> genreRepository;
public GenreService(IRepository<Genre> genreRepository)
{
  this.genreRepository=genreRepository;
}
```

Conveniently, I used the same names for the methods in both the **IGenreRepository** and **Repository**.
The **MovieService** class has both the **IMovieRepository** and the **IGenreRepository** injected. Let's remove them and replace them with the generic repository. That means we go from this:

```
private readonly IRepository<Genre> genreRepository;
public GenreService(IRepository<Genre> genreRepository)
{
  this.genreRepository=genreRepository;
}
```

To this:

```
private readonly IRepository<Movie> movieRepository;
private readonly IRepository<Genre> genreRepository;
public MovieService(IRepository<Movie> movieRepository, IRepository<Genre> genreRepository)
{
  this.movieRepository=movieRepository;
  this.genreRepository=genreRepository;
}
```

This does give a few errors because I used **GetAll()** in the generic repository, but **Get()** in the previous repositories. I renamed the **GetAll()** to **Get()**.
Another 'error' is the **movieRepository.Delete(id)** in the **Delete** method of **MovieService**. Since I don't want any logic or as less as possible, I want to make sure Entity Framework only does what it needs to do. This means that any validations should be done in the logic, **MovieService.Delete(int id)** in this case.
Therefore, I want the repository to only delete an entity and not look up the entity and then delete it. Chances are that the entity doesn't exist. I also don't know what error a generic repository should throw, since it has no idea what entity it will handle.

```
public void Delete(int id)
{
  if (id <= 0) throw new ArgumentOutOfRangeException(nameof(id));
  Movie toDelete = movieRepository.Get(id);
  if (toDelete == null)
    throw new Exception("Movie doesn't exist");
  movieRepository.Delete(toDelete);
```

```
    movieRepository.SaveChanges();
}
```

Injecting The Generic Repository

Only one thing left to do: Configure the dependency injection for the generic repository in C#. Before the generic repository, we need to configure each repository like this:

```
builder.Services.AddScoped<IMovieRepository, MovieRepository>();
builder.Services.AddScoped<IGenreRepository, GenreRepository>();
```

The idea behind the generic repository in C# is to make it easier and less code. But although the following configuration does work, it isn't the correct way:

```
builder.Services.AddScoped<IRepository<Movie>, Repository<Movie>>();
builder.Services.AddScoped<IRepository<Genre>, Repository<Genre>>();
```

What happens if we introduce a third entity? We need to configure the generic repository for that too. Nah, too much work. Let's make it easier:

```
builder.Services.AddScoped(typeof(IRepository<>), typeof(Repository<>));
```

Because it is generic we can configure it as a type, or rather **typeof**. This means that all types of **IRepository** need to go to the **Repository** class, no matter the generic data type. Even if we add or delete an entity, we don't have to change the configuration.
All done!

Conclusion On Generic Repository In C#

And there you have it: A generic repository in C#! Not that hard, is it?

Yes, there are some drawbacks. Especially when you want to do something, especially for a specific entity only. It would be best if you handled this in the logic of your application.

But it has great advantages over the 'normal' repositories. It's easier to maintain, for example. Changing something in the generic repository affects all implementations. If you have 10+ repositories you have to change the code 10+ times.

I have used this kind of repository for a long time now and it never failed me.