What is a Repository pattern and why should we use it?

With the Repository pattern, we create an abstraction layer between the data access and the business logic layer of an application. By using it, we are promoting a more loosely coupled approach to access our data from the database. Also, the code is cleaner and easier to maintain and reuse. Data access logic is in a separate class, or sets of classes called a repository, with the responsibility of persisting the application’s business model.

Implementing the repository pattern is our topic for this post. Additionally, this article has a strong relationship with EF Core, so we strongly recommend reading our [EF Core tutorial](https://code-maze.com/getting-started-with-efcore/) to get familiar or just a better understanding of that topic.

So let’s start.

**VIDEO**: Repository Pattern in ASP.NET Core Web API video.

If you want to see all the basic instructions and complete navigation for this series, please follow the following link: [Introduction page for this tutorial.](https://code-maze.com/net-core-series/)

For the previous part check out: [Creating .NET Core WebApi project – Custom logging in .NET Core](https://code-maze.com/net-core-web-development-part3/)

The source code is available for download at [.NET Core, Angular and MySQL. Part 4 – Source Code](https://github.com/CodeMazeBlog/.NET-Core-Series/tree/master/Part%204)

This post is divided into several sections:

* [Creating Models](https://code-maze.com/net-core-web-development-part4/#creatingModels)
* [Context Class and the Database Connection](https://code-maze.com/net-core-web-development-part4/#context)
* [Repository Pattern Logic](https://code-maze.com/net-core-web-development-part4/#repository)
* [Repository User Classes](https://code-maze.com/net-core-web-development-part4/#repoUserClasses)
* [Creating a Repository Wrapper](https://code-maze.com/net-core-web-development-part4/#repoWrapper)
* [Conclusion](https://code-maze.com/net-core-web-development-part4/#conclusion)

**Creating Models**

Let’s begin by creating a new Class Library ([.NET Core](https://code-maze.com/ultimate-aspnet-core-3-web-api/)) project named Entities and inside it a new folder with the name Models, which will contain all the model classes. Model classes will represent the tables inside the database and will serve us to map the data from the database to the [.NET Core](https://code-maze.com/ultimate-aspnet-core-3-web-api/). After that, we should reference this project to the main project.

In the Models folder, we are going to create two classes and modify them:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;    namespace Entities.Models  {      [Table("owner")]      public class Owner      {          public Guid OwnerId { get; set; }            [Required(ErrorMessage = "Name is required")]          [StringLength(60, ErrorMessage = "Name can't be longer than 60 characters")]          public string Name { get; set; }            [Required(ErrorMessage = "Date of birth is required")]          public DateTime DateOfBirth { get; set; }            [Required(ErrorMessage = "Address is required")]          [StringLength(100, ErrorMessage = "Address cannot be longer than 100 characters")]          public string Address { get; set; }            public ICollection<Account> Accounts { get; set; }      }  } |

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | using System.ComponentModel.DataAnnotations;  using System.ComponentModel.DataAnnotations.Schema;    namespace Entities.Models  {      [Table("account")]      public class Account      {          public Guid AccountId { get; set; }            [Required(ErrorMessage = "Date created is required")]          public DateTime DateCreated { get; set; }            [Required(ErrorMessage = "Account type is required")]          public string AccountType { get; set; }           [ForeignKey(nameof(Owner))]          public Guid OwnerId { get; set; }          public Owner Owner { get; set; }      }  } |

As you can see, there are two models decorated with the attribute Table(“tableName”). This attribute will configure the corresponding table name in the database. All the mandatory fields have the attribute [Required]and if we want to constrain the strings, we can use the[StringLength]attribute**.**In the Owner class, we have the Accounts property which suggests that one Owner is related to multiple Accounts. Additionally, we add the OwnerId and the Owner properties decorated with the [ForeignKey] attribute to state that one Account is related to only one Owner. If you want to learn more about the EF Core configuration, and we strongly suggest you do, visit the [Configuring Nonrelational Properties in EF Core.](https://code-maze.com/configuring-nonrelational-properties/)

**Context Class and the Database Connection**

Now, let us create the context class, which will be a middleware component for the communication with the database. It has DbSet properties that contain the table data from the database. For a better understanding of the Context class and DbSet properties and how they work with EF Core overall, you can read [Getting Started with EF Core](https://code-maze.com/getting-started-with-efcore/) article.

In the root of the Entities project, we are going to create the RepositoryContext class and modify it:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | using Entities.Models;  using Microsoft.EntityFrameworkCore;    namespace Entities  {      public class RepositoryContext: DbContext      {          public RepositoryContext(DbContextOptions options)              :base(options)          {          }            public DbSet<Owner> Owners { get; set; }          public DbSet<Account> Accounts { get; set; }      }  } |

Pay attention that you have to install Microsoft.EntityFrameworkCore package.

To enable communication between the [.NET core](https://code-maze.com/ultimate-aspnet-core-3-web-api/) part and the MySQL database, we have to install a third-party library named Pomelo.EntityFrameworkCore.MySql. In the main project, we can install it with the NuGet package manager or Package manager console (It is important to **Include prerelease** because it is still in the prerelease state).

After the installation, let’s open the appsettings.json file and add DB connection settings inside:

JavaScript



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | {    "Logging": {      "LogLevel": {        "Default": "Warning"      }    },    "mysqlconnection": {      "connectionString": "server=localhost;userid=root;password=yourpass;database=accountowner;"    },    "AllowedHosts": "\*"  } |

In the ServiceExtensions class, we are going to write the code for configuring the MySQL context.

First, let’s add the using directives and then add the method ConfigureMySqlContext:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using Microsoft.Extensions.Configuration;  using Microsoft.EntityFrameworkCore;    public static void ConfigureMySqlContext(this IServiceCollection services, IConfiguration config)  {      var connectionString = config["mysqlconnection:connectionString"];      services.AddDbContext<RepositoryContext>(o => o.UseMySql(connectionString));  } |

With the help of the IConfiguration config parameter, we can access the appsettings.json file and take all the data we need from it. Afterward, in the Startup class in the ConfigureServices method, add the context service to the IOC right above the services.AddControllers():

C#



|  |  |
| --- | --- |
| 1 | services.ConfigureMySqlContext(Configuration); |

**Repository Pattern Logic**

After establishing a connection with the database, it is time to create the generic repository that will serve us all the CRUD methods. As a result, all the methods can be called upon any repository class in our project.

Furthermore, creating the generic repository and repository classes that use that generic repository is not going to be the final step. We will go a  step further and create a wrapper around repository classes and inject it as a service. Consequently,  we can instantiate this wrapper once and then call any repository class we need inside any of our controllers. You will understand the advantages of this wrapper when we use it in the project.

First, let’s create an interface for the repository inside the Contracts project:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | namespace Contracts  {      public interface IRepositoryBase<T>      {          IQueryable<T> FindAll();          IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression);          void Create(T entity);          void Update(T entity);          void Delete(T entity);      }  } |

Right after the interface creation, we are going to create a new Class Library (.NET Core) project with the name Repository (add the reference from Contracts and Entities to this project), and inside the Repository project create the abstract class RepositoryBase which will implement the interface IRepositoryBase.

Reference this project to the main project too.

Let’s add the following code to the RepositoryBase class:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45 | using Contracts;  using Entities;  using Microsoft.EntityFrameworkCore;  using System;  using System.Collections.Generic;  using System.Linq;  using System.Linq.Expressions;    namespace Repository  {      public abstract class RepositoryBase<T> : IRepositoryBase<T> where T : class      {          protected RepositoryContext RepositoryContext { get; set; }            public RepositoryBase(RepositoryContext repositoryContext)          {              this.RepositoryContext = repositoryContext;          }            public IQueryable<T> FindAll()          {              return this.RepositoryContext.Set<T>().AsNoTracking();          }            public IQueryable<T> FindByCondition(Expression<Func<T, bool>> expression)          {              return this.RepositoryContext.Set<T>().Where(expression).AsNoTracking();          }            public void Create(T entity)          {              this.RepositoryContext.Set<T>().Add(entity);          }            public void Update(T entity)          {              this.RepositoryContext.Set<T>().Update(entity);          }            public void Delete(T entity)          {              this.RepositoryContext.Set<T>().Remove(entity);          }      }  } |

This abstract class, as well as IRepositoryBase interface, uses generic type **T** to work with. This type T gives even more reusability to the RepositoryBase class. That means we don’t have to specify the exact model (class) right now for the RepositoryBase to work with, we are going to do that later on.

**Repository User Classes**

Now that we have the RepositoryBase class, let’s create the user classes that will inherit this abstract class. Every user class will have its own interface, for additional model-specific methods. Furthermore, by inheriting from the RepositoryBase class they will have access to all the methods from the RepositoryBase. This way, we are separating the logic, that is common for all our repository user classes and also specific for every user class itself.

Let’s create interfaces in the Contracts project for our Owner and Account classes.

Don’t forget to add a reference from the Entities project. As soon as we do this, we can delete the Entities reference from the main project because it is now provided through the Repository project that already has the Contracts project referenced, with the Entities reference inside.

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using Entities.Models;    namespace Contracts  {      public interface IOwnerRepository : IRepositoryBase<Owner>      {      }  } |

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | using Entities.Models;    namespace Contracts  {      public interface IAccountRepository : IRepositoryBase<Account>      {      }  } |

Now, let’s  create a repository user classes in the Repository project:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Contracts;  using Entities;  using Entities.Models;    namespace Repository  {      public class OwnerRepository : RepositoryBase<Owner>, IOwnerRepository      {          public OwnerRepository(RepositoryContext repositoryContext)              :base(repositoryContext)          {          }      }  } |

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | using Contracts;  using Entities;  using Entities.Models;    namespace Repository  {      public class AccountRepository : RepositoryBase<Account>, IAccountRepository      {          public AccountRepository(RepositoryContext repositoryContext)              :base(repositoryContext)          {          }      }  } |

After these steps, we are finished with creating the repository and repository user classes. But there are still more things to be done.

**Creating a Repository Wrapper**

Let’s imagine if inside a controller we need to collect all the Owners and to collect only the certain Accounts (for example Domestic ones). We would need to instantiate OwnerRepository and AccountRepository classes and then call the FindAll and FindByCondition methods.

Maybe it’s not a problem when we have only two classes, but what if we need logic from 5 different classes or even more. Having that in mind, let’s create a wrapper around our repository user classes. Then place it into the IOC and finally inject it inside the controller’s constructor. Now, with that wrappers instance, we may call any repository class we need.

Let’s start by creating a new interface in the Contract project:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | namespace Contracts  {      public interface IRepositoryWrapper      {          IOwnerRepository Owner { get; }          IAccountRepository Account { get; }          void Save();      }  } |

After that, we are going to add a new class to the Repository project:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | using Contracts;  using Entities;    namespace Repository  {      public class RepositoryWrapper : IRepositoryWrapper      {          private RepositoryContext \_repoContext;          private IOwnerRepository \_owner;          private IAccountRepository \_account;            public IOwnerRepository Owner {              get {                  if(\_owner == null)                  {                      \_owner = new OwnerRepository(\_repoContext);                  }                    return \_owner;              }          }            public IAccountRepository Account {              get {                  if(\_account == null)                  {                      \_account = new AccountRepository(\_repoContext);                  }                    return \_account;              }          }            public RepositoryWrapper(RepositoryContext repositoryContext)          {              \_repoContext = repositoryContext;          }            public void Save()          {              \_repoContext.SaveChanges();          }      }  } |

As you can see, we are creating properties that will expose the concrete repositories and also we have the Save() method to be used after all the modifications are finished on a certain object. This is a good practice because now we can, for example, add two owners, modify two accounts and delete one owner, all in one method, and then just call the Save method once. All changes will be applied or if something fails, all changes will be reverted:

A simple example for the Save method

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | \_repository.Owner.Create(owner);  \_repository.Owner.Create(anotheOwner);  \_repository.Account.Update(account);  \_repository.Account.Update(anotherAccount);  \_repository.Owner.Delete(oldOwner);    \_repository.Save(); |

In the ServiceExtensions class, we are going to add this code:

C#



|  |  |
| --- | --- |
| 1  2  3  4 | public static void ConfigureRepositoryWrapper(this IServiceCollection services)  {      services.AddScoped<IRepositoryWrapper, RepositoryWrapper>();  } |

And in the Startup class inside the ConfigureServices method, above the services.AddControllers() line, add this code:

C#



|  |  |
| --- | --- |
| 1 | services.ConfigureRepositoryWrapper(); |

Excellent.

**Testing**

All we have to do is to test this code the same way we did with our custom logger [in part3 of this series.](https://code-maze.com/net-core-web-development-part3/)

Inject the RepositoryWrapper service inside the WeatherForecast controller and call any method from the RepositoryBase class:

C#



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | [ApiController]  [Route("[controller]")]  public class WeatherForecastController : ControllerBase  {      private IRepositoryWrapper \_repoWrapper;        public ValuesController(IRepositoryWrapper repoWrapper)      {          \_repoWrapper = repoWrapper;      }      // GET api/values      [HttpGet]      public IEnumerable<string> Get()      {          var domesticAccounts = \_repoWrapper.Account.FindByCondition(x => x.AccountType.Equals("Domestic"));          var owners = \_repoWrapper.Owner.FindAll();            return new string[] { "value1", "value2" };      }  } |

Place the breakpoint inside the Get() method and you’ll see the data returned from the database.

In the [next part](https://code-maze.com/net-core-web-development-part5/), we are going to show you how to restrict access to the RepositoryBase methods from the controller, if you don’t want them to be exposed here.

*We have created our Repository Pattern synchronously but it could be done asynchronously as well. If you want to learn how to do that you can visit*[*Implementing Async Repository in .NET Core.*](https://code-maze.com/async-generic-repository-pattern/)*Although we strongly recommend finishing all the parts from this series for an easier understanding of the project’s business logic.*

**Conclusion**

The Repository pattern increases the level of abstraction in your code. This may make the code more difficult to understand for developers who are unfamiliar with the pattern. But once you are familiar with it, it will reduce the amount of redundant code and make the logic much easier to maintain.

In this post you have learned:

* What is repository pattern
* How to create models and model attributes
* How to create context class and database connection
* The right way to create repository logic
* And the way to create a wrapper around your repository classes

Thank you all for reading this post and I hope you read some useful information in it.

See you soon[in the next article](https://code-maze.com/net-core-web-development-part5/), where we will use repository logic to create HTTP requests.