

# Repository Pattern

---

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.

## Creating Models

Let's begin by creating a new Class Library (.NET Core) project named **Entities** and inside it create a folder with the name `Models`, which will contain all the model classes. Model classes will represent the tables inside the database and will map the data from the database to the .NET Core. After that, reference this project to the main project.

In the **Models** folder create two classes and add this code:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Entities.Models
{
    [Table("owner")]
    public class Owner
    {
        [Key]
        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 loner then 100
characters")]
        public string Address { get; set; }
    }
}

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace Entities.Models
{
    [Table("account")]
```

```
public class Account
{
    [Key]
    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; }

    [Required(ErrorMessage = "Owner Id is required")]
    public Guid OwnerId { 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. Also, every primary key has the decoration attribute `Key` that configures the primary key of the table. All the mandatory fields have the attribute `[Required]` and if you want to constrain the strings, you can use the attribute `[StringLength]`.

Notice that these models are clean; they have only those properties that match the columns in the tables. Later, we will create DTO classes with the properties to connect one owner with all of its accounts or to connect a single account with its owner.

## Context Class and the Database Connection

The context class will be a middleware component for communication with the database. It has `DbSet` properties that contain the table data from the database.

In the root of `Entities` project create the `RepositoryContext` class and modify it:

```
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; }
    }
}
```

To enable communication between the .NET core part and the MySQL database, we need to install a third-party library named `Pomelo.EntityFrameworkCore.MySql`. In the main project, install it with the NuGet package manager or Package manager console.

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

```
{
  "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, add the `using` directives and then add the method `ConfigureMySQLContext`:

```
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, you can access the `appsettings.json` file and take all the data you need from it. Afterward, in the `Startup` class in the `ConfigureServices` method, add the context service to the IOC right above `services.AddMvc()`:

```
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 your 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.

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

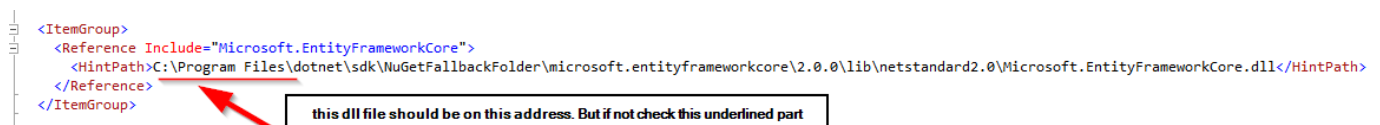
```
namespace Contracts
{
    public interface IRepositoryBase<T>
    {
        IEnumerable<T> FindAll();
        IEnumerable<T> FindByCondition(Expression<Func<T, bool>> expression);
        void Create(T entity);
        void Update(T entity);
        void Delete(T entity);
        void Save();
    }
}
```

After the interface creation, you are going to create a new Class Library (.NET Core) project with the name **Repository** (add the reference from **Contracts** 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.

---

**Tip: If you have problems with referencing EntityFrameworkCore in your assemblies for this new **Repository** project, you need to right-click on the **Repository** project, click on the Unload Project. When the project unloads, right click on it and choose Edit Repository.csproj. Add this code to the opened file:**



```
<ItemGroup>
  <Reference Include="Microsoft.EntityFrameworkCore">
    <HintPath>C:\Program Files\dotnet\sdk\NuGetFallbackFolder\microsoft.entityframeworkcore\2.0.0\lib\netstandard2.0\Microsoft.EntityFrameworkCore.dll</HintPath>
  </Reference>
</ItemGroup>
```

this dll file should be on this address. But if not check this underlined part

**Finally, save the file, right-click on the **Repository** project and click Reload Project.**

---

Add the following code to the **RepositoryBase** class:

```
using Contracts;
using Entities;
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 IEnumerable<T> FindAll()
        {
            return this.RepositoryContext.Set<T>();
        }

        public IEnumerable<T> FindByCondition(Expression<Func<T, bool>>
expression)
        {
            return this.RepositoryContext.Set<T>().Where(expression);
        }

        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);
        }

        public void Save()
        {
            this.RepositoryContext.SaveChanges();
        }
    }
}

```

This abstract class, as well as  **IRepositoryBase** interface, use generic type **T**. This means you don't have to specify exact model (class) right now for the **RepositoryBase** to work with, you'll do that later on.

## Repository User Classes

Now that you have the **RepositoryBase** class, create the user classes that will inherit from 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 from the logic that is specific to every user class itself.

Let's create the interfaces in the **Contracts** project for the **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.

```
using Entities.Models;

namespace Contracts
{
    public interface IOwnerRepository : IRepositoryBase<Owner>
    {
    }
}

using Entities.Models;

namespace Contracts
{
    public interface IAccountRepository: IRepositoryBase<Account>
    {
    }
}
```

Now create repository user classes in the **Repository** project:

```
using Contracts;
using Entities;
using Entities.Models;

namespace Repository
{
    public class OwnerRepository: RepositoryBase<Owner>, IOwnerRepository
    {
        public OwnerRepository(RepositoryContext repositoryContext)
            :base(repositoryContext)
        {
        }
    }
}

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.

## Creating a Repository Wrapper

Imagine if inside a controller you needed to collect all the Owners and to collect only certain Accounts (for example, Domestic). You would need to instantiate `OwnerRepository` and `AccountRepository` classes and then call the `FindAll` and `FindByCondition` methods.

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

Create a new interface in the `Contracts` project:

```
namespace Contracts  
{  
    public interface IRepositoryWrapper  
    {  
        IOwnerRepository Owner { get; }  
        IAccountRepository Account { get; }  
    }  
}
```

Add a new class to the `Repository` project:

```
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;
    }
}

```

In the `ServiceExtensions` class add this code:

```

public static void ConfigureRepositoryWrapper(this IServiceCollection services)
{
    services.AddScoped<IRepositoryWrapper, RepositoryWrapper>();
}

```

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

```

services.ConfigureRepositoryWrapper();

```

We can test this code the same way we did with the custom logger.

Inject the `RepositoryWrapper` service in `ValuesController` and call any method from the `RepositoryBase` class:

```

[Route("api/[controller]")]
[ApiController]
public class ValuesController : 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 a breakpoint in the `Get()` method and you'll see the data returned from the database.

## 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.

Lessons learned:

- What is a repository pattern
- How to create models and model attributes
- How to create a context class and database connection
- How to create repository logic
- How to create a wrapper around your repository classes