Build an ASP.NET Core MVC App with EF Core One-Day Hands-On Lab

Lab 5

This lab walks you through creating the repositories and their interfaces for the data access library. Prior to starting this lab, you must have completed Lab 4.

Part 1: Create the Base Repository

Step 1: Create the Base Repository Interface

While the `DbContext` can be considered an implementation of the repository pattern, it's better to create specific repositories for the entities. These repos will be added into the ASP.NET Core Dependency Injection container later today.

- Create a new folder in the `AutoLot.Dal` project named Repos. Create a subfolder under that named Base.
- Add a new interface to the `Base` folder named `IRepo.cs`
- Update the using statements to the following:

```
using System;
using System.Collections.Generic;
```

- Update the code for the `IRepo.cs` class to the following:

```
namespace AutoLot.Dal.Repos.Base
{
  public interface IRepo<T>: IDisposable
  {
    int Add(T entity, bool persist = true);
    int AddRange(IEnumerable<T> entities, bool persist = true);
    int Update(T entity, bool persist = true);
    int UpdateRange(IEnumerable<T> entities, bool persist = true);
    int Delete(int id, byte[] timeStamp, bool persist = true);
    int Delete(T entity, bool persist = true);
    int DeleteRange(IEnumerable<T> entities, bool persist = true);
    T? Find(int? id);
    T? FindAsNoTracking(int id);
    T? FindIgnoreQueryFilters(int id);
    IEnumerable<T> GetAll();
    IEnumerable<T> GetAllIgnoreQueryFilters();
    int SaveChanges();
  }
}
```

Step 2: Create the Base Repository

- Add a new class to the `Repos/Base` folder named `BaseRepo.cs`
- Add the following using statements to the class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Dal.Exceptions;
using AutoLot.Models.Entities.Base;
using Microsoft.EntityFrameworkCore;
```

- Make the class `public` and `abstract`, generic, and implement `IRepo<T>`:

```
public abstract class BaseRepo<T> : IRepo<T> where T : BaseEntity, new() { }
```

- Add a Boolean flag for disposing of the context, a protected variable to represent the `DbSet` for the derived repo, and a public property to hold the `ApplicationDbContext`:

```
private readonly bool _disposeContext;
public DbSet<T> Table {get;}
public ApplicationDbContext Context { get; }
```

- Add a constructor that takes an instance of the `ApplicationDbContext` that sets the `Context` and `Table` properties. A `DbSet<T>` property can be referenced using the `Context.Set<T>()` method.
  NOTE: This constructor is used by the DI container in ASP.NET Core. The ASP.NET Core DI container manages lifetime, so set the flag for context disposal to false.

```
protected BaseRepo(ApplicationDbContext context)
{
  Context = context;
  Table = Context.Set<T>();
  _disposeContext = false;
}
```

- Add another constructor that takes in `DbContextOptions`, calls the previous constructor while creating a new `ApplicationDbContext` using the options. Since this is not used by DI, set the disposal flag to true.

```
protected BaseRepo(DbContextOptions<ApplicationDbContext> options)
  : this(new ApplicationDbContext(options))
{
  _disposeContext = true;
}
```


- Implement the `Dispose` pattern:

```
public void Dispose()
{
  Dispose(true);
  GC.SuppressFinalize(this);
}

private bool _isDisposed;
protected virtual void Dispose(bool disposing)
{
  if (_isDisposed) { return; }
  if (disposing)
  {
    if (_disposeContext)
    {
      Context.Dispose();
    }
  }
  _isDisposed = true;
}

~BaseRepo()
{
  Dispose(false);
}
```

- Implement the three `Find` variations using the built-in `Find` method, the `AsNoTrackingWithIdentityResolution` method, as well as the `IgnoreQueryFilters` method:

```
public virtual T? Find(int? id) => Table.Find(id);
public virtual T? FindAsNoTracking(int id)
  => Table.AsNoTrackingWithIdentityResolution().FirstOrDefault(x => x.Id == id);
public T? FindIgnoreQueryFilters(int id)
  => Table.IgnoreQueryFilters().FirstOrDefault(x => x.Id == id);
```

- `TheGetAll` methods are virtual, allowing for the derived repos to override them.

```
public virtual IEnumerable<T> GetAll() => Table;
public virtual IEnumerable<T> GetAllIgnoreQueryFilters() => Table.IgnoreQueryFilters();
```

- The Add[Range], Update[Range], and Delete[Range] methods all take an optional parameter to signal if `SaveChanges` should be called immediately or not.

```
public virtual int Add(T entity, bool persist = true)
{
  Table.Add(entity);
  return persist ? SaveChanges() : 0;
}
public virtual int AddRange(IEnumerable<T> entities, bool persist = true)
{
  Table.AddRange(entities);
  return persist ? SaveChanges() : 0;
}
public virtual int Update(T entity, bool persist = true)
{
  Table.Update(entity);
  return persist ? SaveChanges() : 0;
}
public virtual int UpdateRange(IEnumerable<T> entities, bool persist = true)
{
  Table.UpdateRange(entities);
  return persist ? SaveChanges() : 0;
}
public int Delete(int id, byte[] timeStamp, bool persist = true)
{
  Context.Entry(new T {Id = id, TimeStamp = timeStamp}).State = EntityState.Deleted;
  return persist ? SaveChanges() : 0;
}
public virtual int Delete(T entity, bool persist = true)
{
  Table.Remove(entity);
  return persist ? SaveChanges() : 0;
}
public virtual int DeleteRange(IEnumerable<T> entities, bool persist = true)
{
  Table.RemoveRange(entities);
  return persist ? SaveChanges() : 0;
}
```

- The `BaseRepo SaveChanges` method shells out to the `Context.SaveChanges`

```
public int SaveChanges()
{
  try
  {
      return Context.SaveChanges();
  }
  catch (CustomException ex)
  {
    //Should handle intelligently - already logged
    throw;
  }
  catch (Exception ex)
  {
    //Should log and handle intelligently
    throw new CustomException("An error occurred updating the database", ex);
  }
}
```

Part 2: Add the Entity Specific Repo Interfaces

There is an interface and repo for each model that uses the base repository for the common functionality. Each specific repo extends or overwrites that base functionality as needed.

Step 1: Create the Interface Files

- Create a new folder under the `Repos` folder named `Interfaces`.
- Create the following files in the `Interfaces` folder:

```
ICarRepo.cs
ICreditRiskRepo.cs
ICustomerRepo.cs
IMakeRepo.cs
IOrderRepo.cs
```

Step 2: Define the ICarRepo Interface

The `Category Repo` uses the methods in the base repo and does not add any methods.

- Add the following using statements to the `ICarRepo.cs` interface:

```
using System.Collections.Generic;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
```

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces
{
    public interface ICarRepo : IRepo<Car>
    {
        IEnumerable<Car> GetAllBy(int makeId);
    }
}
```

Step 3: Define the ICreditRiskRepo Interface

- Add the following using statements to the `ICreditRiskRepo.cs` interface:

```
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
```

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces
{
  public interface ICreditRiskRepo : IRepo<CreditRisk> { }
}
```

Step 4: Define the ICustomerRepo Interface

- Add the following using statements to the `ICustomerRepo.cs` interface:

```
using AutoLot.Dal.Repos.Base;
using AutoLot.Models.Entities;
```

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces
{
  public interface ICustomerRepo : IRepo<Customer> { }
}
```

Step 5: Define the IMakeRepo Interface

- Add the following using statements to the `IMakeRepo.cs` interface:

```
using System.Collections.Generic;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
```

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces
{
  public interface IMakeRepo : IRepo<Make>
  {
    IEnumerable<Make> GetOrderByMake();
  }
}
```

Step 6: Define the IOrderRepo Interface

- Add the following using statements to the `IOrderRepo.cs` interface:

```
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
```

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces
{
  public interface IOrderRepo : IRepo<Order> { }
}
```


Part 3: Implement the Entity Specific Repos

Step 1: Create the Class Files

- Create the following files in the Repos folder:

```
CarRepo.cs
CreditRiskRepo.cs
CustomerRepo.cs
MakeRepo.cs
OrderRepo.cs
```

Step 2: Implement the CarRepo Class

- Add the following using statements to the `CarRepo.cs` class:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

- Make the class public, inherit `BaseRepo<Car>`, and implement `ICarRepo`. Add the two required constructors

```
namespace AutoLot.Dal.Repos
{
  public class CarRepo : BaseRepo<Car>, ICarRepo
  {
    public CarRepo(ApplicationDbContext context) : base(context) { }
    internal CarRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
  }
}
```

- Add overrides for the `GetAll` methods:

```
public override IEnumerable<Car> GetAll()
  => Table.Include(c => c.MakeNavigation).OrderBy(o => o.PetName);
public override IEnumerable<Car> GetAllIgnoreQueryFilters()
  => Table.Include(c => c.MakeNavigation).OrderBy(o => o.PetName).IgnoreQueryFilters();
```

- Add override for the `Find` method to include the Make information:

```
public override Car? Find(int? id) => Table.IgnoreQueryFilters()
       .Where(x => x.Id == id)
       .Include(m => m.MakeNavigation)
       .FirstOrDefault();
```


- Add method to get all by Make ID:

```
public IEnumerable<Car> GetAllBy(int makeId)
{
  Context.MakeId = makeId;
  return Table.Include(c => c.MakeNavigation).OrderBy(c => c.PetName);
}
```

Step 3: Implement the CreditRiskRepo Class

- Add the following using statements to the `CreditRiskRepo.cs` class:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

- Make the class `public`, inherit `BaseRepo<CreditRisk>`, and implement `ICreditRiskRepo`. Add the two required constructors as follows:

```
namespace AutoLot.Dal.Repos
{
  public class CreditRiskRepo : BaseRepo<CreditRisk>, ICreditRiskRepo
  {
    public CreditRiskRepo(ApplicationDbContext context) : base(context) { }
    internal CreditRiskRepo(DbContextOptions<ApplicationDbContext> options) : base(options) {}
  }
}
```

Step 4: Implement the CustomerRepo Class

- Add the following using statements to the `CustomerRepo.cs` class:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

- Make the class public, inherit `BaseRepo<Customer>`, and implement `ICustomerRepo`. Add the two required constructors, as well as the override to `GetAll`, as follows:

```
namespace AutoLot.Dal.Repos
{
  public class CustomerRepo : BaseRepo<Customer>, ICustomerRepo
  {
    public CustomerRepo(ApplicationDbContext context) : base(context) { }
    internal CustomerRepo(DbContextOptions<ApplicationDbContext> options) : base(options) {}
    public override IEnumerable<Customer> GetAll()
      => Table.Include(c => c.Orders).OrderBy(o => o.PersonalInformation.LastName);
  }
}
```


Step 5: Implement the MakeRepo Class

- Add the following using statements to the `MakeRepo.cs` class:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.EfStructures;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

- Make the class public, inherit `BaseRepo<Make>`, and implement `IMakeRepo`. Add the required constructors and the two overrides for the `GetAll` methods, as follows:

```
namespace AutoLot.Dal.Repos
{
  public class MakeRepo : BaseRepo<Make>, IMakeRepo
  {
    public MakeRepo(ApplicationDbContext context) : base(context) { }
    internal MakeRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }

    public override IEnumerable<Make> GetAll()=> Table.OrderBy(m => m.Name);
    public override IEnumerable<Make> GetAllIgnoreQueryFilters()
      => Table.IgnoreQueryFilters().OrderBy(m => m.Name);

    public IEnumerable<Make> GetOrderByMake()
    {
      var orderByMake = Table.IgnoreQueryFilters()
        .Include(m => m.Cars.Where(c => c.Orders.Any()));
      var q = orderByMake.ToQueryString();
      return orderByMake;
    }
  }
}
```


Step 6: Implement the OrderRepo Class

- Add the following using statements to the `OrderRepo.cs` class:

```
using AutoLot.Dal.EfStructures;
using AutoLot.Models.Entities;
using AutoLot.Dal.Repos.Base;
using AutoLot.Dal.Repos.Interfaces;
using Microsoft.EntityFrameworkCore;
```

- Make the class public, inherit `BaseRepo<Order>`, and implement `IOrderRepo`. Add the standard constructors, as shown below.

```
namespace AutoLot.Dal.Repos
{
  public class OrderRepo : BaseRepo<Order>, IOrderRepo
  {
    public OrderRepo(ApplicationDbContext context) : base(context) { }
    internal OrderRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
  }
}
```

Summary

The lab created all of the repositories and their interfaces.

Next steps

In the next part of this tutorial series, you will create a data initializer.