Build an EF and ASP.NET Core 3.0 App HOL

Lab 4

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 3.

Part 1: Update the DbContext

Overriding the SaveChanges method in the DbContext allows for encapsulation of error handling.

Step 1: Override the Save Changes method

1) Add the following using statements to the top of the ApplicationDbContext.cs file:

```
using Microsoft.Data.SqlClient;
using Microsoft.EntityFrameworkCore.Storage;
using AutoLot.Dal.Exceptions;
```

2) Navigate to the bottom of the file. Add the following code, which overrides one of the SaveChanges methods:

```
public override int SaveChanges()
{
  try
  {
    return base.SaveChanges();
  }
  catch (DbUpdateConcurrencyException ex)
  {
    //A concurrency error occurred - should log and handle intelligently
    throw new CustomConcurrencyException("A concurrency error happened.", ex);
  }
  catch (RetryLimitExceededException ex)
  {
    //DbResiliency retry limit exceeded - should log and handle intelligently
    throw new CustomRetryLimitExceededException("There is a problem with SQl Server.", ex);
  }
  catch (DbUpdateException ex)
  {
    //Should log and handle intelligently
    throw new CustomDbUpdateException("An error occurred updating the database", ex);
  }
  catch (Exception ex)
  {
    //Should log and handle intelligently
    throw new CustomException("An error occurred updating the database", ex);
  }
}
```

Part 2: Add the Repositories

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.

Step 1: Create the folders and base class and interface

1) Create a new directory named Repos in the AutoLot.Dal project. In this directory, add a child directory named Base. Add a class named BaseRepo.cs and an interface named IRepo.cs to the Base directory.

Step 2: Update Base Repository Interface

1) Add the following using statements to the IRepoBase interface:

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

2) Update the interface code to match the following:

```
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();
  void ExecuteQuery(string sql, object[] sqlParametersObjects);
  int SaveChanges();
}
```

Step 1: Update Base Repository

1) Add the following using statements to the IRepoBase interface:

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

2) 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 `StoreContext`:

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

3) Add a constructor that takes an instance of the `StoreContext` 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 RepoBase(StoreContext context)
{
  Context = context;
  Table = Context.Set<T>();
  _disposeContext = false;
}
```

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

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

5) 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);
}
```

6) Implement the three `Find` variations show using the built-in `Find` method, the `AsNoTracking` method, as well as the `IgnoreQueryFilters` method:

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

7) `TheGetAll` methods are virtual, allowing for the derived repos to override them. The first just returns the records in database order, the second uses LINQ expressions to return the records in a specific order.

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

8) The ExecuteQuery is used to call stored procedures with parameters:

```
public void ExecuteQuery(string sql, object[] sqlParametersObjects)
  => Context.Database.ExecuteSqlRaw(sql, sqlParametersObjects);
```

9) The Add[Range], Update[Range], and Delete[Range] methods all take an optional parameter to signal if `SaveChanges` should be called immediately or not.
Note: The EF method name to delete a record is `Remove`, since it is technically just removing the instance from the `DbSet<T>`. `Delete` doesn't happen until `SaveChanges` is called. I use the name `Delete` in my repos because it is clearer to me.

```
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 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;
}
```

10) The `RepoBase SaveChanges` method calls 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 3: Add the Entity Specific 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: Add the ICarRepo Interface

1) Create a new directory named Interfaces under the Repos directory. Add a new interface named ICarRepo.cs. Update the using statements as follows:

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

2) Update the interface to implement the strongly typed IRepo and add one method as follows:

```
public interface ICarRepo : IRepo<Car>
{
  IEnumerable<Car> GetAllBy(int makeId);
}
```

Step 2: Add the ICreditRiskRepo Interface

1) Add a new interface named ICreditRiskRepo.cs. Update the using statements as follows:

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

2) Update the interface to implement the strongly typed IRepo as follows:

```
public interface ICreditRiskRepo : IRepo<CreditRisk> { }
```

Step 3: Add the ICustomerRepo Interface

1) Add a new interface named ICustomerRepo.cs. Update the using statements as follows:

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

2) Update the interface to implement the strongly typed IRepo as follows:

```
public interface ICustomerRepo : IRepo<Customer> { }
```

Step 4: Add the IMakeRepo Interface

1) Add a new interface named IMakeRepo.cs. Update the using statements as follows:

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

2) Update the interface to implement the strongly typed IRepo as follows:

```
public interface IMakeRepo : IRepo<Make> { }
```

Step 5: Add the IOrderRepo Interface

3) Add a new interface named IOrderRepo.cs. Update the using statements as follows:

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

4) Update the interface to implement the strongly typed IRepo as follows:

```
public interface IOrderRepo : IRepo<Order> { }
```

Part 4: Add the Entity Specific Repo Implementations

Step 1: Add the CarRepo Class

1) Add a new class named CarRepo.cs into the Repos folder. Update the using statements as follows:

```
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;
```

2) Inherit from the strongly typed BaseRepo and implement the ICarRepo interface. Add the two constructors required for the base class:

```
public class CarRepo : BaseRepo<Car>, ICarRepo
{
  public CarRepo(ApplicationDbContext context) : base(context) { }

  internal CarRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

3) Add overrides for the `GetAll` methods to include the Make navigation property and sorts by PetName:

```
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();
```

4) Add the GetAllBy method. Note that this sets the Context MakeId property that is used in the Car global query filter:

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

5) Override the Find method to include the Make navigation property and ignores the global query filter, as follows:

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

Step 2: Add the CreditRiskRepo Class

1) Add a new class named CreditRiskRepo.cs into the Repos folder. Update the using statements as follows:

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

2) Inherit from the strongly typed BaseRepo and implement the ICreditRiskRepo interface. Add the two constructors required for the base class:

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

Step 3: Update the CustomerRepo Class

1) Add a new class named CustomerRepo.cs into the Repos folder. Update the using statements as follows:

```
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;
```

2) Inherit from the strongly typed BaseRepo and implement the ICustomerRepo interface. Add the two constructors required for the base class:

```
public class CustomerRepo : BaseRepo<Customer>, ICustomerRepo
{
  public CustomerRepo(ApplicationDbContext context) : base(context) { }
  internal CustomerRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

3) Add an override for the GetAll method to include the Orders navigation property and sort by LastName:

```
public override IEnumerable<Customer> GetAll()
  => Table.Include(c => c.Orders).OrderBy(o => o.PersonalInformation.LastName);
```

Step 4: Update the MakeRepo Class

1) Add a new class named MakeRepo.cs into the Repos folder. Update the using statements as follows:

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

2) Inherit from the strongly typed BaseRepo and implement the IMakeRepo interface. Add the two constructors required for the base class:

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

Step 5: Add the OrderRepo Class

1) Add a new class named OrderRepo.cs into the Repos folder. Update the using statements as follows:

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

2) Inherit from the strongly typed BaseRepo and implement the IOrderRepo interface. Add the two constructors required for the base class:

```
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.