

.NET 10 App Dev Hands-On Lab

EF Lab 5 - Repositories

This creates repositories and interfaces for the data access library. You must have completed EF Lab 4 before starting this lab.

Part 1: Create the Base Repositories

Copilot Agent Mode

Setup Prompt: Always use file scoped namespaces. Always combine attributes on a single line when possible. The project does not use nullable reference types. There is a GlobalUsings.cs file that includes common usings, don't include using statements in new files if they are already in the globalusings.cs file. I prefer expression bodied members when possible. Single line if statements should still use braces. Use ternary operators when appropriate. Use internal over private. All classes and methods are public unless told otherwise. Don't add a constructor unless instructed to do so. Use primary constructors when possible and don't declare a class level variable if the parameter from the constructor can be used. Don't initialize properties unless instructed to do so.

Prompt: In the Autolot.Dal project, create a new folder named Repos. In that folder, create two folders, one named Base and the other named Interfaces. Add a new folder named Base to the Interfaces folder. In the Interfaces/Base folder, create an interface named IBaseViewRepo, make it generic with a constraint of class and new(), and implement IDisposable. Add with the following method signatures:

```
ApplicationContext Context { get; }  
IQueryable<T> ExecuteSqlString(string sql)  
IQueryable<T> GetAll()  
IQueryable<T> GetAllIgnoreQueryFilters()  
IQueryable<T> GetAllIgnoreQueryFilters(string[] filtersToIgnore)
```

Prompt: Create an interface named IBaseRepo<T> that implements IBaseViewRepo and has a generic constraint of BaseEntity and new() in the Interfaces/Base folder.

Implement the following methods signatures:

```
T Find(int? id);  
T FindAsNoTracking(int id);  
T FindIgnoreQueryFilters(int id);  
void ExecuteParameterizedQuery(string sql, object[] sqlParametersObjects);  
int Add(T entity, bool persist = true);  
int AddRange(IQueryable<T> entities, bool persist = true);  
int Update(T entity, bool persist = true);  
int UpdateRange(IQueryable<T> entities, bool persist = true);  
int Delete(int id, long timeStamp, bool persist = true);  
int Delete(T entity, bool persist = true);  
int DeleteRange(IQueryable<T> entities, bool persist = true);  
int ExecuteBulkUpdate(Expression<Func<T, bool>> whereClause,  
    Action<UpdateSettersBuilder<T>> setPropertyCalls);  
int ExecuteBulkDelete(Expression<Func<T, bool>> whereClause);  
int SaveChanges();
```

EF Lab 5 - Repositories

Prompt: Add the following global usings to the GlobalUsings.cs file in the AutoLot.Dal and the AutoLot.Models projects if they do not already exist (sorted alphabetically. Don't remove any existing global using statements).

```
global using AutoLot.Dal.Repos;
global using AutoLot.Dal.Repos.Interfaces;
global using AutoLot.Dal.Repos.Interfaces.Base;
```

Prompt: In the Repos/Base folder, create an abstract class named BaseViewRepo<T> that implements IBaseViewRepo with a generic constraint of class and new(). Add three fields:

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

Implement a constructor that takes an ApplicationDbContext and sets disposeContext to false and sets the Table field.

Implement a protected constructor that takes an DbContextOptions that calls the first constructor and sets disposeContext to true.

Implement IDisposable and the Dispose pattern to dispose of the context if _disposeContext is true.

Include GC.SuppressFinalize in the Dispose method. Make the dispose method virtual.

Implement the rest of the methods from the interface. Make GetAll and both

GetAllIgnoreQueryFilters methods virtual. GetAllIgnoreQueryFilters with the parameter should call Table.IgnoreQueryFilters(filtersToIgnore).

Prompt: Add the following global usings to the GlobalUsings.cs file in the AutoLot.Dal and the AutoLot.Models projects if they do not already exist (sorted alphabetically. Don't remove any existing global using statements).

```
global using AutoLot.Dal.Repos.Base;
```

Prompt: In the Repos/Base folder, create an abstract class named BaseRepo<T> that inherits from BaseViewRepo<T> and implements IBaseRepo<T> with a generic constraint of BaseEntity and new(). The SaveChanges method should call the Context's SaveChanges method. Implement the rest of the methods from the interface, marking all interface methods as virtual.

For the Find method, return null if id is null, else call Table.Find(id.Value).

Any AsNoTracking methods should call AsNotrackingWithIdentityResolution.

All of the crud methods take an optional persist parameter that should call SaveChanges if true.

Prompt: Implement the Delete(int id, long timeStamp, bool persist=true), ExecuteBulkUpdate, ExecuteBulkDelete, and SaveChanges as follows:

```
public int Delete(int id, long timeStamp, bool persist = true)
{
    var entity = new T {Id = id, TimeStamp = timeStamp};
    Context.Entry(entity).State = EntityState.Deleted;
    return persist ? SaveChanges() : 0;
}
public int ExecuteBulkUpdate(Expression<Func<T, bool>> whereClause,
    Action<UpdateSettersBuilder<T>> setPropertyCalls)
    => Table.IgnoreQueryFilters().Where(whereClause).ExecuteUpdate(setPropertyCalls);

public int ExecuteBulkDelete(Expression<Func<T, bool>> whereClause)
    => Table.IgnoreQueryFilters().Where(whereClause).ExecuteDelete();
```

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

Manual

Step 1: Create the Base Repository Interfaces

- Create a new folder in the AutoLot.Dal project named `Repos`. Create two subfolders under that directory named `Base` and `Interfaces`. Add a new folder named `Base` to the `Interfaces` folder, and in that folder, add a new interface named `IBaseViewRepo.cs`.
- Update the code for the `IBaseViewRepo.cs` interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces.Base;
public interface IBaseViewRepo<T> : IDisposable where T : class, new()
{
    ApplicationDbContext Context { get; }
    IQueryable<T> ExecuteSqlString(string sql);
    IQueryable<T> GetAll();
    IQueryable<T> GetAllIgnoreQueryFilters();
    IQueryable<T> GetAllIgnoreQueryFilters(string[] filtersToIgnore);
}
```

- Add a new interface named `IBaseRepo.cs` and update it to the following:

```
namespace AutoLot.Dal.Repos.Interfaces.Base;
public interface IBaseRepo<T> : IBaseViewRepo<T> where T : BaseEntity, new()
{
    T Find(int? id);
    T FindAsNoTracking(int id);
    T FindIgnoreQueryFilters(int id);
    void ExecuteParameterizedQuery(string sql, object[] sqlParametersObjects);
    int Add(T entity, bool persist = true);
    int AddRange(IQueryable<T> entities, bool persist = true);
    int Update(T entity, bool persist = true);
    int UpdateRange(IQueryable<T> entities, bool persist = true);
    int Delete(int id, long timeStamp, bool persist = true);
    int Delete(T entity, bool persist = true);
    int DeleteRange(IQueryable<T> entities, bool persist = true);
    int ExecuteBulkUpdate(Expression<Func<T, bool>> whereClause,
        Action<UpdateSettersBuilder<T>> setPropertyCalls);
    int ExecuteBulkDelete(Expression<Func<T, bool>> whereClause);
    int SaveChanges();
}
```

- Add the following global using statements to the `GlobalUsings.cs` file:

```
global using AutoLot.Dal.Repos;
global using AutoLot.Dal.Repos.Interfaces;
global using AutoLot.Dal.Repos.Interfaces.Base;
```

Step 2: Create the BaseView Repository

- Add a new class to the `Repos/Base` folder named `BaseViewRepo.cs`, and update the initial code to the following:

```
namespace AutoLot.Dal.Repos.Base;
public abstract class BaseViewRepo<T> : IBaseViewRepo<T> where T : class, new()
{
    private readonly bool _disposeContext;
    public DbSet<T> Table { get; }
    public ApplicationDbContext Context { get; }
    private bool _disposed;
}
```

- Add two constructors as follows.

```
protected BaseViewRepo(ApplicationDbContext context)
=> (Context, Table, _disposeContext) = (context, context.Set<T>(), false);

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

- Implement the `Dispose` pattern:

```
protected virtual void Dispose(bool disposing)
{
    if (disposing && _disposeContext)
    {
        Context.Dispose();
    }
}
public virtual void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}
```

- Implement the method that executes a raw SQL query using `FromSqlRaw()` to return a list of the entities:

```
public IQueryable<T> ExecuteSqlString(string sql) => Table.FromSqlRaw(sql);
```

- Implement the three `GetAll()` variations:

```
public virtual IQueryable<T> GetAll() => Table;
public virtual IQueryable<T> GetAllIgnoreQueryFilters() => Table.IgnoreQueryFilters();
public virtual IQueryable<T> GetAllIgnoreQueryFilters(string[] filtersToIgnore)
=> Table.IgnoreQueryFilters(filtersToIgnore);
```

- Add the following global using statement to the `GlobalUsings.cs` file:

```
global using AutoLot.Dal.Repos.Base;
```

Step 3: Create the Base Repository

- Add a new class to the `Repos/Base` folder named `BaseRepo.cs` and update the code to the following:

```
namespace AutoLot.Dal.Repos.Base;
public abstract class BaseRepo<T> : BaseViewRepo<T>, IBaseRepo<T> where T : BaseEntity, new()
{
    protected BaseRepo(ApplicationDbContext context) : base(context) { }
    protected BaseRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
    //rest of the code goes here
}
```

- The `BaseRepo SaveChanges()` method shells out to the `ApplicationDbContext SaveChanges()` method:

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

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

- The next method executes a parameterized query:

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

- The `Add()/AddRange()`, `Update()/UpdateRange()`, and `Delete()/DeleteRange()` 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(IQueryable<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(IQueryable<T> entities, bool persist = true)
{
    Table.UpdateRange(entities);
    return persist ? SaveChanges() : 0;
}
//This delete shows using entity state to eliminate a query
public int Delete(int id, long timeStamp, bool persist = true)
{
    var entity = new T {Id = id, TimeStamp = timeStamp};
    Context.Entry(entity).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(IQueryable<T> entities, bool persist = true)
{
    Table.RemoveRange(entities);
    return persist ? SaveChanges() : 0;
}

```

- Implement the `ExecuteBulkUpdate()` and `ExecuteBulkDelete()` methods:

```

public int ExecuteBulkUpdate(Expression<Func<T, bool>> whereClause,
    Action<UpdateSettersBuilder<T>> setPropertyCalls)
    => Table.IgnoreQueryFilters().Where(whereClause).ExecuteUpdate(setPropertyCalls);

public int ExecuteBulkDelete(Expression<Func<T, bool>> whereClause)
    => Table.IgnoreQueryFilters().Where(whereClause).ExecuteDelete();

```

Part 2: Add the Entity-Specific Repo Interfaces And Repos

Each model has an interface and a repo that uses the base repository for the shared functionality. Each specific repo extends or overwrites that base functionality as needed.

Copilot Agent Mode

Prompt: In the Repos/Interfaces folder, create an interface named for each of the entities (Car,CarDriver,Driver,Make,Radio) that inherit from IBaseRepo<T>. For the ICarRepo interface, add the following method signatures:

```
IQueryable<Car> GetAllBy(int makeId);
string GetPetName(int id);
int SetAllDrivableCarsColorAndMakeId(string color, int makeId);
int DeleteNonDrivableCars();
```

Prompt: In the Repos folder, create a class named CarDriverRepo that inherits from BaseRepo<CarDriver> and implements ICarDriverRepo. make all constructors public. Build a helper method named BuildBaseQuery that returns an IIcludableQueryable<CarDriver,Driver> that returns the CarDrivers table including the CarNavigation and DriverNavigationProperties. Use this helper method when implementing GetAll, GetAllIgnoreQueryFilters, and Find. The constructor that takes DbContextOptions should be internal.

Prompt: In the Repos folder, create a class named CarRepo that inherits from BaseRepo<Car> and implements ICarRepo. Build a helper method named BuildBaseQuery that returns an IOrededQueryable<Car> that includes MakeNavigation and is ordered by PetName. Implement the helper method when implementing all methods on the interface. Find should also call IngoreQueryFilters. SetAllDrivableCarsColorAndMakeId should use ExecuteBulkUpdate. DeleteNonDrivableCars should use ExecuteBulkDelete. GetPetName should use the GetPetName stored procedure and call into ExecuteParameterizedQuery using an in and an out parameter. The constructor that takes DbContextOptions should be internal.

Prompt: In the Repos folder, create a class named DriverRepo that inherits from BaseRepo<Driver> and implements IDriverRepo. make all constructors public. Build a helper method named BuildBaseQuery that returns an IOrededQueryable<Driver> that returns the Drivers table ordered by PersonInformation.LastName then by PersonInformation.FirstName. Use this helper method when implementing GetAll and GetAllIgnoreQueryFilters. The constructor that takes DbContextOptions should be internal.

Prompt: In the Repos folder, create a class named MakeRepo that inherits from BaseRepo<Make> and implements IMakeRepo. make all constructors public. Build a helper method named BuildBaseQuery that returns an IOrededQueryable<Make> that returns the Makes table ordered by Name. Use this helper method when implementing GetAll and GetAllIgnoreQueryFilters. The constructor that takes DbContextOptions should be internal.

Prompt: In the Repos folder, create a class named RadioRepo that inherits from BaseRepo<Radio> and implements IRadioRepo. The constructor that takes DbContextOptions should be internal.

Manual

Step 1: Create the Interface Files

- Create the following files in the `Repos\Interfaces` folder:

```
//ICarDriverRepo.cs
namespace AutoLot.Dal.Repos.Interfaces;
public interface ICarDriverRepo : IBaseRepo<CarDriver> { }

//ICarRepo.cs
namespace AutoLot.Dal.Repos.Interfaces;
public interface ICarRepo : IBaseRepo<Car>
{
    IQueryable<Car> GetAllBy(int makeId);
    string GetPetName(int id);
    int SetAllDrivableCarsColorAndMakeId(string color, int makeId);
    int DeleteNonDrivableCars();
}

//IDriverRepo.cs
namespace AutoLot.Dal.Repos.Interfaces;
public interface IDriverRepo : IBaseRepo<Driver> { }

//IMakeRepo.cs
namespace AutoLot.Dal.Repos.Interfaces;
public interface IMakeRepo : IBaseRepo<Make> { }

//IRadioRepo.cs
namespace AutoLot.Dal.Repos.Interfaces;
public interface IRadioRepo : IBaseRepo<Radio> { }
```

Step 2: Create the CarDriverRepo Class

- In the `Repos` folder, create a new class named `CarDriverRepo.cs` and update the code to the following:

```
namespace AutoLot.Dal.Repos;
public class CarDriverRepo : BaseRepo<CarDriver>, ICarDriverRepo
{
    public CarDriverRepo(ApplicationDbContext context) : base(context) { }
    internal CarDriverRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
    //rest of the code goes here
}
```

- Add a helper method to build a base query that includes the `Car` and `Driver` entities:

```
internal IIIncludableQueryable<CarDriver, Driver> BuildBaseQuery()
=> Table.Include(cd => cd.CarNavigation).Include(cd => cd.DriverNavigation);
```

- Override the GetAll methods and Find() method:

```
public override IQueryable<CarDriver> GetAll() => BuildBaseQuery();
public override IQueryable<CarDriver> GetAllIgnoreQueryFilters()
  => BuildBaseQuery().IgnoreQueryFilters();
public override IQueryable<CarDriver> GetAllIgnoreQueryFilters(string[] filtersToIgnore)
  => BuildBaseQuery().IgnoreQueryFilters(filtersToIgnore);
public override CarDriver Find(int? id)
  => id == null ? null : BuildBaseQuery().IgnoreQueryFilters().FirstOrDefault(cd => cd.Id == id);
```

Step 3: Create the CarRepo Class

- Create a new class named CarRepo.cs in the Repos directory and 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) { }
  //rest of the code goes here
}
```

- Add a helper method to build a base query that includes the Make entity:

```
internal IOOrderedQueryable<Car> BuildBaseQuery()
  => Table.Include(x => x.MakeNavigation).OrderBy(p => p.PetName);
```

- Add overrides for the GetAll methods:

```
public override IQueryable<Car> GetAll() => BuildBaseQuery();
public override IQueryable<Car> GetAllIgnoreQueryFilters()
  => BuildBaseQuery().IgnoreQueryFilters();
public override IQueryable<Car> GetAllIgnoreQueryFilters(string[] filtersToIgnore)
  => BuildBaseQuery().IgnoreQueryFilters(filtersToIgnore);
```

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

```
public override Car Find(int? id)
  => id == null ? null : BuildBaseQuery().IgnoreQueryFilters().FirstOrDefault(c => c.Id == id);
```

- Add the method to get all by Make ID:

```
public IQueryable<Car> GetAllBy(int makeId)
  => BuildBaseQuery().Where(x => x.MakeId == makeId);
```

- Add the method to update all drivable cars:

```
public int SetAllDrivableCarsColorAndMakeId(string color, int makeId)
  => ExecuteBulkUpdate(x => x.IsDrivable,
    c => c SetProperty(x => x.Color, color). SetProperty(x => x.MakeId, makeId));
```

- Optionally, you can write the update like this:

```
public int SetAllDrivableCarsColorAndMakeId(string color, int makeId)
  => ExecuteBulkUpdate(c => c.IsDrivable,
    u => u SetProperty(c => c.Color, color). SetProperty(c => c.MakeId, makeId));
```

- Add the method to delete all non-drivable cars:

```
public int DeleteNonDrivableCars() => ExecuteBulkDelete(x => !x.IsDrivable);
```

- Add method to get the PetName using the GetPetName sproc:

```
public string GetPetName(int id)
{
    var outParam = new SqlParameter("@petName", SqlDbType.NVarChar, 50)
    { Direction = ParameterDirection.Output };
    ExecuteParameterizedQuery("EXEC GetPetName @id, @petName OUT",
        [new SqlParameter("@id", id), outParam]);
    return outParam.Value?.ToString();
}
```

Step 4: Create the DriverRepo Class

- Create a new class named `DriverRepo.cs` in the `Repos` directory and make the class `public`, inherit `BaseRepo<Driver>` and implement `IDriverRepo`. Add the two required constructors as follows:

```
namespace AutoLot.Dal.Repos;
public class DriverRepo : BaseRepo<Driver>, IDriverRepo
{
    public DriverRepo(ApplicationDbContext context) : base(context) { }
    internal DriverRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

- Add a helper method to build a base query that orders by `Lastname` then `Firstname`:

```
internal IOrderedQueryable<Driver> BuildBaseQuery()
=> Table.OrderBy(m => m.PersonInformation.LastName).ThenBy(f => f.PersonInformation.FirstName);
```

- Override the `GetAll()` methods to use the base query builder:

```
public override IQueryable<Driver> GetAll() => BuildBaseQuery();
public override IQueryable<Driver> GetAllIgnoreQueryFilters()
=> BuildBaseQuery().IgnoreQueryFilters();
public override IQueryable<Driver> GetAllIgnoreQueryFilters(string[] filtersToIgnore)
=> BuildBaseQuery().IgnoreQueryFilters(filtersToIgnore);
```

Step 5: Create the MakeRepo Class

- Create a new class named `MakeRepo.cs` in the `Repos` directory and make the class `public`, inherit `BaseRepo<Make>`, and implement `IMakeRepo`. Add the required constructors 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) { }
}
```

- Add a helper method to build a base query that orders by `Name`:

```
internal IOrderedQueryable<Make> BuildBaseQuery() => Table.OrderBy(m => m.Name);
```

- Override the GetAll() methods to use the base query builder:

```
public override IQueryable<Make> GetAll() => BuildBaseQuery();
public override IQueryable<Make> GetAllIgnoreQueryFilters()
    => BuildBaseQuery().IgnoreQueryFilters();
public override IQueryable<Make> GetAllIgnoreQueryFilters(string[] filtersToIgnore)
    => BuildBaseQuery().IgnoreQueryFilters(filtersToIgnore);
```

Step 6: Create the RadioRepo Class

- Create a new class named RadioRepo.cs in the Repos directory and make the class public, inherit BaseRepo<Radio>, and implement IRadioRepo. Add the standard constructors, as shown below.

```
namespace AutoLot.Dal.Repos;
public class RadioRepo : BaseRepo<Radio>, IRadioRepo
{
    public RadioRepo(ApplicationDbContext context) : base(context) { }
    internal RadioRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

Summary

The lab created the repositories and their interfaces.

Next steps

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