.NET App Dev Hands-On Workshop

Lab 5 - Repositories

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 Repositories

Step 1: Create the Base Repository Interfaces

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 in this workshop.

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

```
namespace AutoLot.Dal.Repos.Base;

public interface IBaseViewRepo<T> : IDisposable where T : class, new()
{
  ApplicationDbContext Context { get; }
  IEnumerable<T> ExecuteSqlString(string sql);
  IEnumerable<T> GetAll();
  IEnumerable<T> GetAllIgnoreQueryFilters();
}
```

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

```
namespace AutoLot.Dal.Repos.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(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);
    int SaveChanges();
}
```

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

```
namespace AutoLot.Dal.Repos.Base;
```

```
public interface ITemporalTableBaseRepo<T> : IBaseRepo<T> where T : BaseEntity, new()
{
  IEnumerable<TemporalViewModel<T>> GetAllHistory();
  IEnumerable<TemporalViewModel<T>> GetHistoryAsOf(DateTime dateTime);
  IEnumerable<TemporalViewModel<T>> GetHistoryBetween(
    DateTime startDateTime, DateTime endDateTime);
  IEnumerable<TemporalViewModel<T>> GetHistoryContainedIn(
    DateTime startDateTime, DateTime endDateTime);
  IEnumerable<TemporalViewModel<T>> GetHistoryFromTo(
    DateTime startDateTime, DateTime endDateTime);
}
```

Step 2: Create the BaseView Repository

- Add a new class to the `Repos/Base` folder named `BaseViewRepo.cs`, make the class `public` and `abstract`, generic, and implement `IBaseViewRepo<T>`:

```
namespace AutoLot.Dal.Repos.Base;

public abstract class BaseViewRepo<T> : IBaseViewRepo<T> where T : class, new()
{
//implementation goes here
}
```

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


- Implement the `Dispose` pattern:

```
public virtual 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;
}

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

- Implement the two `GetAll()` variations:

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

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

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

Step 3: Create the Base Repository

- Add a new class to the `Repos/Base` folder named `BaseRepo.cs`, make the class `public` and `abstract`, generic, inherit from `BaseViewRepo<T>` and implement `IBaseRepo<T>`:

```
namespace AutoLot.Dal.Repos.Base;

public abstract class BaseRepo<T> : BaseViewRepo<T>,IBaseRepo<T> where T : BaseEntity, new()
{
//implementation goes here
}
```

- Add the two constructors supported by the BaseViewRepo:

```
protected BaseRepo(ApplicationDbContext context) : base(context) { }
protected BaseRepo(DbContextOptions<ApplicationDbContext> options)
  : this(new ApplicationDbContext(options)) { }
```

- 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 virtual T FindIgnoreQueryFilters(int id)
  => Table.IgnoreQueryFilters().FirstOrDefault(x => x.Id == id);
```

- The next method executes a parameterized query:

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

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

```
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);
  }
}
```

- 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(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;
}
```

- The final `Delete()` method uses `EntityState` to perform the delete operation:

```
public int Delete(int id, byte[] timeStamp, bool persist = true)
{
  var entity = new T {Id = id, TimeStamp = timeStamp};
  Context.Entry(entity).State = EntityState.Deleted;
  return persist ? SaveChanges() : 0;
}
```

Step 4: Create the Temporal Table Base Repository

- Add a new class to the `Repos/Base` folder named `TemporalTableBaseRepo.cs`, make the class `public` and `abstract`, generic, inherit from `BaseRepo<T>` and implement `ITemporalTableBaseRepo<T>`:

```
namespace AutoLot.Dal.Repos.Base;

public abstract class TemporalTableBaseRepo<T>
  : BaseRepo<T>, ITemporalTableBaseRepo<T> where T : BaseEntity, new()
{
//implementation goes here
}
```

- Add the two constructors supported by the `BaseViewRepo`:

```
protected TemporalTableBaseRepo(ApplicationDbContext context) : base(context) {}
protected TemporalTableBaseRepo(DbContextOptions<ApplicationDbContext> options)
  : base(options) { }
```

- Add an internal helper to convert the current time to UTC:

```
internal static DateTime ConvertToUtc(DateTime dateTime)
  => TimeZoneInfo.ConvertTimeToUtc(dateTime, TimeZoneInfo.Local);
```

- Add an internal helper to execute one of the temporal queries:

```
internal static IEnumerable<TemporalViewModel<T>> ExecuteQuery(IQueryable<T> query)
  => query.OrderBy(e => EF.Property<DateTime>(e, "ValidFrom"))
          .Select(e => new TemporalViewModel<T>
          {
            Entity = e,
            ValidFrom = EF.Property<DateTime>(e, "ValidFrom"),
            ValidTo = EF.Property<DateTime>(e, "ValidTo")
          });
```

- The public methods execute the five temporal queries:

```
public IEnumerable<TemporalViewModel<T>> GetAllHistory()
  => ExecuteQuery(Table.TemporalAll());

public IEnumerable<TemporalViewModel<T>> GetHistoryAsOf(DateTime dateTime)
  => ExecuteQuery(Table.TemporalAsOf(ConvertToUtc(dateTime)));

public IEnumerable<TemporalViewModel<T>> GetHistoryBetween(
    DateTime startDateTime, DateTime endDateTime)
  => ExecuteQuery(Table.TemporalBetween(ConvertToUtc(startDateTime), ConvertToUtc(endDateTime)));

public IEnumerable<TemporalViewModel<T>> GetHistoryContainedIn(
    DateTime startDateTime, DateTime endDateTime)
  => ExecuteQuery(Table.TemporalContainedIn(ConvertToUtc(startDateTime),
ConvertToUtc(endDateTime)));

public IEnumerable<TemporalViewModel<T>> GetHistoryFromTo(
    DateTime startDateTime, DateTime endDateTime)
  => ExecuteQuery(Table.TemporalFromTo(ConvertToUtc(startDateTime), ConvertToUtc(endDateTime)));
```

Step 5: Update the GlobalUsings.cs

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

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

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:

```
ICarDriverRepo.cs
ICarRepo.cs
ICreditRiskRepo.cs
ICustomerOrderViewModelRepo.cs
ICustomerRepo.cs
IDriverRepo.cs
IMakeRepo.cs
IOrderRepo.cs
IRadioRepo.cs
```

Step 2: Define the ICarDriverRepo Interface

- Update the code for the interface to the following:

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

Step 3: Define the ICarRepo Interface

- Update the code for the interface to the following:

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

Step 4: Define the ICreditRiskRepo Interface

- Update the code for the interface to the following:

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

Step 5: Define the ICustomerOrderViewModelRepo Interface

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces;
public interface ICustomerOrderViewModelRepo : IBaseViewRepo<CustomerOrderViewModel> { }
```

Step 6: Define the ICustomerRepo Interface

- Update the code for the interface to the following:

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

Step 7: Define the IDriverRepo Interface

- Update the code for the interface to the following:

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

Step 8: Define the IMakeRepo Interface

- Update the code for the interface to the following:

```
namespace AutoLot.Dal.Repos.Interfaces;
public interface IMakeRepo : ITemporalTableBaseRepo<Make> { }
```

Step 9: Define the IOrderRepo Interface

- Update the code for the interface to the following:

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

Step 10: Define the IRadioRepo Interface

- Update the code for the interface to the following:

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

Step 11: Update the GlobalUsings.cs

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

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

Part 3: Implement the Entity Specific Repos

Step 1: Create the Class Files

- Create the following files in the Repos folder:

```
CarDriverRepo.cs
CarRepo.cs
CreditRiskRepo.cs
CustomerOrderViewModelRepo.cs
CustomerRepo.cs
DriverRepo.cs
MakeRepo.cs
OrderRepo.cs
RadioRepo.cs
```

Step 2: Implement the CarDriverRepo

- Make the class `public`, inherit `TemporalTableBaseRepo<CarDriver>` and implement `ICarDriverRepo`. Add the two required constructors as follows:

```
namespace AutoLot.Dal.Repos;

public class CarDriverRepo : TemporalTableBaseRepo<CarDriver>, ICarDriverRepo
{
  public CarDriverRepo(ApplicationDbContext context) : base(context) { }
  internal CarDriverRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

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

```
internal IIncludableQueryable<CarDriver, Driver> BuildBaseQuery()
  => Table.Include(c => c.CarNavigation).Include(d => d.DriverNavigation);
```

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

```
public override IEnumerable<CarDriver> GetAll()=> BuildBaseQuery();
public override IEnumerable<CarDriver> GetAllIgnoreQueryFilters()
  => BuildBaseQuery().IgnoreQueryFilters();
```

- Override the Find() method to use the bae query builder:

```
public override CarDriver Find(int? id)
  => BuildBaseQuery().IgnoreQueryFilters().FirstOrDefault(x => x.Id == id);
```

Step 3: Implement the CarRepo

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

```
namespace AutoLot.Dal.Repos;

public class CarRepo : TemporalTableBaseRepo<Car>, ICarRepo
{
  public CarRepo(ApplicationDbContext context) : base(context) { }
  internal CarRepo(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

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

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

- Add overrides for the `GetAll` methods:

```
public override IEnumerable<Car> GetAll() => BuildBaseQuery();
public override IEnumerable<Car> GetAllIgnoreQueryFilters()
  => BuildBaseQuery().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)
  => BuildBaseQuery().Where(x => x.MakeId == makeId);
```


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

```
public string GetPetName(int id)
{
  var parameterId = new SqlParameter
  {
    ParameterName = "@carId",
    SqlDbType = SqlDbType.Int,
    Value = id,
  };
  var parameterName = new SqlParameter
  {
    ParameterName = "@petName",
    SqlDbType = SqlDbType.NVarChar,
    Size = 50,
    Direction = ParameterDirection.Output
  };
  ExecuteParameterizedQuery("EXEC [dbo].[GetPetName] @carId, @petName OUTPUT",
    new[] { parameterId, parameterName });
  return (string)parameterName.Value;
}
```

Step 4: Implement the CreditRiskRepo

- 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 5: Implement the CustomerOrderViewModelRepo

- Make the class `public`, inherit `BaseViewRepo<CustomerOrderViewModel>`, and implement `ICustomerOrderViewModelRepo`. Add the two required constructors as follows:

```
namespace AutoLot.Dal.Repos;
public class CustomerOrderViewModelRepo
 : BaseViewRepo<CustomerOrderViewModel>, ICustomerOrderViewModelRepo
{
  public CustomerOrderViewModelRepo(ApplicationDbContext context) : base(context) { }
  internal CustomerOrderViewModelRepo(DbContextOptions<ApplicationDbContext> options) :
    base(options) { }
}
```

Step 6: Implement the CustomerRepo

- Make the class public, inherit `BaseRepo<Customer>`, and implement `ICustomerRepo`. Add the two required constructors 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) { }
}
```

- Override the `GetAll()` to include `Orders` and sort by `LastName`:

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

Step 7: Implement the DriverRepo

- 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> BuildQuery()
  => Table.OrderBy(m => m.PersonInformation.LastName).ThenBy(f => f.PersonInformation.FirstName);
```

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

```
public override IEnumerable<Driver> GetAll() => BuildQuery();
public override IEnumerable<Driver> GetAllIgnoreQueryFilters()
  => BuildQuery().IgnoreQueryFilters();
```

Step 8: Implement the MakeRepo

- Make the class public, inherit `TemporalTableBaseRepo<Make>`, and implement `IMakeRepo`. Add the required constructors as follows:

```
namespace AutoLot.Dal.Repos;
public class MakeRepo : TemporalTableBaseRepo<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> BuildQuery() => Table.OrderBy(m => m.Name);
```

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

```
public override IEnumerable<Make> GetAll() => BuildQuery();
public override IEnumerable<Make> GetAllIgnoreQueryFilters()
  => BuildQuery().IgnoreQueryFilters();
```

Step 9: Implement the OrderRepo

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

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

Step 10: Implement the RadioRepo

- Make the class public, inherit `TemporalTableBaseRepo<Radio>`, and implement `IRadioRepo`. Add the standard constructors, as shown below.

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

