Build an EF and ASP.NET Core 3.0 App HOL

Lab 6

This lab walks you through integration testing for the Data Access Layer. Prior to starting this lab, you must have completed Lab 5.

Part 1: Giving the Test Project Access to Dal Internals

Many of the methods needed to testing are set with internal access control. These can be made available to the test project using an assembly level attribute.

1) Create a new file named LibraryAttributes.cs in the root of the AutoLot.Dal project. Add the following using statement to the top of the file:

```
using System.Runtime.CompilerServices;
```

2) Add the following attribute into the file. Note: If you changed the name of the test project, make sure to use the full name in the attribute:

```
[assembly:InternalsVisibleTo("AutoLot.Dal.Tests")]
```

Part 2: Create the Testing Framework

Step 1: Use the Configuration System

The test project will leverage the .NET Core configuration system for connection string configuration.

1) Create a new class named TestHelpers in the root of the AutoLot.Dal.Tests project. Make the class public and static, and add the following using statements to the top of the file:

```
using System.IO;
using AutoLot.Dal.EfStructures;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Configuration;
```

2) Add a method that will use the ConfigurationBuilder to load an appsettings.json file and then return an instance of IConfiguration:

```
public static IConfiguration GetConfiguration() =>
  new ConfigurationBuilder()
        .SetBasePath(Directory.GetCurrentDirectory())
        .AddJsonFile("appsettings.json", true, true)
        .Build();
```

3) Add a method that will use the IConfiguration to get the connection string from the settings file and then create an instance of the ApplicationDbContext using DbContextOptionsBuilder:

```
public static ApplicationDbContext GetContext(IConfiguration configuration)
{
  var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
  var connectionString = configuration.GetConnectionString("AutoLot");
  optionsBuilder.UseSqlServer(connectionString, sqlOptions => sqlOptions.EnableRetryOnFailure());
  return new ApplicationDbContext(optionsBuilder.Options);
}
```

4) Add a method that will create a new instance of ApplicationDbContext sharing the same connection as another ApplicationDbContext. This allows for sharing transactions between context instances:

```
public static ApplicationDbContext GetSecondContext(
  ApplicationDbContext oldContext, IDbContextTransaction trans)
{
  var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
  optionsBuilder.UseSqlServer(
    oldContext.Database.GetDbConnection(),
    sqlServerOptions => sqlServerOptions.EnableRetryOnFailure());
  var context = new ApplicationDbContext(optionsBuilder.Options);
  context.Database.UseTransaction(trans.GetDbTransaction());
  return context;
}
```

5) Add a new JSON file named appsettings.json into the root of the AutoLot.Dal.Tests project. Update the JSON to match this (update your connection string appropriately):

```
{
  "ConnectionStrings": {
    "AutoLot": "server=.,5433;Database=AutoLotWorkshop;User Id=sa;Password=P@ssw0rd;"
  }
}
```

6) Update the appsettings.json file to copy to the output directory on build. This can be done in the project file by adding this:

```
<ItemGroup>
  <None Update="appsettings.json">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```

Step 2: Create a Base Test Class

1) Create a new folder named Base in the AutoLot.Dal.Tests project. In this folder, add a new class named BaseTest.cs. Add the following using statements to the top of the file:

```
using System;
using System.Data;
using AutoLot.Dal.EfStructures;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.Extensions.Configuration;
```

2) Make the class public and abstract and implement IDisposable:

```
public abstract class BaseTest : IDisposable
{
  protected BaseTest()
  {
  }

  public void Dispose()
  {
    Context.Dispose();
  }
}
```

3) In the constructor, use the TestHelpers methods to get class level instances of IConfiguration and ApplicationDbContext:

```
protected readonly IConfiguration Configuration;
protected readonly ApplicationDbContext Context;
protected BaseTest()
{
  Configuration = TestHelpers.GetConfiguration();
  Context = TestHelpers.GetContext(Configuration);
}
```

4) In the Dispose method, dispose of the context instance:

```
public void Dispose()
{
  Context.Dispose();
}
```

5) Create a method to execute an Action in a transaction:

```
protected void ExecuteInATransaction(Action actionToExecute)
{
  var strategy = Context.Database.CreateExecutionStrategy();
  strategy.Execute(() =>
  {
    using var trans = Context.Database.BeginTransaction();
    actionToExecute();
    trans.Rollback();
  });
}
```

6) Create a method to execute an Action in a shared transaction:

```
protected void ExecuteInASharedTransaction(Action<IDbContextTransaction> actionToExecute)
{
  var strategy = Context.Database.CreateExecutionStrategy();
  strategy.Execute(() =>
  {
    using IDbContextTransaction trans =
      Context.Database.BeginTransaction(IsolationLevel.ReadUncommitted);
    actionToExecute(trans);
    trans.Rollback();
  });
}
```

Step 3: Create the EnsureAutoLotDatabaseTestFixture

Test Fixtures allow for running fixture setup and tear down methods.

1) In the Base folder add a new class named EnsureAutoLotDatabaseTestFixture.cs. Add the following using statements to the top of the file:

```
using System;
using AutoLot.Dal.Initialization;
```

2) Make the class public and sealed and implement IDisposable. The constructor will create and seed the database. The Dispose method does nothing:

```
public sealed class EnsureAutoLotDatabaseTestFixture : IDisposable
{
  public EnsureAutoLotDatabaseTestFixture()
  {
    var configuration =  TestHelpers.GetConfiguration();
    var context = TestHelpers.GetContext(configuration);
    SampleDataInitializer.ClearAndReseedDatabase(context);
    context.Dispose();
  }

  public void Dispose()
  {
  }
}
```

Part 3: Creating the Tests

Step 1: Add the Initializer Tests

1) Create a new folder named Initialization in the AutoLot.Dal.Tests project. In this folder, add a new class named InitializerTests.cs. Add the following using statements to the top of the file:

```
using System.Linq;
using AutoLot.Dal.Initialization;
using AutoLot.Dal.Tests.Base;
using Microsoft.EntityFrameworkCore;
using Xunit;
```

2) Make the class public and derive from BaseTest. Add the Collection attribute to the class:

```
[Collection("Integration Tests")]
public class IntializerTests : BaseTest
{
}
```

3) Add the following tests into the class:

```
[Fact]
public void ShouldDropAndCreateTheDatabase()
{
  SampleDataInitializer.DropAndCreateDatabase(Context);
  var cars = Context.Cars.IgnoreQueryFilters();
  Assert.Empty(cars);
}

[Fact]
public void ShouldDropAndRecreateTheDatabaseThenLoadData()
{
  SampleDataInitializer.InitializeData(Context);
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.Equal(9, cars.Count);
}

[Fact]
public void ShouldClearAndReseedTheDatabase()
{
  SampleDataInitializer.InitializeData(Context);
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.NotNull(cars);
  Assert.Equal(9, cars.Count);
}
[Fact]
public void ShouldClearTheData()
{
  SampleDataInitializer.InitializeData(Context);
  SampleDataInitializer.ClearData(Context);
  var cars = Context.Cars.IgnoreQueryFilters();
  Assert.NotNull(cars);
  Assert.Empty(cars);
}
[Fact]
public void ShouldReseedTheTables()
{
  SampleDataInitializer.ClearAndReseedDatabase(Context);
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.NotNull(cars);
  Assert.Equal(9, cars.Count);
}
```

Step 2: Add the Car Tests

1) Create a new folder named ContextTests in the AutoLot.Dal.Test project. In this folder, add a new class named CarTests.cs. Add the following using statements to the top of the file:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Models.Entities;
using AutoLot.Dal.Tests.Base;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Microsoft.EntityFrameworkCore.Storage;
using Xunit;
```

2) Make the class public, derive from BaseTest, implement IClassFixture<EnsureAutoLotDatabaseTestFixture>, and add the Collection attribute:

```
[Collection("Integration Tests")]
public class CarTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
{
}
```

3) Add the following tests to the class:

```
[Fact]
public void ShouldReturnNoCarsWithQueryFilterNotSet()
{
  var cars = Context.Cars.ToList();
  Assert.Empty(cars);
}

[Fact]
public void ShouldGetAllOfTheCars()
{
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.Equal(9, cars.Count);
}

[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetTheCarsByMake(int makeId, int expectedCount)
{
  Context.MakeId = makeId;
  var cars = Context.Cars.ToList();
  Assert.Equal(expectedCount, cars.Count);
}
[Fact]
public void ShouldNotGetTheCarsUsingFromSql()
{
  var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
  var tableName = entity.GetTableName();
  var schemaName = entity.GetSchema();
  var cars = Context.Cars.FromSqlRaw($"Select * from {schemaName}.{tableName}").ToList();
  Assert.Empty(cars);
}
[Fact]
public void ShouldGetTheCarsUsingFromSqlWithIgnoreQueryFilters()
{
  var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
  var tableName = entity.GetTableName();
  var schemaName = entity.GetSchema();
  var cars = Context.Cars.FromSqlRaw(
    $"Select * from {schemaName}.{tableName}").IgnoreQueryFilters().ToList();
  Assert.Equal(9, cars.Count);
}
```

```
[Fact]
public void ShouldGetOneCarUsingInterpolation()
{
  var carId = 1;
  var car = Context.Cars
      .FromSqlInterpolated($"Select * from dbo.Inventory where Id = {carId}")
      .Include(x => x.MakeNavigation)
      .IgnoreQueryFilters().First();
  Assert.Equal("Black", car.Color);
  Assert.Equal("VW", car.MakeNavigation.Name);
}

[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetTheCarsByMakeUsingFromSql(int makeId, int expectedCount)
{
  Context.MakeId = makeId;
  var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
  var tableName = entity.GetTableName();
  var schemaName = entity.GetSchema();
  var cars = Context.Cars.FromSqlRaw($"Select * from {schemaName}.{tableName}").ToList();
  Assert.Equal(expectedCount, cars.Count);
}

[Fact]
public void ShouldGetAllOfTheCarsWithMakes()
{
  var cars = Context.Cars.IgnoreQueryFilters().Include(c => c.MakeNavigation).ToList();
  Assert.Equal(9, cars.Count);
}

[Fact]
public void ShouldGetCarsOnOrderWithRelatedProperties()
{
  var cars =
    Context.Cars.IgnoreQueryFilters()
           .Where(c => c.Orders.Any())
           .Include(c => c.MakeNavigation)
           .Include(c => c.Orders).ThenInclude(o => o.CustomerNavigation).ToList();
  Assert.Equal(4, cars.Count);
  cars.ForEach(c =>
  {
    Assert.NotNull(c.MakeNavigation);
    Assert.NotNull(c.Orders.ToList()[0].CustomerNavigation);
  });
}
```

```
[Fact]
public void ShouldGetRelatedInformationExplicitly()
{
  var car = Context.Cars.IgnoreQueryFilters().First(x => x.Id == 1);
  Assert.Null(car.MakeNavigation);
  Context.Entry(car).Reference(c => c.MakeNavigation).Load();
  Assert.NotNull(car.MakeNavigation);

  Assert.Empty(car.Orders);
  Context.Entry(car).Collection(c => c.Orders).Load();
  Assert.Single(car.Orders);
}

[Fact]
public void ShouldAddACar()
{
  ExecuteInATransaction(RunTheTest);

  void RunTheTest()
  {
    var car = new Car
    {
      Color = "Yellow",
      MakeId = 1,
      PetName = "Herbie"
    };
    var carCount = Context.Cars.IgnoreQueryFilters().Count();
    Context.Cars.Add(car);
    Context.SaveChanges();
    var newCarCount = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(carCount + 1, newCarCount);
  }
}
```

```
[Fact]
public void ShouldAddMultipleCars()
{
  ExecuteInATransaction(RunTheTest);

  void RunTheTest()
  {
    var cars = new List<Car>
    {
      new Car
      {
        Color = "Yellow",
        MakeId = 1,
        PetName = "Herbie"
      },
      new Car
      {
        Color = "White",
        MakeId = 2,
        PetName = "Mach 5"
      },
    };
    var carCount = Context.Cars.IgnoreQueryFilters().Count();
    Context.Cars.AddRange(cars);
    Context.SaveChanges();
    var newCarCount = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(carCount + 2, newCarCount);
  }
}

[Fact]
public void ShouldAddAnObjectGraph()
{
  ExecuteInATransaction(RunTheTest);

  void RunTheTest()
  {
    var make = new Make { Name = "Honda" };
    var car = new Car
    {
      Color = "Yellow",
      MakeId = 1,
      PetName = "Herbie"
    };
    ((List<Car>)make.Cars).Add(car);
    Context.Makes.Add(make);
    var carCount = Context.Cars.IgnoreQueryFilters().Count();
    var makeCount = Context.Makes.IgnoreQueryFilters().Count();
    Context.SaveChanges();
    var newCarCount = Context.Cars.IgnoreQueryFilters().Count();
    var newMakeCount = Context.Makes.IgnoreQueryFilters().Count();
    Assert.Equal(carCount + 1, newCarCount);
    Assert.Equal(makeCount + 1, newMakeCount);
  }
}
```

```
[Fact]
public void ShouldUpdateACar()
{
  ExecuteInASharedTransaction(RunTheTest);

  void RunTheTest(IDbContextTransaction trans)
  {
    var car = Context.Cars.IgnoreQueryFilters().First(c => c.Id == 1);
    Assert.Equal("Black", car.Color);
    car.Color = "White";
    Context.SaveChanges();
    Assert.Equal("White", car.Color);
    var context2 = TestHelpers.GetSecondContext(Context, trans);
    var car2 = context2.Cars.IgnoreQueryFilters().First(c => c.Id == 1);
    Assert.Equal("White", car2.Color);
  }
}

 [Fact]
public void ShouldUpdateACarUsingState()
{
  ExecuteInASharedTransaction(RunTheTest);

  void RunTheTest(IDbContextTransaction trans)
  {
    var car = Context.Cars.IgnoreQueryFilters().AsNoTracking().First(c => c.Id == 1);
    Assert.Equal("Black", car.Color);
    var updatedCar = new Car
    {
      Color = "White", //Original is Black
      Id = car.Id,
      MakeId = car.MakeId,
      PetName = car.PetName,
      TimeStamp = car.TimeStamp
    };
    var context2 = TestHelpers.GetSecondContext(Context, trans);
    context2.Entry(updatedCar).State = EntityState.Modified;
    //context2.Cars.Update(updatedCar);
    context2.SaveChanges();
    var context3 = TestHelpers.GetSecondContext(Context, trans);
    var car2 = context3.Cars.IgnoreQueryFilters().First(c => c.Id == 1);
    Assert.Equal("White", car2.Color);
  }
}
```

```
[Fact]
public void ShouldThrowConcurrencyException()
{
  ExecuteInATransaction(RunTheTest);

  void RunTheTest()
  {
    var car = Context.Cars.IgnoreQueryFilters().First();
    //Update the database outside of the context
    Context.Database.ExecuteSqlInterpolated(
      $"Update dbo.Inventory set Color='Pink' where Id = {car.Id}");
    car.Color = "Yellow";
    var ex = Assert.Throws<CustomConcurrencyException>(() => Context.SaveChanges());
    var entry = ((DbUpdateConcurrencyException)ex.InnerException)?.Entries[0];
    PropertyValues originalProps = entry.OriginalValues;
    PropertyValues currentProps = entry.CurrentValues;
    //This needs another database call
    PropertyValues databaseProps = entry.GetDatabaseValues();
  }
}

[Fact]
public void ShouldRemoveACar()
{
  ExecuteInATransaction(RunTheTest);

  void RunTheTest()
  {
    var carCount = Context.Cars.IgnoreQueryFilters().Count();
    var car = Context.Cars.IgnoreQueryFilters().First(c => c.Id == 2);
    Context.Cars.Remove(car);
    Context.SaveChanges();
    var newCarCount = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(carCount - 1, newCarCount);
    Assert.Equal(EntityState.Detached,Context.Entry(car).State);
  }
}

[Fact]
public void ShouldFailToRemoveACar()
{
  ExecuteInATransaction(RunTheTest);

  void RunTheTest()
  {
    var car = Context.Cars.IgnoreQueryFilters().First(c => c.Id == 1);
    Context.Cars.Remove(car);
    Assert.Throws<CustomDbUpdateException>(()=>Context.SaveChanges());
  }
}
```

```
[Fact]
public void ShouldRemoveACarUsingState()
{
  ExecuteInASharedTransaction(RunTheTest);

  void RunTheTest(IDbContextTransaction trans)
  {
    var carCount = Context.Cars.IgnoreQueryFilters().Count();
    var car = Context.Cars.IgnoreQueryFilters().AsNoTracking().First(c => c.Id == 2);
    var context2 = TestHelpers.GetSecondContext(Context, trans);
    //context2.Entry(car).State = EntityState.Deleted;
    context2.Cars.Remove(car);
    context2.SaveChanges();
    var newCarCount = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(carCount - 1, newCarCount);
    Assert.Equal(EntityState.Detached, Context.Entry(car).State);
  }
}
```

Step 3: Add the Customer Tests

1) In the ContextTests folder, add a new class named CustomerTests.cs. Add the following using statements to the top of the file:

```
using System;
using System.Linq;
using System.Linq.Expressions;
using AutoLot.Models.Entities;
using AutoLot.Dal.Tests.Base;
using Xunit;
```

2) Make the class public, derive from BaseTest, implement IClassFixture<EnsureAutoLotDatabaseTestFixture>, and add the Collection attribute:

```
[Collection("Integration Tests")]
public class CustomerTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
{
}
```

3) Add the following tests to the class:

```
[Fact]
public void ShouldGetAllOfTheCustomers()
{
  var customers = Context.Customers.ToList();
  Assert.Equal(5, customers.Count);
}

[Fact]
public void FirstGetFirstMatchingRecord()
{
  //Gets the first record, database order
  var customer = Context.Customers.First();
  Assert.Equal(1,customer.Id);
}
```

```
[Fact]
public void FirstShouldThrowExceptionIfNoneMatch()
{
  //Filters based on Id. Throws due to no match
  Assert.Throws<InvalidOperationException>(()=> Context.Customers.First(x=>x.Id == 10));
}

[Fact]
public void FirstOrDefaultShouldReturnDefaultIfNoneMatch()
{
  //Expression<Func<Customer>> is a lambda expression
  Expression<Func<Customer, bool>> expression = x => x.Id == 10;
  //Returns null when nothing is found
  var customer = Context.Customers.FirstOrDefault(expression);
  Assert.Null(customer);
}

[Fact]
public void GetOneMatchingRecordWithSingle()
{
  //Gets the first record, database order
  var customer = Context.Customers.Single(x=>x.Id == 1);
  Assert.Equal(1,customer.Id);
}
[Fact]
public void SingleShouldThrowExceptionIfNoneMatch()
{
  //Filters based on Id. Throws due to no match
  Assert.Throws<InvalidOperationException>(()=> Context.Customers.Single(x=>x.Id == 10));
}
[Fact]
public void SingleShouldThrowExceptionIfMoreThenOneMatch()
{
  //Filters based on Id. Throws due to no match
  Assert.Throws<InvalidOperationException>(()=> Context.Customers.Single());
}
[Fact]
public void SingleOrDefaultShouldReturnDefaultIfNoneMatch()
{
  //Expression<Func<Customer>> is a lambda expression
  Expression<Func<Customer, bool>> expression = x=>x.Id == 10;
  //Returns null when nothing is found
  var customer = Context.Customers.SingleOrDefault(expression);
  Assert.Null(customer);
}

[Fact]
public void ShouldGetCustomersWithLastNameWithW()
{
    var customers =
        Context.Customers.Where(x => x.PersonalInformation.LastName.StartsWith("W")).ToList();
    Assert.Equal(2, customers.Count);
}
```

```
[Fact]
public void ShouldGetCustomersWithLastNameWithWAndFirstNameM()
{
  var customers =
    Context.Customers.Where(x => x.PersonalInformation.LastName.StartsWith("W"))
                     .Where(x=>x.PersonalInformation.FirstName.StartsWith("M")).ToList();
  Assert.Single(customers);
}
[Fact]
public void ShouldGetCustomersWithLastNameWithWOrH()
{
  var customers =
    Context.Customers.Where(x =>
      x.PersonalInformation.LastName.StartsWith("W") ||
      x.PersonalInformation.LastName.StartsWith("H")).ToList();
  Assert.Equal(3, customers.Count);
}

[Fact]
public void ShouldSortByFirstNameThenLastName()
{
  var customers = Context.Customers
      .OrderBy(x => x.PersonalInformation.LastName)
      .ThenBy(x => x.PersonalInformation.FirstName).ToList();
}
```

Summary

This lab created added in integration tests for the data access library.

Next steps

In the next part of this tutorial series, you will start working with ASP.Net Core.