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

Lab 7 (Optional)

This lab walks you through adding and running the tests for the data access layer. This Lab is an optional lab. Prior to starting this lab, you must have completed Lab 6. Start by deleting the `UnitTest1.cs` file from the `AutoLot.Dal.Tests` project.

Part 1: Adding Configuration Information and Base Test Framework

The `AutoLot.Dal.Tests` project uses .NET 5 configuration to set the connection string.

Step 1: Adding the configuration file

- Add a new JSON file named `appsettings.json` to the root of the project. Add the following content to the file (adjusting the connection string to your environment):

```
{
  "ConnectionStrings": {
    "AutoLot": "server=.,5433;Database=AutoLot50;User Id=sa;Password=P@ssw0rd;"
  }
}
```

- Set the file to be copied to the output directory when the project is built. Add the following to the project file:

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

Step 2: Create the TestHelpers static Class

The `TestHelpers` class will use the configuration system to get the connection string and create an instance of the `ApplicatonDbContext`.

- Add a new file named `TestHelpers.cs`. 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;
```

- Make the class public and static:

```
namespace AutoLot.Dal.Tests
{
  public static class TestHelpers
  {
  }
}
```

- Add a method to get an instance of `IConfiguration` using the `appsettings.json` file:

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

- Add a method to get an instance of `ApplicationDbContext` using the connection string:

```
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);
}
```

- Add a method to get a second `ApplicationDbContext` that shares a transaction with the first `ApplicationDbContext`:

```
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;
}
```

Step 3: Create the BaseTest Class

The BaseTest class will use the configuration system to get the connection string.

- Add a new folder named Base to the project. In that folder, add a new file 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;
```

- Make the file public and abstract and implement `IDisposable`:

```
namespace AutoLot.Dal.Tests.Base
{
  public abstract class BaseTest : IDisposable
  {
  }
}
```

- Add protected variables for `IConfiguration` and `ApplicationDbContext` and initialize them in the constructor, using the `TestHelpers` methods, disposing of the context in the `Dispose` method:

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

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

- Add transaction support for tests with a method to execute in a transaction as execute in a shared transaction (across ApplicationDbContexts):

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

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();
  });
}
```

Part 2: Testing the Initialization Code

- Create a new folder named `Initialization` and, in that folder, create a new class named `InitializerTests.cs`. Add the following using statements to the class:

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

- Make the class public and inherit from `BaseTest`. Add the Collection attribute to prevent xUnit from running integration tests in parallel. All tests in a collection will run sequentially.

```
namespace AutoLot.Dal.Tests.Initialization
{
  [Collection("Integration Tests")]
  public class InitializerTests : BaseTest
  {
  }
}
```

- Add tests to exercise the initialization code:

```
[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.ClearAndReseedDatabase(Context);
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.NotNull(cars);
  Assert.Equal(9, cars.Count);
}
```



```
[Fact]
public void ShouldClearTheData()
{
  SampleDataInitializer.InitializeData(Context);
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.NotNull(cars);
  Assert.Equal(9, cars.Count);
  SampleDataInitializer.ClearData(Context);
  var cars2 = Context.Cars.IgnoreQueryFilters();
  Assert.NotNull(cars2);
  Assert.Empty(cars2);
}
[Fact]
public void ShouldReseedTheTables()
{
  SampleDataInitializer.ClearAndReseedDatabase(Context);
  var cars = Context.Cars.IgnoreQueryFilters().ToList();
  Assert.NotNull(cars);
  Assert.Equal(9, cars.Count);
}
```

- Run the tests to make sure the initialization code behaves as expected. If using Visual Studio, use the Test menu. If running from the command line, from the same directory as the test project file, enter:

```
dotnet test
```

Part 3: Create the Test Fixture

Test fixtures enable tests to run in a specified context. In this lab, it will make sure the database clean before every test. Note: this process significantly slows down the test runs. These are not unit tests, but integration tests.

- In the Base directory, create a new file named `EnsureAutoLotDatabaseTestFixture.cs`. Add the following using statements to the top:

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

- Update the class to the following:

```
namespace AutoLot.Dal.Tests.Base
{
  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 4: The Customer Tests

The `CustomerTests` exercise basic aspects of using EF Core with the `Customer` class.

- Create a new folder named `ContextTests`, and in the folder create a new class file name `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 Microsoft.EntityFrameworkCore;
using Xunit;
```

- Make the class public, inherit `BaseTest`, and implement `IClassFixture<EnsureAutoLotDatabaseTestFixture>`:

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

- Add in the following tests to exercise the Customer table:

```
[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 ShouldSortByLastNameThenFirstName()
{
  var query = Context.Customers
    .OrderBy(x => x.PersonalInformation.LastName)
    .ThenBy(x => x.PersonalInformation.FirstName);
  var qs = query.ToQueryString();
  var customers = query.ToList();
  if (customers.Count <= 1) { return; }
  for (int x = 0; x < customers.Count - 1; x++)
  {
    var pi = customers[x].PersonalInformation;
    var pi2 = customers[x + 1].PersonalInformation;
    var compareLastName = string.Compare(
      pi.LastName, pi2.LastName, StringComparison.CurrentCultureIgnoreCase);
    Assert.True(compareLastName <= 0);
    if (compareLastName != 0) continue;
    var compareFirstName = string.Compare(
      pi.FirstName, pi2.FirstName, StringComparison.CurrentCultureIgnoreCase);
    Assert.True(compareFirstName <= 0);
  }
}

[Fact]
public void ShouldSortByFirstNameThenLastNameUsingReverse()
{
  var query = Context.Customers
    .OrderBy(x => x.PersonalInformation.LastName)
    .ThenBy(x => x.PersonalInformation.FirstName).Reverse();
  var qs = query.ToQueryString();
  var customers = query.ToList();
  if (customers.Count <= 1) { return; }
  for (int x = 0; x < customers.Count - 1; x++)
  {
    var pi1 = customers[x].PersonalInformation;
    var pi2 = customers[x + 1].PersonalInformation;
    var compareLastName = string.Compare(
      pi1.LastName, pi2.LastName, StringComparison.CurrentCultureIgnoreCase);
    Assert.True(compareLastName >= 0);
    if (compareLastName != 0) continue;
    var compareFirstName = string.Compare(
      pi1.FirstName, pi2.FirstName, StringComparison.CurrentCultureIgnoreCase);
    Assert.True(compareFirstName >= 0);
  }
}
```


Part 5: The Car Tests

The `CarTests` class exercises many more scenarios of using EF Core. The entire class is listed here, as the setup is the same as with the `CustomerTests` class:

```
using System.Collections.Generic;
using System.Linq;
using AutoLot.Dal.Exceptions;
using AutoLot.Models.Entities;
using AutoLot.Dal.Tests.Base;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Microsoft.EntityFrameworkCore.Storage;
using Xunit;
namespace AutoLot.Dal.Tests.ContextTests
{
  [Collection("Integration Tests")]
  public class CarTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
  {
    [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).Query().IgnoreQueryFilters().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);
      }
    }
  }
}
```

Part 6: The Make Tests

The `MakeTests` class exercises filtering on included tables. The entire class is listed here, as the setup is the same as with the previous test classes:

```
using System.Linq;
using AutoLot.Dal.Repos;
using AutoLot.Dal.Repos.Interfaces;
using AutoLot.Dal.Tests.Base;
using Xunit;

namespace AutoLot.Dal.Tests.ContextTests
{
  [Collection("Integration Tests")]
  public class MakeTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
  {
    private readonly IMakeRepo _repo;
    public MakeTests()
    {
      _repo = new MakeRepo(Context);
    }

    public override void Dispose()
    {
      _repo.Dispose();
    }

    [Fact]
    public void ShouldgetMakesThatHaveOrders()
    {
      var makes = _repo.GetOrderByMake().ToList();
      Assert.NotNull(makes);
      Assert.NotEmpty(makes);
      Assert.NotEmpty(makes.Where(x=>x.Cars.Any()));
      Assert.Equal(2,makes.First(m=>m.Id==5).Cars.Count());
      Assert.Equal(1,makes.First(m=>m.Id==1).Cars.Count());
      Assert.Equal(1,makes.First(m=>m.Id==4).Cars.Count());
    }
  }
}
```

Summary

The lab added the integration tests and ran them to make sure the data access layer behaves as expected.

Next steps

In the next part of this tutorial series, you will create the shared services code.