.NET 10 App Dev Hands-On Lab

EF Lab 7 (Optional) – Testing EF Core

This lab walks you through adding and running the tests for the data access layer. This Lab is optional. All of the work is to be completed in the `AutoLot.Dal.Tests` project. Before starting this lab, you must have completed EF Lab 6.

There might be a file named `Usings.cs` or `GlobalUsings.cs`. If it's named `Usings.cs`, rename it `GlobalUsings.cs`. If neither exists, create a new file named `GlobalUsings.cs` and update it to the following:

```
global using AutoLot.Dal.EfStructures;
global using AutoLot.Dal.Exceptions;
global using AutoLot.Dal.Initialization;
global using AutoLot.Dal.Repos;
global using AutoLot.Dal.Repos.Interfaces;

global using AutoLot.Models.Entities;
global using AutoLot.Models.Entities.Configuration;
global using AutoLot.Models.Entities.ComplexTypes;
global using AutoLot.Models.ViewModels;

global using Microsoft.EntityFrameworkCore;
global using Microsoft.EntityFrameworkCore.ChangeTracking;
global using Microsoft.EntityFrameworkCore.Storage;
global using Microsoft.EntityFrameworkCore.Query;
global using Microsoft.Extensions.Configuration;

global using System.Data;

global using Xunit;
```

Part 1: Adding and Running Some Sample Tests

- Rename the `UnitTest1.cs` file to `SampleTests.cs` in the `AutoLot.Dal.Tests` project. Update the code to the following:

```
namespace AutoLot.Dal.Tests;
public class SampleTests
{
  [Fact]
  public void SimpleFactTest()
  {
    Assert.Equal(5, 3 + 2);
  }
  [Theory]
  [InlineData(3, 2, 5)]
  [InlineData(1, -1, 0)]
  public void SimpleTheoryTest(int addend1, int addend2, int expectedResult)
  {
    Assert.Equal(expectedResult, addend1 + addend2);
  }
}
```

- 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 2: Adding Configuration Information and the Base Test Framework

The `AutoLot.Dal.Tests` project uses .NET configuration to set the connection string and create instances of the derived `DbContext`.

Step 1: Adding the configuration file

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

```
{
  "ConnectionStrings": {
    "AutoLot": "Server=(localdb)\\MSSQLLocalDB;Database=AutoLot_Hol;Trusted_Connection=True;"
  }
}
```

- 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.testing.json">
    <CopyToOutputDirectory>Always</CopyToOutputDirectory>
  </None>
</ItemGroup>
```


Step 2: Create the Test Helpers 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`. Make the class public and static:

```
namespace AutoLot.Dal.Tests;

public static class TestHelpers
{
  //implementation goes here
}
```

- 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.testing.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);
  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());
  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`. Make the file public and abstract and implement `IDisposable`:

```
namespace AutoLot.Dal.Tests.Base;
public abstract class BaseTest : IDisposable
{
  //implementation code goes here
}
```

- Add `protected readonly` variables for `IConfiguration`, `ITestOutputHelper`, 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 readonly ITestOutputHelper OutputHelper;
protected BaseTest(ITestOutputHelper outputHelper)
{
  Configuration = TestHelpers.GetConfiguration();
  Context = TestHelpers.GetContext(Configuration);
  OutputHelper = outputHelper;
}

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

- Add transaction support for tests with a single transaction and with shared transactions:

```
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();
  });
}
```

Step 4: Update the GlobalUsings.cs File

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

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

Part 3: Create the Test Fixture

Test fixtures enable tests to run in a specified context. In this lab, it will ensure the database is clean before every test. Note that this process significantly slows down the test runs. These are not unit tests but integration tests, so we are sacrificing speed for thoroughness.

- In the Base directory, create a new file named `EnsureAutoLotDatabaseTestFixture.cs`. 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 Car Tests

The `CarTests` class exercises many more scenarios of using EF Core. Create a folder named `IntegrationTests`, and in that folder, create the `CarTests.cs` class. The entire class is listed here.

```
namespace AutoLot.Dal.Tests.IntegrationTests;

[Collection("Integration Tests")]
public class CarTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
{
  private readonly ICarRepo _carRepo;
  public CarTests(ITestOutputHelper outputHelper) : base(outputHelper)
  {
    _carRepo = new CarRepo(Context);
  }
  public override void Dispose()
  {
    _carRepo.Dispose();
    base.Dispose();
  }
  [Theory]
  [InlineData(1, 2)]
  [InlineData(2, 1)]
  [InlineData(3, 1)]
  [InlineData(4, 2)]
  [InlineData(5, 3)]
  [InlineData(6, 1)]
  public void ShouldGetTheCarsByMake(int makeId, int expectedCount)
  {
    IQueryable<Car> query = Context.Cars.IgnoreQueryFilters().Where(x => x.MakeId == makeId);
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var cars = query.ToList();
    Assert.Equal(expectedCount, cars.Count);
  }
  [Theory]
  [InlineData(1, 1)]
  [InlineData(2, 1)]
  [InlineData(3, 1)]
  [InlineData(4, 2)]
  [InlineData(5, 3)]
  [InlineData(6, 1)]
  public void ShouldGetTheCarsByMakeUsingTheRepo(int makeId, int expectedCount)
  {
    var qs = _carRepo.GetAllBy(makeId).AsQueryable().ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var cars = _carRepo.GetAllBy(makeId).ToList();
    Assert.Equal(expectedCount, cars.Count);
  }
```

```
[Fact]
public void ShouldReturnDrivableCarsWithQueryFilterSet()
{
  IQueryable<Car> query = Context.Cars;
  var qs = query.ToQueryString();
  OutputHelper.WriteLine($"Query: {qs}");
  var cars = query.ToList();
  Assert.NotEmpty(cars);
  Assert.Equal(9, cars.Count);
}
[Fact]
public void ShouldGetAllOfTheCars()
{
  IQueryable<Car> query = Context.Cars.IgnoreQueryFilters();
  var qs = query.ToQueryString();
  OutputHelper.WriteLine($"Query: {qs}");
  var cars = query.ToList();
  Assert.Equal(10, cars.Count);
}
[Fact]
public void ShouldGetAllOfTheDriveableNewCars()
{
  IQueryable<Car> query =
    Context.Cars.IgnoreQueryFilters([CarConfiguration.IsDriveableFilterName]);
  var qs = query.ToQueryString();
  OutputHelper.WriteLine($"Query: {qs}");
  var cars = query.ToList();
  Assert.Equal(10, cars.Count);
}
[Fact]
public void ShouldGetAllOfTheCarsWithMakes()
{
  IIncludableQueryable<Car, Make> query = Context.Cars.Include(c => c.MakeNavigation);
  var qs = query.ToQueryString();
  OutputHelper.WriteLine($"Query: {qs}");
  var cars = query.ToList();
  Assert.Equal(9, cars.Count);
}
[Fact]
public void ShouldGetReferenceRelatedInformationExplicitly()
{
  var car = Context.Cars.First(x => x.Id == 1);
  Assert.Null(car.MakeNavigation);
  var query = Context.Entry(car).Reference(c => c.MakeNavigation).Query();
  var qs = query.ToQueryString();
  OutputHelper.WriteLine($"Query: {qs}");
  query.Load();
  Assert.NotNull(car.MakeNavigation);
}
```



```
  [Fact]
  public void ShouldNotGetTheLemonsUsingFromSql()
  {
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
#pragma warning disable EF1002 // Risk of vulnerability to SQL injection.
    var query = Context.Cars.FromSqlRaw(
      $"Select * from {schemaName}.{tableName}");
#pragma warning restore EF1002 // Risk of vulnerability to SQL injection.
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var cars = query.ToList();
    Assert.Equal(9, cars.Count);
  }
  [Fact]
  public void ShouldGetTheCarsUsingSqlQuery()
  {
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
    var sql =  $"Select Id, IsDrivable,DateBuilt, Price, MakeId, Color, PetName from
{schemaName}.{tableName}";
    var query = Context.Database.SqlQueryRaw<CarViewModel>(sql);
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var cars = query.ToList();
    Assert.Equal(10,cars.Count);
  }
  [Fact]
  public void ShouldGetTheCarsUsingFromSqlWithIgnoreQueryFilters()
  {
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
#pragma warning disable EF1002 // Risk of vulnerability to SQL injection.
    var query = Context.Cars.FromSqlRaw(
      $"Select * from {schemaName}.{tableName}").IgnoreQueryFilters();
#pragma warning restore EF1002 // Risk of vulnerability to SQL injection.
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var cars = query.ToList();
    Assert.Equal(10, cars.Count);
  }
```

```
  [Fact]
  public void ShouldGetOneCarUsingInterpolation()
  {
    var carId = 1;
    var query = Context.Cars
        .FromSqlInterpolated(
            $"Select * from dbo.Inventory where Id = {carId}")
        .Include(x => x.MakeNavigation);
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var car = query.First();
    Assert.Equal("Black", car.Color);
    Assert.Equal("VW", car.MakeNavigation.Name);
  }
  [Fact]
  public void ShouldGetTheCountOfCarsIgnoreQueryFilters()
  {
    var count = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(10, count);
  }
  [Fact]
  public void ShouldGetTheCountOfCars()
  {
    var count = Context.Cars.Count();
    Assert.Equal(9, count);
  }
  [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)
  {
    var entity = Context.Model.FindEntityType($"{typeof(Car).FullName}");
    var tableName = entity.GetTableName();
    var schemaName = entity.GetSchema();
#pragma warning disable EF1002 // Risk of vulnerability to SQL injection.
    var cars = Context
              .Cars
              .FromSqlRaw($"Select * from {schemaName}.{tableName}")
              .Where(x => x.MakeId == makeId).ToList();
#pragma warning restore EF1002 // Risk of vulnerability to SQL injection.
    Assert.Equal(expectedCount, cars.Count);
  }
```

```
[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetTheCountOfCarsByMakeP1(int makeId, int expectedCount)
{
  var count = Context.Cars.Count(x => x.MakeId == makeId);
  Assert.Equal(expectedCount, count);
}

[Theory]
[InlineData(1, 1)]
[InlineData(2, 1)]
[InlineData(3, 1)]
[InlineData(4, 2)]
[InlineData(5, 3)]
[InlineData(6, 1)]
public void ShouldGetTheCountOfCarsByMakeP2(int makeId, int expectedCount)
{
  var count = Context.Cars.Where(x => x.MakeId == makeId).Count();
  Assert.Equal(expectedCount, count);
}
[Theory]
[InlineData(1, true)]
[InlineData(11, false)]
public void ShouldCheckForAnyCarsWithMake(int makeId, bool expectedResult)
{
  var result = Context.Cars.Any(x => x.MakeId == makeId);
  Assert.Equal(expectedResult, result);
}
[Theory]
[InlineData(1, false)]
[InlineData(11, false)]
public void ShouldCheckForAllCarsWithMake(int makeId, bool expectedResult)
{
  var result = Context.Cars.All(x => x.MakeId == makeId);
  Assert.Equal(expectedResult, result);
}
[Theory]
[InlineData(1, "Zippy")]
[InlineData(2, "Rusty")]
[InlineData(3, "Mel")]
[InlineData(4, "Clunker")]
[InlineData(5, "Bimmer")]
[InlineData(6, "Hank")]
[InlineData(7, "Pinky")]
[InlineData(8, "Pete")]
[InlineData(9, "Brownie")]
public void ShouldGetValueFromStoredProc(int id, string expectedName)
{
  Assert.Equal(expectedName, _carRepo.GetPetName(id));
}
[Fact]
```

```
public void ShouldAddACar()
{
  ExecuteInATransaction(RunTheTest);
  void RunTheTest()
  {
    var car = new Car
    {
      Color = "Yellow",
      MakeId = 1,
      PetName = "Herbie"
    };
    var carCount = Context.Cars.Count();
    Context.Cars.Add(car);
    Context.SaveChanges();
    var newCarCount = Context.Cars.Count();
    Assert.Equal(carCount + 1, newCarCount);
  }
}
[Fact]
public void ShouldAddACarWithAttach()
{
  ExecuteInATransaction(RunTheTest);
  void RunTheTest()
  {
    var car = new Car
    {
      Color = "Yellow",
      MakeId = 1,
      PetName = "Herbie"
    };
    var carCount = Context.Cars.Count();
    Context.Cars.Attach(car);
    Assert.Equal(EntityState.Added, Context.Entry(car).State);
    Context.SaveChanges();
    var newCarCount = Context.Cars.Count();
    Assert.Equal(carCount + 1, newCarCount);
  }
}
```

```
[Fact]
public void ShouldAddMultipleCars()
{
  ExecuteInATransaction(RunTheTest);
  void RunTheTest()
  {
    //Have to add 4 to activate batching
    var cars = new List<Car>
        { new() { Color = "Yellow", MakeId = 1, PetName = "Herbie" },
          new() { Color = "White", MakeId = 2, PetName = "Mach 5" },
          new() { Color = "Pink", MakeId = 3, PetName = "Avon" },
          new() { Color = "Blue", MakeId = 4, PetName = "Blueberry" },
        };
    var carCount = Context.Cars.Count();
    Context.Cars.AddRange(cars);
    Context.SaveChanges();
    var newCarCount = Context.Cars.Count();
    Assert.Equal(carCount + 4, 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",
      RadioNavigation = new Radio
      {
        HasSubWoofers = true,
        HasTweeters = true,
        RadioId = "Bose 1234"
      }
    };
    ((List<Car>)make.Cars).Add(car);
    Context.Makes.Add(make);
    var carCount = Context.Cars.Count();
    var makeCount = Context.Makes.Count();
    Context.SaveChanges();
    var newCarCount = Context.Cars.Count();
    var newMakeCount = Context.Makes.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.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.First(c => c.Id == 1);
    Assert.Equal("White", car2.Color);
  }
}
[Fact]
public void ShouldUpdateACarUsingState()
{
  ExecuteInASharedTransaction(RunTheTest);
  void RunTheTest(IDbContextTransaction trans)
  {
    var car = Context.Cars.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,
      IsDrivable = car.IsDrivable,
      DateBuilt = car.DateBuilt
    };
    var context2 = TestHelpers.GetSecondContext(Context, trans);
    context2.Entry(updatedCar).State = EntityState.Modified;
    context2.SaveChanges();
    var context3 = TestHelpers.GetSecondContext(Context, trans);
    var car2 = context3.Cars.First(c => c.Id == 1);
    Assert.Equal("White", car2.Color);
  }
}
```

```
[Fact]
public void ShouldThrowConcurrencyException()
{
  ExecuteInATransaction(RunTheTest);
  void RunTheTest()
  {
    //Get a car record (doesn't matter which one)
    var car = Context.Cars.First();
    //Update the database outside of the context
    Context.Database.ExecuteSqlInterpolated(
      $"Update dbo.Inventory set Color='Pink' where Id = {car.Id}");
    //update the car record in the change tracker
    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.Count();
    var car = Context.Cars.First(c => c.Id == 8);
    Context.Cars.Remove(car);
    Context.SaveChanges();
    var newCarCount = Context.Cars.Count();
    Assert.Equal(carCount - 1, newCarCount);
    Assert.Equal(EntityState.Detached, Context.Entry(car).State);
  }
}
[Fact]
public void ShouldRemoveACarUsingState()
{
  ExecuteInASharedTransaction(RunTheTest);
  void RunTheTest(IDbContextTransaction trans)
  {
    var carCount = Context.Cars.Count();
    var car = Context.Cars.AsNoTracking().First(c => c.Id == 8);
    var context2 = TestHelpers.GetSecondContext(Context, trans);
    //context2.Entry(car).State = EntityState.Deleted;
    context2.Cars.Remove(car);
    context2.SaveChanges();
    var newCarCount = Context.Cars.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.First(c => c.Id == 1);
      Context.Cars.Remove(car);
      Assert.Throws<CustomDbUpdateException>(() => Context.SaveChanges());
    }
  }
  [Fact]
  public void ShouldUpdateInBulk()
  {
    var color = "Pink";
    var makeId = 3;
    ExecuteInATransaction(RunTheTest);
    Context.ChangeTracker.Clear();
    var cars = Context.Cars.Where(c => c.IsDrivable).ToList();
    Assert.False(cars.All(x=>x.Color == color));
    Assert.False(cars.All(x=>x.MakeId == makeId));
    void RunTheTest()
    {
      var count = _carRepo.SetAllDrivableCarsColorAndMakeId(color, makeId);
      var cars = Context.Cars.Where(c => c.IsDrivable).ToList();
      Assert.True(cars.All(x=>x.Color == color));
      Assert.True(cars.All(x=>x.MakeId == makeId));
    }
  }
  [Fact]
  public void ShouldDeleteInBulk()
  {
    var carCount1 = Context.Cars.IgnoreQueryFilters().Count();
    ExecuteInATransaction(RunTheTest);
    Context.ChangeTracker.Clear();
    var carCount2 = Context.Cars.IgnoreQueryFilters().Count();
    Assert.Equal(carCount1,carCount2);
    void RunTheTest()
    {
      var count = _carRepo.DeleteNonDrivableCars();
      Assert.NotEqual(carCount1,Context.Cars.IgnoreQueryFilters().Count());
    }
  }
}
```

Part 5: The Make Tests

The entire class is listed here, as the setup is the same as with the previous test classes:

```
namespace AutoLot.Dal.Tests.IntegrationTests;

[Collection("Integration Tests")]
public class MakeTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
{
  private readonly IMakeRepo _repo;
  public MakeTests(ITestOutputHelper outputHelper) : base(outputHelper)
  {
    _repo = new MakeRepo(Context);
  }
  public override void Dispose()
  {
    _repo.Dispose();
    base.Dispose();
  }
  [Fact]
  public void ShouldGetAllMakesAndCarsThatAreYellow()
  {
    var query = Context.Makes.IgnoreQueryFilters()
            .Include(x => x.Cars.Where(x => x.Color == "Yellow"));
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    var makes = query.ToList();
    Assert.NotNull(makes);
    Assert.NotEmpty(makes);
    Assert.NotEmpty(makes.Where(x => x.Cars.Any()));
    Assert.Empty(makes.First(m => m.Id == 1).Cars);
    Assert.Empty(makes.First(m => m.Id == 2).Cars);
    Assert.Empty(makes.First(m => m.Id == 3).Cars);
    Assert.Single(makes.First(m => m.Id == 4).Cars);
    Assert.Empty(makes.First(m => m.Id == 5).Cars);
  }
  [Fact]
  public void ShouldGetAllMakesAndCarsThatAreYellowAsSplitQuery()
  {
    var query = Context.Makes.AsSplitQuery().IgnoreQueryFilters()
            .Include(x => x.Cars.Where(x => x.Color == "Yellow"));
    var makes = query.ToList();
    Assert.NotNull(makes);
    Assert.NotEmpty(makes);
    Assert.NotEmpty(makes.Where(x => x.Cars.Any()));
    Assert.Empty(makes.First(m => m.Id == 1).Cars);
    Assert.Empty(makes.First(m => m.Id == 2).Cars);
    Assert.Empty(makes.First(m => m.Id == 3).Cars);
    Assert.Single(makes.First(m => m.Id == 4).Cars);
    Assert.Empty(makes.First(m => m.Id == 5).Cars);
  }
```



```
  [Theory]
  [InlineData(1, 1)]
  [InlineData(2, 1)]
  [InlineData(3, 1)]
  [InlineData(4, 2)]
  [InlineData(5, 3)]
  [InlineData(6, 1)]
  public void ShouldGetAllCarsForAMakeExplicitlyWithQueryFilters(int makeId, int carCount)
  {
    var make = Context.Makes.First(x => x.Id == makeId);
    IQueryable<Car> query = Context.Entry(make).Collection(c => c.Cars).Query();
    var qs = query.ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    query.Load();
    Assert.Equal(carCount, make.Cars.Count());
  }
  [Theory]
  [InlineData(1, 2)]
  [InlineData(2, 1)]
  [InlineData(3, 1)]
  [InlineData(4, 2)]
  [InlineData(5, 3)]
  [InlineData(6, 1)]
  public void ShouldGetAllCarsForAMakeExplicitly(int makeId, int carCount)
  {
    var make = Context.Makes.First(x => x.Id == makeId);
    IQueryable<Car> query = Context.Entry(make).Collection(
       c => c.Cars).Query().IgnoreQueryFilters();
    var qs = query.IgnoreQueryFilters().ToQueryString();
    OutputHelper.WriteLine($"Query: {qs}");
    query.Load();
    Assert.Equal(carCount, make.Cars.Count());
  }
  [Theory]
  [InlineData(1, 2)]
  [InlineData(2, 1)]
  [InlineData(3, 1)]
  [InlineData(4, 2)]
  [InlineData(5, 3)]
  [InlineData(6, 1)]
  public void ShouldGetAllCarsForAMakeWithFunction(int makeId, int carCount)
  {
    var cars = Context.GetCarsFor(makeId).IgnoreQueryFilters();
    string qs = Context.GetCarsFor(makeId).ToQueryString();
    Assert.Equal(carCount, cars.Count());
  }
  [Fact]
  public void ShouldUseFunctionInServerSideQuery()
  {
    var query =
      Context.Makes.Where(x => ApplicationDbContext.InventoryCountFor(x.Id) > 2);
    String qs = query.ToQueryString();
    var list = query.ToList();
    Assert.Single(list);
  }
}
```

Part 6: The Driver Tests

The entire class is listed here, as the setup is the same as with the previous test classes:

```
namespace AutoLot.Dal.Tests.IntegrationTests;

[Collection("Integration Tests")]
public class DriverTests : BaseTest, IClassFixture<EnsureAutoLotDatabaseTestFixture>
{
  private readonly IDriverRepo _repo;

  public DriverTests(ITestOutputHelper outputHelper) : base(outputHelper)
  {
    _repo = new DriverRepo(Context);
  }

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

  [Fact(Skip="Just showing the example, doesn't work due to the computed column")]
  public void ShouldUpdateInBulk()
  {
    var person = new Person { FirstName = "George", LastName = "Jetson" };
    var driver = _repo.Find(1);
    Assert.Equal("Fred", driver.PersonInformation.FirstName);
    Assert.Equal("Flintstone", driver.PersonInformation.LastName);
    ExecuteInATransaction(RunTheTest);
    void RunTheTest()
    {
      var count = _repo.ExecuteBulkUpdate(x => x.Id == 1,
         s => s.SetProperty(b => b.PersonInformation, person));
      Context.ChangeTracker.Clear();
      var updatedDriver = _repo.Find(1);
      Assert.Equal(person.FirstName, updatedDriver.PersonInformation.FirstName);
      Assert.Equal(person.LastName, updatedDriver.PersonInformation.LastName);
    }
  }
}
```

Summary

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

Next steps

In the next optional part of this tutorial series, you will update the tables to be temporal.

