Build an ASP.NET Core Service and App with .NET (Core) 5.0 Two-Day Hands-On Lab

Lab 3

This lab walks you through creating the DbContext and the DbContextFactory as well as running your first migration. Prior to starting this lab, you must have completed Lab 2.

Part 1: Create the DbContext

The derived DbContext class is the hub of using EF Core with C#. The lab works on the AutoLot.Dal project.

Step 1: Create the ApplicationDbContext file and its constructor

- Create a new folder in the AutoLot.Dal project named EfStructures. Add a new class to the folder named ApplicationDbContext.cs.
- Add the following using statements to the class:

```
using System;
using AutoLot.Models.Entities;
using AutoLot.Models.Entities.Owned;
using AutoLot.Models.ViewModels;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Storage;
using Microsoft.EntityFrameworkCore.ChangeTracking;
```

- Make the class public, sealed, and inherit from DbContext. Add in a constructor that takes an instance of DbContextOptions and passes it to the base class:

```
namespace AutoLot.Dal.EfStructures
{
  public sealed class ApplicationDbContext : DbContext
  {
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
      : base(options)
    {
    }
  }
}
```

Step 2: Add the public property for query filters

- Add a property to hold the Make Id for the global query filters:

```
public int MakeId { get; set; }
```

Step 3: Add the DbSet<T> properties

- Add a DbSet<T> for each of the model classes.

```
public DbSet<SeriLogEntry>? LogEntries { get; set; }
public DbSet<CreditRisk>? CreditRisks { get; set; }
public DbSet<Customer>? Customers { get; set; }
public DbSet<Make>? Makes { get; set; }
public DbSet<Car>? Cars { get; set; }
public DbSet<Order>? Orders { get; set; }
public DbSet<CustomerOrderViewModel>? CustomerOrderViewModels { get; set; }
```

Step 4: Add the OnModelCreating method and Fluent API Calls

- Add the override for `OnModelCreating`. This method is where the Fluent API code provides additional model information.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
}
```

a) Configure the SerilogEntry entity

```
modelBuilder.Entity<SeriLogEntry>(entity =>
{
  entity.Property(e => e.Properties).HasColumnType("Xml");
  entity.Property(e => e.TimeStamp).HasDefaultValueSql("GetDate()");
});
```

- If you want to exclude the table from migrations, change the code to this:

```
modelBuilder.Entity<SeriLogEntry>(entity =>
{
  entity.ToTable("Serilog", "Logging", t => t.ExcludeFromMigrations());
});
```

Note: Comment this out if you add it, as you will need the SeriLog table later in this HOL.

b) Add the query filters for Make

```
modelBuilder.Entity<Car>().HasQueryFilter(c => c.MakeId == MakeId);
//New in EF Core 5 are bi-directional query filters
modelBuilder.Entity<Order>().HasQueryFilter(e => e.CarNavigation!.MakeId == MakeId);
```

c) Map the CustomerOrderViewModel to a SQL Server View

The view will be created in the next lab

```
modelBuilder.Entity<CustomerOrderViewModel>().HasNoKey().ToView("CustomerOrderView", "dbo");
```

- Optionally, instead of using a view, you can map the view model directory to a SQL statement:

```
modelBuilder.Entity<CustomerOrderViewModel>(entity =>
{
  entity.HasNoKey()
    .ToSqlQuery(@"SELECT c.FirstName, c.LastName, i.Color, i.PetName, m.Name AS Make
        FROM dbo.Orders o
        INNER JOIN dbo.Customers c ON o.CustomerId = c.Id
        INNER JOIN dbo.Inventory  i ON o.CarId = i.Id
        INNER JOIN dbo.Makes m ON m.Id = i.MakeId
  ");
});
```

Note: Comment this out, as you will be using the view to load the data.

d) Configure the FK Cascade rules and Person owned class for CreditRisk

```
modelBuilder.Entity<CreditRisk>(entity =>
{
  entity.HasOne(d => d.CustomerNavigation)
    .WithMany(p => p!.CreditRisks)
    .HasForeignKey(d => d.CustomerId)
    .HasConstraintName("FK_CreditRisks_Customers");

  entity.OwnsOne(o => o.PersonalInformation, pd =>
    {
      pd.Property<string>(nameof(Person.FirstName))
          .HasColumnName(nameof(Person.FirstName))
          .HasColumnType("nvarchar(50)");
      pd.Property<string>(nameof(Person.LastName))
          .HasColumnName(nameof(Person.LastName))
          .HasColumnType("nvarchar(50)");
    });
});
```

e) Configure the Person owned class for Customer

```
modelBuilder.Entity<Customer>(entity =>
{
  entity.OwnsOne(o => o.PersonalInformation, pd =>
  {
    pd.Property(p => p.FirstName).HasColumnName(nameof(Person.FirstName));
    pd.Property(p => p.LastName).HasColumnName(nameof(Person.LastName));
  });
});
```

f) Configure the FK Casecade rules for Makes and Orders

```
modelBuilder.Entity<Make>(entity =>
{
  entity.HasMany(e => e.Cars)
    .WithOne(c => c.MakeNavigation!)
    .HasForeignKey(k => k.MakeId)
    .OnDelete(DeleteBehavior.Restrict)
    .HasConstraintName("FK_Make_Inventory");
});

modelBuilder.Entity<Order>(entity =>
{
  entity.HasOne(d => d.CarNavigation)
    .WithMany(p => p!.Orders)
    .HasForeignKey(d => d.CarId)
    .OnDelete(DeleteBehavior.ClientSetNull)
    .HasConstraintName("FK_Orders_Inventory");
  entity.HasOne(d => d.CustomerNavigation)
    .WithMany(p => p!.Orders)
    .HasForeignKey(d => d.CustomerId)
    .OnDelete(DeleteBehavior.Cascade)
    .HasConstraintName("FK_Orders_Customers");
  entity.HasIndex(cr => new {cr.CustomerId, cr.CarId}).IsUnique(true);
});
```

Part 2: Create the DbContextFactory

The `IDesignTimeDbContextFactory` is used by the design time tools to instantiate a new instance of the `ApplicationDbContext`. The section of the lab works on the AutoLot.Dal project.

Step 1: Create the ApplicationDbContextFactory

- Add a new class named `ApplicationDbContextFactory.cs` the `EfStructures` folder.
- Add the following using statements to the class:

```
using System;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
```

- Make the class public and inherit from `ApplicationDbContextFactory<T>` where T is the DbContext class

```
public class ApplicationDbContextFactory : IDesignTimeDbContextFactory<ApplicationDbContext>
{
}
```

Step 2: Add the CreateDbContext Method

- Add a new method named `CreateDbContext`.
- The method creates a new instance of `ApplicationDbContext` using a hard coded, development connection string:

```
public ApplicationDbContext CreateDbContext(string[] args)
{
  var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
  var connectionString = @"server=.,5433;Database=AutoLot50;User Id=sa;Password=P@ssw0rd;";
  optionsBuilder.UseSqlServer(connectionString);
  Console.WriteLine(connectionString);
  return new ApplicationDbContext(optionsBuilder.Options);
}
```

- **NOTE:** If you are not using the SQL Server Docker container created in Lab 0, update connection string as necessary.

```
Docker:
var connectionString =
@"Server=.,6433;Database=AutoLot50;User ID=sa;Password=P@ssw0rd;MultipleActiveResultSets=true;";
Console.WriteLine(connectionString);

LocalDb:
@"Server=(localdb)\mssqllocaldb;Database=AutoLot50;Integrated
Security=true;MultipleActiveResultSets=true;";
Console.WriteLine(connectionString);
```

Part 3: Update the Database Using Migrations

Migrations can be created and executed using the .NET Core EF Command Line Interface in a command window or the Package Manager Console in Visual Studio. With either option, the commands must be executed from the same directory as the `AutoLot.Dal csproj` file.

The NuGet style commands can be used in the Package Manager Console in Visual Studio if the `Microsoft.EntityFrameworkCore.Tools` package was installed.

Step 1: Create and Execute the Initial Migration

- Open a command prompt in the same directory as the `AutoLot.Dal` project
  OR
  [Visual Studio]Open Package Manager Console (View -> Other Windows -> Package Manager Console) and navigate to the correct directory using:

```
[Windows]cd .\AutoLot.Dal
[Non-Windows]cd ./AutoLot.Dal
```

- Create the initial migration with the following command (-o = output directory, -c = Context File):

```
[Windows]
NOTE: The following lines must be entered as one line – copying and pasting from this document
doesn’t work
dotnet ef migrations add Initial -o EfStructures\Migrations -c
AutoLot.Dal.EfStructures.ApplicationDbContext
NOTE: The above lines must be entered as one line – copying and pasting from this document doesn’t
work
[Non-Windows]
NOTE: The following lines must be entered as one line – copying and pasting from this document
doesn’t work
dotnet ef migrations add Initial -o EfStructures/Migrations -c
AutoLot.Dal.EfStructures.ApplicationDbContext
NOTE: The above lines must be entered as one line – copying and pasting from this document doesn’t
work
```

- This creates three files in the EfStructures\Migrations (EfStructures/Migrations) Directory:
- A file named YYYYMMDDHHmmSS_Initial.cs (where date time is UTC)
- A file named YYYYMMDDHHmmSS _Initial.Designer.cs (same numbers)
- ApplicationDbContextModelSnapshot.cs
- Open up the YYYYMMDDHHmmSS _Initial.cs file. Check the Up and Down methods to make sure the database and table/column creation code is there
- Update the database with the following command:

```
dotnet ef database update
```

- Examine your database in SQL Server Management Studio to make sure the tables were created

Summary

In this lab, you created the `ApplicationDbContext` and the `ApplicationDbContextFactory`. The final step was creating the initial migration and updating the database.

Next steps

In the next part of this tutorial series, you will create the SQL Server objects, including a stored procedure, two views, and a user defined function. Then two computed columns will be added (to the Orders and OrderDetails tables), and finally add in all of the ViewModels as well as the DbQuery types.