Build an EF and ASP.NET Core 3.0 App HOL

Lab 2

This lab walks you through creating the Models and the inherited DbContext as well as running your first migration. Prior to starting this lab, you must have completed Lab 1.

Part 1: Create/Update the Entities

The entities represent the data that is persisted in SQL Server but can be shaped to be more application specific. Go to the AutoLot.Models project and delete the autogenerated `Class1.cs`.

NOTE: The project will not build until this section is complete.

Step 1: Create the Base Entity

1) Create a new folder in the `AutoLot.Models` project named `Entities`. Create a subfolder named `Base` under the `Entities` folder.
2) Add a new class named `EntityBase.cs` to the `Base` folder
3) Add the following using statements to the class:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
```

4) Update the code for the `EntityBase.cs` class to the following:

```
public abstract class EntityBase
{
  [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public int Id { get; set; }
  [Timestamp]
  public byte[] TimeStamp { get; set; }
}
```

NOTES:

- The Key attribute explicitly set the Id field to the primary key for the table.
- The DatabaseGenerated attribute sets the Id field to a SQL Server sequence.
  Note: This is not required because of EF conventions: Any field named Id or [ClassName]Id will be set to the PK of the table, and any primary key field that is numeric will be set to an Identity field in SQL Server.
- Set the TimeStamp property to be a concurrency token using the [Timestamp] attribute. This creates a timestamp datatype in the SQL Server table.

Step 2: Create the Owned Entity

1) Create a new folder named Owned under the Entities folder.
2) Add a new class named Person in the Owned folder, and add the following using statements:

```
using System.ComponentModel.DataAnnotations;
using Microsoft.EntityFrameworkCore;
```

3) Update the code for the `Person.cs` class to the following:

```
[Owned]
public class Person
{
  [Required, StringLength(50)]
  public string FirstName { get; set; } = "New";
  [Required, StringLength(50)]
  public string LastName { get; set; } = "Customer";
}
```

NOTES:

- The `StringLength` attribute sets the field size in SQL Server and is also used for ASP.NET Core validations.
- The owned attribute indicates that this class is wholly contained by another entity.
- The `Required` attribute sets fields to be `NotNull` in SQL Server (and is also used in MVC validations). By EF Convention, any non-nullable .NET type is set to Not Null in SQL Server and any nullable type is set to `Null` unless marked as `Required` via Data Annotations or the Fluent API.

Step 3: Create the Car Entity

1) Create a new class named Car.cs in the Entities folder
2) Add the following using statements to the class:

```
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using AutoLot.Models.Entities.Base;
```

3) Update the code for the `Car.cs` class to the following:

```
[Table("Inventory",Schema = "Dbo")]
public partial class Car : BaseEntity
{
  [Required]
  [DisplayName("Make")]
  public int MakeId { get; set; }

  [ForeignKey(nameof(MakeId))]
  [InverseProperty(nameof(Make.Cars))]
  public Make? MakeNavigation { get; set; }

  [StringLength(50),Required]
  public string Color { get; set; } = "Gold";

  [StringLength(50), Required]
  [DisplayName("Pet Name")]
  public string PetName { get; set; } = "My Precious";

  [JsonIgnore]
  [InverseProperty(nameof(Order.CarNavigation))]
  public IEnumerable<Order> Orders { get; set; } = new List<Order>();

  [NotMapped]
  public string MakeColor => $"{MakeNavigation?.Name} ({Color})";
  public override string ToString()
  {
    // Since the PetName column could be empty, supply
    // the default name of **No Name**.
    return $"{PetName ?? "**No Name**"} is a {Color} {MakeNavigation?.Name} with ID {Id}.";
  }
}
```

NOTES:

- The `Table` attribute sets the data Schema and Table
  NOTE: In EF Core, the database table name defaults to the name of the `DbSet<T>` in the `DbContext` (covered later in this lab).
- The `ForeignKey` attribute explicitly declares the property to use for backing the navigation property to the one end of the one-to-many relationship.
  NOTE: By convention, a property of the same data type as the primary key for the related type and named `<PrimaryKeyPropertyName>`, `<NavigationPropertyName><PrimaryKeyPropertyName>` or `<EntityName><PrimaryKeyPropertyName>` will be the foreign key.
- The `InverseProperty` attribute explicitly declares the other end of the entity's navigation property.
  NOTE: The `Orders` list is the many end of a one-to-many relationship. The `Car` class itself is the one end. While EF conventions can usually determine the inverse properties, it is better to be explicit in defining the relationships.
- NotMapped properties are ignored by EF Core and are not added to the datastore.
- The DisplayName attribute is used by EF Core to change the automatically rendered name for the property
- The `JsonIgnore` attribute prevents JSON serialization from traversing the navigation properties in order to prevent circular serialization.

Step 4: Create the CreditRisk Entity

1) Create a new class named CreditRisk.cs in the Entities folder
2) Add the following using statements to the class:

```
using System.ComponentModel.DataAnnotations.Schema;
using AutoLot.Dal.Models.Entities;
using AutoLot.Models.Entities.Base;
using AutoLot.Models.Entities.Owned;
```

3) Update the code for the `CreditRisk.cs` class to the following:

```
[Table("CreditRisks",Schema = "Dbo")]
public partial class CreditRisk : BaseEntity
{
  public Person PersonalInformation { get; set; } = new Person();
  public int CustomerId { get; set; }

  [ForeignKey(nameof(CustomerId))]
  [InverseProperty(nameof(Customer.CreditRisks))]
  public virtual Customer? CustomerNavigation { get; set; }
}
```

NOTES:

- This class uses the Owned Person class to add the FirstName and LastName fields to the table. The column names will be updated using the fluent API later in this lab.

Step 5: Create the Customer Class

1) Create a new class named Customer.cs in the Entities folder
2) Add the following using statements to the class:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using AutoLot.Models.Entities.Base;
using AutoLot.Models.Entities.Owned;
```

3) Update the code for the `Customer.cs` class to the following:

```
[Table("Customers",Schema = "Dbo")]
public partial class Customer : BaseEntity
{
  public Person PersonalInformation { get; set; } = new Person();

  [JsonIgnore]
  [InverseProperty(nameof(CreditRisk.CustomerNavigation))]
  public IEnumerable<CreditRisk> CreditRisks { get; set; } = new List<CreditRisk>();

  [JsonIgnore]
  [InverseProperty(nameof(Order.CustomerNavigation))]
  public IEnumerable<Order> Orders { get; set; } = new List<Order>();

  [NotMapped]
  public string FullName => $"{PersonalInformation?.FirstName} {PersonalInformation?.LastName}";
}
```

Step 6: Create the Make Entity Class

1) Create a new class named Make.cs in the Entities folder
2) Add the following using statements to the class:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using System.Text.Json.Serialization;
using AutoLot.Models.Entities.Base;
```

3) Update the code for the `Make.cs` class to the following:

```
[Table("Makes", Schema = "dbo")]
public partial class Make : BaseEntity
{
  [StringLength(50), Required] public string Name { get; set; } = "Ford";
  [JsonIgnore]
  [InverseProperty(nameof(Car.MakeNavigation))]
  public IEnumerable<Car> Cars { get; set; } = new List<Car>();
}
```

Step 7: Update the Order Entity Classes

1) Create a new class named Order.cs in the Entities folder
2) Add the following using statements to the class:

```
using System.ComponentModel.DataAnnotations.Schema;
using AutoLot.Models.Entities.Base;
```

3) Update the code for the `Make.cs` class to the following:

```
[Table("Orders",Schema = "Dbo")]
public partial class Order : BaseEntity
{
  public int CustomerId { get; set; }
  public int CarId { get; set; }

  [ForeignKey(nameof(CarId))]
  [InverseProperty(nameof(Car.Orders))]
  public virtual Car? CarNavigation { get; set; }

  [ForeignKey(nameof(CustomerId))]
  [InverseProperty(nameof(Customer.Orders))]
  public virtual Customer? CustomerNavigation { get; set; }
}
```

Step 8: Create the CustomerOrderViewModel Class

1) Create a new folder in the `AutoLot.Models` project named `ViewModels`. Add a new class name CustomerOrderViewModel.cs to the folder
2) Add the following using statements to the class:

```
using System.ComponentModel.DataAnnotations.Schema;
```

3) Update the code for the `CustomerOrderViewModel.cs` class to the following:

```
public class CustomerOrderViewModel
{
  public string? FirstName { get; set; }
  public string? LastName { get; set; }
  public string? Color { get; set; }
  public string? PetName { get; set; }
  public string? Make { get; set; }

  [NotMapped]
  public string FullDetail => $"{FirstName} {LastName} ordered a {Color} {Make} named {PetName}";
  public override string ToString() => FullDetail;
}
```

Part 2: Create the DbContext and DbContextFactory

The derived `DbContext` class is the hub of using EF Core with C#. The `IDesignTimeDbContextFactory` is used by the design time tools to instantiate a new instance of the `ApplicationDbContext`.

Step 1: Create the ApplicationDbContext

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

```
using AutoLot.Models.Entities;
using AutoLot.Models.Entities.Owned;
using AutoLot.Models.ViewModels;
using Microsoft.EntityFrameworkCore;
```

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

```
public class ApplicationDbContext : DbContext
{
  public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options) { }
}
```

4) Add a property to hold the `MakeId` for the global query filters:

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

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

```
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; }
```

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

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

7) Add the following code **<u>into</u>** the `OnModelCreating` method:
   a) Add a global query filter Car entity:

```
modelBuilder.Entity<Car>(entity => {
  entity.HasQueryFilter(c => c.MakeId == MakeId);
});
```

b) Model the CreditRisk entity:

```
modelBuilder.Entity<CreditRisk>(entity =>
{
  //define the relationship between CreditRisk and Customer
  entity.HasOne(d => d.CustomerNavigation)
        .WithMany(p => p.CreditRisks)
        .HasForeignKey(d => d.CustomerId)
        .HasConstraintName("FK_CreditRisks_Customers");
  //Configure the owned property field names
  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)");
          });
});
```

c) Set the SQL Server column names for the owned Person property:

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

d) Define the cascade behavior on the Make entity:

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

e) Update the cascade behaviors for the Order entity:

```
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);
});
```

f) Configure the viewmodel as keyless and tied to a database View:

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

Step 2: Create the ApplicationDbContextFactory

The EF Core Tools Migrate and Database Commands must be able to create a context. If an implementation of the `IDesignTimeDbContextFactory` interface class is found it is used by the EF Core Tools to create an instance of the `ApplicationDbContext`.

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

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

3) Make the class public and implement IDesignTimeDbContextFactory<ApplicationDbContext> and add the CreateDbContext method from the interface:
NOTE: The `args` parameter is not used by EF Core at this time (will be used in EF Core 5).

```
public class ApplicationDbContextFactory : IDesignTimeDbContextFactory<ApplicationDbContext>
{
  public ApplicationDbContext CreateDbContext(string[] args)
  {
  }
}
```

4) Create a `DbContextOptionsBuilder` in the `CreateDbContext` method:

```
var optionsBuilder = new DbContextOptionsBuilder<ApplicationDbContext>();
```

5) Create a variable to hold the connection string and a `Console.WriteLine` to output the connection string (this is useful for debugging).
NOTE: If you are not using the SQL Server Docker container created in Lab 0, update connection string as necessary.

```
Docker:
var connectionString =
@"Server=.,5433;Database=AutoLotWorkshop;User
ID=sa;Password=P@ssw0rd;MultipleActiveResultSets=true;";
Console.WriteLine(connectionString);

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

6) Add the option to use the SQL Server provider, setting the connection string and enabling connection resiliency. Next, configure EF to treat mixed mode query evaluation as an exception and not a warning. Finally, return the configured `ApplicationDbContext` using the `DbContextOptions`.

```
optionsBuilder.UseSqlServer(connectionString);
return new ApplicationDbContext(optionsBuilder.Options);
```

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: Install the EF Core Global Tool

Starting with EF Core 3.0, the EF Core global tool is no longer automatically installed. Install the global tool with the following command:

```
dotnet tool install --global dotnet-ef
```

Step 1: Create and Execute the Initial Migration

1) Open a command prompt in the same directory as the AutoLot.Dal project
OR
Open Package Manager Console (View -> Other Windows -> Package Manager Console) and navigate to the correct directory using: cd .\AutoLot.Dal

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

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

3) This creates three files in the EfStructures\Migrations Directory:
   a) A file named YYYYMMDDHHmmSS_Initial.cs (where date time is UTC)
   b) A file named YYYYMMDDHHmmSS _Initial.Designer.cs (same numbers)
   c) ApplicationDbContextModelSnapshot.cs
4) 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
5) Update the database with the following command:

```
dotnet ef database update
```

6) Examine your database in SQL Server Management Studio or Azure Data Studio to make sure the tables were created correctly.

Summary

In this lab, you created the Entities, the ApplicationDbContext, and the ApplicationDbContextFactory. Finally, you created the initial migration and updated the database.

Next steps

In the next part of this tutorial series, you will create a SQL Server view, the custom exceptions, and support for change tracking events.