.NET App Dev Hands-On Lab

EF Lab 2 – Entities and Models

This lab takes you through creating the `Models` and `ViewModels`. Before starting this lab, you must have completed EF Lab 1.

All work in this lab is to be completed in the `AutoLot.Models` project.

Part 1: The GlobalUsings.cs File

- Begin by renaming the autogenerated `Class1.cs` to `GlobalUsings.cs` and clear out the templated code so it's an empty file. Add the following `global using` statements to the file:

```
global using Microsoft.EntityFrameworkCore;
global using Microsoft.EntityFrameworkCore.Metadata.Builders;
global using System.ComponentModel;
global using System.ComponentModel.DataAnnotations;
global using System.ComponentModel.DataAnnotations.Schema;
global using System.Globalization;
global using System.Xml.Linq;
```

- You can also add global usings statements into the project file. I find this method abstracts the using statements too far from the code. However, if you would like to try this method, here is an example:

```
<ItemGroup>
  <Using Include="Microsoft.EntityFrameworkCore" />
</ItemGroup>
```

Part 2: Creating the Entities in AutoLot.Models

The entities represent the data that is persisted in SQL Server. We use the Table Per Hierarchy (TPH) pattern for this workshop.

Step 1: Create the Base Entity

- Create a new folder in the `AutoLot.Models` project named `Entities`. Create a subfolder named `Base` under the `Entities` folder. Add a new class to the `Base` folder named `BaseEntity.cs`, and update the code to the following:

```
namespace AutoLot.Models.Entities.Base;
public abstract class BaseEntity
{
  [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public int Id { get; set; }
  [Timestamp]
  public long TimeStamp { get; set; }
}
```

Step 2: Create the Person ComplexType Class

`ComplexType` classes can be reused between other entities and folded into the parent entity's table.

- Add a folder named `ComplexTypes` under the `Entities` folder, and add a new class named `Person.cs`. Update the code for the class to the following:

```
namespace AutoLot.Models.Entities.ComplexTypes;
[ComplexType]
public class Person
{
  [Required, StringLength(50)]
  public string FirstName { get; set; }
  [Required, StringLength(50)]
  public string LastName { get; set; }
  [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
  public string FullName { get; set; }
}
```

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

```
global using AutoLot.Models.Entities;
global using AutoLot.Models.Entities.Base;
global using AutoLot.Models.Entities.ComplexTypes;
```

Step 3: Create the Car Entity

- Add a new class to the `Entities` folder named `Car.cs` and update the code to the following:

```
namespace AutoLot.Models.Entities;
[Table("Inventory", Schema = "dbo")]
[Index(nameof(MakeId), Name = "IX_Inventory_MakeId")]
public class Car : BaseEntity
{
  [Required,StringLength(50)]
  public string Color { get; set; }
  public string Price { get; set; }
  [DisplayName("Is Drivable")]
  public bool IsDrivable { get; set; } = true;
  public DateTime? DateBuilt { get; set; }
  [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
  public string Display { get; set; }
  [Required, StringLength(50), DisplayName("Pet Name")]
  public string PetName { get; set; }
  [Required, DisplayName("Make")]
  public int MakeId { get; set; }
}
```


Step 4: Create the CarDriver Entity

- Add a new class to the `Entities` folder named `CarDriver.cs` and update the code to the following:

```
namespace AutoLot.Models.Entities;
[Table("InventoryToDrivers", Schema = "dbo")]
public class CarDriver : BaseEntity
{
  public int DriverId { get; set; }
  [Column("InventoryId")]
  public int CarId { get; set; }
}
```

Step 5: Create the Driver Entity

- Add a new class to the `Entities` folder named `Driver.cs` and update the code to the following:

```
namespace AutoLot.Models.Entities;
[Table("Drivers", Schema = "dbo")]
public class Driver : BaseEntity
{
  public Person PersonInformation { get; set; } = new Person();
}
```

Step 6: Create the Make Entity

- Add a new class to the `Entities` folder named `Make.cs` and update the code to the following:

```
namespace AutoLot.Models.Entities;
[Table("Makes", Schema = "dbo")]
public class Make : BaseEntity
{
  [Required, StringLength(50)]
  public string Name { get; set; }
}
```

Step 7: Create the Radio Entity

- Add a new class to the `Entities` folder named `Radio.cs` and update the code to the following:

```
namespace AutoLot.Models.Entities;
[Table("Radios", Schema = "dbo")]
public class Radio : BaseEntity
{
  public bool HasTweeters { get; set; }
  public bool HasSubWoofers { get; set; }
  [Required, StringLength(50)]
  public string RadioId { get; set; }
  [Column("InventoryId")]
  public int CarId { get; set; }
}
```


Step 8: Create the Logging Entity

The `SeriLog` logging framework has an option to write log entries to a database table. We will use EF Core to create the tables for us, even though it doesn't represent a domain entity.

- Add a new class to the `Entities` folder named `SeriLogEntry.cs` and update the code to the following:

```
namespace AutoLot.Models.Entities;
[Table("SeriLogs", Schema = "Logging")]
public class SeriLogEntry
{
  [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
  public int Id { get; set; }
  public string Message { get; set; }
  public string MessageTemplate { get; set; }
  [MaxLength(128)]
  public string Level { get; set; }
  [DataType(DataType.DateTime)]
  public DateTime TimeStamp { get; set; }
  public string Exception { get; set; }
  public string Properties { get; set; }
  public string LogEvent { get; set; }
  public string SourceContext { get; set; }
  public string RequestPath { get; set; }
  public string ActionName { get; set; }
  public string ApplicationName { get; set; }
  public string MachineName { get; set; }
  public string FilePath { get; set; }
  public string MemberName { get; set; }
  public int? LineNumber { get; set; }
  [NotMapped]
  public XElement PropertiesXml => (Properties != null) ? XElement.Parse(Properties) : null;
}
```

Part 3: Creating the Navigation Properties

Navigation properties represent foreign key relationships between entities. Collection navigation properties represent the "many" end of a one-to-many or many-to-many relationship and Reference navigation properties represent the "one" end of a one-to-many or one-to-one relationship.

NOTE: The code will not successfully compile until this section is completed.

Step 1: Update the Car Entity

- Update the `Car` entity by adding the following reference and collection navigation properties:

```
public class Car: BaseEntity
{
  //omitted for brevity

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

  [InverseProperty(nameof(Radio.CarNavigation))]
  public Radio RadioNavigation { get; set; }

  [InverseProperty(nameof(Driver.Cars))]
  public IEnumerable<Driver> Drivers { get; set; } = new List<Driver>();

  [InverseProperty(nameof(CarDriver.CarNavigation))]
  public IEnumerable<CarDriver> CarDrivers { get; set; } = new List<CarDriver>();
}
```

- Add the `[NotMapped]` `MakeName` property and the override for `ToString()`, both of which use the `MakeNavigation` reference navigation property:

```
public class Car : BaseEntity
{
  //omitted for brevity

  [NotMapped]
  public string MakeName => MakeNavigation?.Name ?? "Unknown";

  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}.";
  }
}
```

Step 2: Update the CarDriver Entity

- Update the `CarDriver` entity by adding the following reference navigation properties:

```
public class CarDriver : BaseEntity
{
  public int DriverId { get; set; }
  [ForeignKey(nameof(DriverId))]
  public Driver DriverNavigation { get; set; }

  [Column("InventoryId")]
  public int CarId { get; set; }
  [ForeignKey(nameof(CarId))]
  public Car CarNavigation { get; set; }
}
```

Step 3: Update the Driver Entity

- Update the `Driver` entity by adding the following collection navigation properties:

```
public class Driver : BaseEntity
{
  public Person PersonInformation { get; set; } = new Person();
  [InverseProperty(nameof(Car.Drivers))]
  public IEnumerable<Car> Cars { get; set; } = new List<Car>();
  [InverseProperty(nameof(CarDriver.DriverNavigation))]
  public IEnumerable<CarDriver> CarDrivers { get; set; } = new List<CarDriver>();
}
```

Step 4: Update the Make Entity

- Update the `Make` entity by adding the following collection navigation property:

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

Step 5: Update the Radio Entity

- Update the `Radio` entity by adding the following reference and collection navigation properties:

```
public class Radio : BaseEntity
{
  //omitted for brevity
  public int CarId { get; set; }
  [ForeignKey(nameof(CarId))]
  public Car CarNavigation { get; set; }
}
```

- The project should now build.

Part 4: Configure the Entities

The Fluent API is used to fine-tune the models. The `IEntityTypeConfiguration<T>` interface (introduced in EF Core 6) allows for placing each entity's Fluent API code in specific configuration classes. This section configures the entities, the final step to utilize these configuration classes will be coded when the `ApplicationDbContext` is built.

Start by creating a folder named `Configuration` under the Entities folder.

Step 1: Configure the Car Entity

- Add a new class named `CarConfiguration` in the `Configuration` folder. Clear out the code and update it to match the following:

```
namespace AutoLot.Models.Entities.Configuration;
public class CarConfiguration : IEntityTypeConfiguration<Car>
{
  public void Configure(EntityTypeBuilder<Car> builder)
  {
    builder.Property(e => e.TimeStamp).HasConversion<byte[]>();
    builder.ToTable( b => b.IsTemporal(t =>
    {
      t.HasPeriodEnd("ValidTo");
      t.HasPeriodStart("ValidFrom");
      t.UseHistoryTable("InventoryAudit");
    }));
    builder.HasQueryFilter(c => c.IsDrivable);
    builder.Property(p => p.IsDrivable).HasDefaultValue(true);
    builder.Property(e => e.DateBuilt).HasDefaultValueSql("getdate()");
    builder.Property(e => e.Display)
           .HasComputedColumnSql("[PetName] + ' (' + [Color] + ')'", stored: true);
    CultureInfo provider = new("en-us");
    NumberStyles style = NumberStyles.Number | NumberStyles.AllowCurrencySymbol;
    builder.Property(p => p.Price).HasConversion(
            v => decimal.Parse(v, style, provider),
            v => v.ToString("C2"));
    builder.HasOne(d => d.MakeNavigation).WithMany(p => p.Cars).HasForeignKey(d => d.MakeId)
           .OnDelete(DeleteBehavior.ClientSetNull).HasConstraintName("FK_Inventory_Makes_MakeId");
    builder.HasMany(p => p.Drivers).WithMany(p => p.Cars).UsingEntity<CarDriver>(
        j => j.HasOne(cd => cd.DriverNavigation).WithMany(d => d.CarDrivers)
          .HasForeignKey(nameof(CarDriver.DriverId))
          .HasConstraintName("FK_InventoryDriver_Drivers_DriverId")
          .OnDelete(DeleteBehavior.Cascade),
        j => j.HasOne(cd => cd.CarNavigation).WithMany(c => c.CarDrivers)
          .HasForeignKey(nameof(CarDriver.CarId))
          .HasConstraintName("FK_InventoryDriver_Inventory_InventoryId")
          .OnDelete(DeleteBehavior.ClientCascade),
        j => { j.HasKey(x => x.Id);
               j.HasIndex(cd => new { cd.CarId, cd.DriverId }).IsUnique(true);
             });
    }
}
```

- Add the following to the `GlobalUsings.cs` file:

```
global using AutoLot.Models.Entities.Configuration;
```

- Add the `[EntityTypeConfiguration]` attribute to the `Car` class:

```
[Table("Inventory", Schema = "dbo")]
[Index(nameof(MakeId), Name = "IX_Inventory_MakeId")]
[EntityTypeConfiguration(typeof(CarConfiguration))]
public class Car : BaseEntity
{
  //omitted for brevity
}
```

Step 2: Configure the CarDriver Entity

- Add a new class named `CarDriverConfiguration` in the `Configuration` folder and update it to the following:

```
namespace AutoLot.Models.Entities.Configuration;
public class CarDriverConfiguration : IEntityTypeConfiguration<CarDriver>
{
  public void Configure(EntityTypeBuilder<CarDriver> builder)
  {
    builder.Property(e => e.TimeStamp).HasConversion<byte[]>();
    builder.ToTable( b => b.IsTemporal(t =>
    {
      t.HasPeriodEnd("ValidTo");
      t.HasPeriodStart("ValidFrom");
      t.UseHistoryTable("InventoryToDriversAudit");
    }));
    builder.HasQueryFilter(cd => cd.CarNavigation.IsDrivable);
  }
}
```

- Add the `[EntityTypeConfiguration]` attribute to the `CarDriver` class:

```
[Table("InventoryToDrivers", Schema = "dbo")]
[EntityTypeConfiguration(typeof(CarDriverConfiguration))]
public class CarDriver : BaseEntity
{
  //omitted for brevity
}
```

Step 3: Configure the Driver Entity

- Add a new class named `DriverConfiguration` in the `Configuration` folder and update it to the following:

```
namespace AutoLot.Models.Entities.Configuration;
public class DriverConfiguration : IEntityTypeConfiguration<Driver>
{
  public void Configure(EntityTypeBuilder<Driver> builder)
  {
    builder.Property(e => e.TimeStamp).HasConversion<byte[]>();
    builder.ToTable(b => b.IsTemporal(t =>
    {
      t.HasPeriodEnd("ValidTo");
      t.HasPeriodStart("ValidFrom");
      t.UseHistoryTable("DriverAudit");
    }));
    builder.ComplexProperty(cp => cp.PersonInformation,
      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)");
        pd.Property(p => p.FullName)
          .HasColumnName(nameof(Person.FullName))
          .HasComputedColumnSql("[LastName] + ', ' + [FirstName]");
        pd.IsRequired(true);
    });
  }
}
```

- Add the `[EntityTypeConfiguration]` attribute to the `Driver` class:

```
[Table("Drivers", Schema = "dbo")]
[EntityTypeConfiguration(typeof(DriverConfiguration))]
public class Driver : BaseEntity
{
  //omitted for brevity
}
```

Step 4: Configure the Make Entity

- Add a new class named `MakeConfiguration` in the `Configuration` folder and update it to the following:

```
namespace AutoLot.Models.Entities.Configuration;
public class MakeConfiguration : IEntityTypeConfiguration<Make>
{
  public void Configure(EntityTypeBuilder<Make> builder)
  {
    builder.Property(e => e.TimeStamp).HasConversion<byte[]>();
    builder.ToTable( b => b.IsTemporal(t =>
    {
      t.HasPeriodEnd("ValidTo");
      t.HasPeriodStart("ValidFrom");
      t.UseHistoryTable("MakesAudit");
    }));
  }
}
```

- Add the `[EntityTypeConfiguration]` attribute to the `Make` class:

```
[Table("Makes", Schema = "dbo")]
[EntityTypeConfiguration(typeof(MakeConfiguration))]
public class Make : BaseEntity
{
  //omitted for brevity
}
```

Step 5: Configure the Radio Entity

- Add a new class named `RadioConfiguration` in the `Configuration` folder and update it to the following:

```
namespace AutoLot.Models.Entities.Configuration;
public class RadioConfiguration : IEntityTypeConfiguration<Radio>
{
  public void Configure(EntityTypeBuilder<Radio> builder)
  {
    builder.Property(e => e.TimeStamp).HasConversion<byte[]>();
    builder.ToTable( b => b.IsTemporal(t =>
    {
       t.HasPeriodEnd("ValidTo");
       t.HasPeriodStart("ValidFrom");
       t.UseHistoryTable("RadiosAudit");
    }));
    builder.HasIndex(e => e.CarId, "IX_Radios_CarId").IsUnique();
    builder.HasQueryFilter(e => e.CarNavigation.IsDrivable);
    builder.HasOne(d => d.CarNavigation).WithOne(p => p.RadioNavigation)
           .HasForeignKey<Radio>(d => d.CarId);
  }
}
```

- Add the `[EntityTypeConfiguration]` attribute to the `Radio` class:

```
[Table("Radios", Schema = "dbo")]
[EntityTypeConfiguration(typeof(RadioConfiguration))]
public class Radio : BaseEntity
{
  //omitted for brevity
}
```

Step 6: Configure the SeriLogEntry Entity

- Add a new class named `SeriLogEntryConfiguration` in the `Configuration` folder and update it to the following:

```
namespace AutoLot.Models.Entities.Configuration;
public class SeriLogEntryConfiguration : IEntityTypeConfiguration<SeriLogEntry>
{
  public void Configure(EntityTypeBuilder<SeriLogEntry> builder)
  {
    builder.Property(e => e.Properties).HasColumnType("Xml");
    builder.Property(e => e.TimeStamp).HasDefaultValueSql("GetDate()");
    builder.Property(p => p.LineNumber).HasDefaultValue(0).HasSentinel(-1);
  }
}
```

- Add the `[EntityTypeConfiguration]` attribute to the `SeriLogEntry` class:

```
[Table("SeriLogs", Schema = "Logging")]
[EntityTypeConfiguration(typeof(SeriLogEntryConfiguration))]
public class SeriLogEntry
{
  //omitted for brevity
}
```

Part 5: Create the ViewModels in AutoLot.Models

The applications use two View Models.

- Create a new folder named `ViewModels` under the `AutoLot.Models` project. Add another folder named `Configuration` under the `ViewModels` folder.

Step 1: Create and Configure the CarViewModel

- Add a new class named `CarViewModel.cs` into the `ViewModels` folder and update the code for the class to the following:

```
namespace AutoLot.Models.ViewModels;
[Keyless]
public class CarViewModel
{
  public int Id { get; set; }
  public bool IsDrivable { get; set; }
  public DateTime? DateBuilt { get; set; }
  public string Price { get; set; }
  public int MakeId { get; set; }
  public string Color { get; set; } = string.Empty;
  public string PetName { get; set; } = string.Empty;
}
```

- Add a new class named `CarViewModelConfiguration.cs` into the `Configuration` folder. Update the code for the class to the following:

```
namespace AutoLot.Models.ViewModels.Configuration;
public class CarViewModelConfiguration : IEntityTypeConfiguration<CarViewModel>
{
  public void Configure(EntityTypeBuilder<CarViewModel> builder)
  {
    builder.HasNoKey();
    builder.ToTable(t => t.ExcludeFromMigrations());
    CultureInfo provider = new("en-us");
    NumberStyles style = NumberStyles.Number | NumberStyles.AllowCurrencySymbol;
    builder.Property(p => p.Price)
           .HasConversion(
                v => decimal.Parse(v, style, provider),
                v => v.ToString("C2"));
    }
}
```

- Add the following to the `GlobalUsings.cs` file:

```
global using AutoLot.Models.ViewModels;
global using AutoLot.Models.ViewModels.Configuration;
```

- Add the `[EntityTypeConfiguration]` attribute to the class:

```
[Keyless]
[EntityTypeConfiguration(typeof(CarViewModelConfiguration))]
public class CarViewModel
{
{
  //omitted for brevity
}
```

Step 2: Create the TemporalViewModel Class

This class is used when querying temporal tables and does not exist in the database.

- Add a new class named `TemporalViewModel.cs` into the `ViewModels` folder and update the code to the following:

```
namespace AutoLot.Models.ViewModels;
public class TemporalViewModel<T> where T: BaseEntity, new()
{
  public T Entity { get; set; }
  public DateTime ValidFrom { get; set; }
  public DateTime ValidTo { get; set; }
}
```

Summary

In this lab, you created the Models (Entities) and the `ViewModels` for the applications.

Next steps

In the next part of this tutorial series, you will create the `DbContext`, `DbContextFactory`, and run your first migration.