Build an EF and ASP.NET Core 3.0 App HOL

Lab 2

This lab walks you through the first part of 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 and can be shaped to be more application specific. Begin by deleting the autogenerated `Class1.cs`.

Step 1: Create the Base Entity

1) Create a new folder in the `SpyStore.Hol.Models` project named `Entities`. Create a subfolder named `Base` under the `Entities` folder.
2) Add a new class to the `Base` folder named `EntityBase.cs`
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: Update the Category Entity

NOTE: Copy the entities from the Lab2/Assets/Models folder. This part of the lab will update the attributes and create the navigation properties.

1) Update the code for the `Category.cs` class to the following:

```
[Table("Categories", Schema = "Store")]
public class Category : EntityBase
{
  [DataType(DataType.Text), MaxLength(50)]
  public string CategoryName { get; set; }
  [InverseProperty(nameof(Product.CategoryNavigation))]
  public List<Product> Products { get; set; } = new List<Product>();
}
```

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 `MaxLength` attribute sets the field size in SQL Server and is also used for ASP.NET Core validations.
- The DataType attribute sets the specific SQL Server data type, which is more specific than the CLR data type of string.
- The `InverseProperty` attribute explicitly declares the other end of the entity's navigation property.
  NOTE: The `Products` list is the many end of a one-to-many relationship. The `Category` 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.

Step 3: Update the Customer Entity

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

```
[Table("Customers", Schema = "Store")]
public class Customer : EntityBase
{
  [DataType(DataType.Text), MaxLength(50), Display(Name = "Full Name")]
  public string FullName { get; set; }
  [Required,EmailAddress]
  [DataType(DataType.EmailAddress), MaxLength(50), Display(Name = "Email Address")]
  public string EmailAddress { get; set; }
  [Required,DataType(DataType.Password), MaxLength(50)]
  public string Password { get; set; }
  [JsonIgnore, InverseProperty(nameof(Order.CustomerNavigation))]
  public List<Order> Orders { get; set; } = new List<Order>();
  [JsonIgnore, InverseProperty(nameof(ShoppingCartRecord.CustomerNavigation))]
  public List<ShoppingCartRecord> ShoppingCartRecords { get; set; } =
    new List<ShoppingCartRecord>();
}
```

NOTES:

- The `Display` attribute sets the text for view labels in ASP.NET Core.
- 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.
- The `JsonIgnore` attribute prevents JSON serialization from traversing the navigation properties in order to prevent circular serialization.

Step 4: Update the Order Entity Classes

The `Order` entity has an additional base class beyond `EntityBase` that will be used by the View Models.

Update the OrderBase Class

1) Update the code for the `OrderBase.cs` class (in the `Base` folder) to the following:

```
public class OrderBase : EntityBase
{
  [DataType(DataType.Date),Display(Name = "Date Ordered")]
  public DateTime OrderDate { get; set; }
  [DataType(DataType.Date), Display(Name = "Date Shipped")]
  public DateTime ShipDate { get; set; }
  [Display(Name = "Customer")]
  public int CustomerId { get; set; }
}
```

Update the Order Entity Class

1) Update the code for the `Order.cs` class to the following:

```
[Table("Orders", Schema = "Store")]
public class Order : OrderBase
{
  [ForeignKey(nameof(CustomerId))]
  public Customer CustomerNavigation { get; set; }
  [InverseProperty(nameof(OrderDetail.OrderNavigation))]
  public List<OrderDetail> OrderDetails { get; set; } = new List<OrderDetail>();
}
```

NOTES:

- 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.

Step 5: Update the OrderDetail Entity Classes

The `OrderDetail` entity has an additional base class beyond `EntityBase` that will be used by the View Models.

Update the OrderDetailBase Class

1) Update the code for the `OrderBase.cs` class to the following:

```
public class OrderDetailBase : EntityBase
{
  [Required]
  public int OrderId { get; set; }
  [Required]
  public int ProductId { get; set; }
  [Required]
  public int Quantity { get; set; }
  [Required, DataType(DataType.Currency), Display(Name = "Unit Cost")]
  public decimal UnitCost { get; set; }
}
```

Update the OrderDetail Entity Class

1) Update the code for the `OrderDetail.cs` class to the following:

```
[Table("OrderDetails", Schema = "Store")]
public class OrderDetail : OrderDetailBase
{
  [ForeignKey(nameof(OrderId))]
  public Order OrderNavigation { get; set; }
  [ForeignKey(nameof(ProductId))]
  public Product ProductNavigation { get; set; }
}
```

Step 6: Update the Product Entity Classes

The Product entity takes advantage of the new `[Owned]` attribute to encapsulate properties. Owned types can only be contained by other types.

Update the ProductDetails Owned Class

1) Update the code for the `ProductDetails.cs` class to the following:

```
[Owned]
public class ProductDetails
{
  [MaxLength(3800)]
  public string Description { get; set; }
  [MaxLength(50)]
  public string ModelNumber { get; set; }
  [MaxLength(50)]
  public string ModelName { get; set; }
  [MaxLength(150)]
  public string ProductImage { get; set; }
  [MaxLength(150)]
  public string ProductImageLarge { get; set; }
  [MaxLength(150)]
  public string ProductImageThumb { get; set; }
}
```

NOTES:

- The Owned attribute allows a class to become part of the parent class. The database property names will default to <parent_propertyname>_<ownedclass_propertyname> unless modified with the Fluent API.

Update the Product Entity Class

1) Update the code for the `Product.cs` class to the following:

```
[Table("Products", Schema = "Store")]
public class Product : EntityBase
{
  public ProductDetails Details { get; set; } = new ProductDetails();
  public bool IsFeatured { get; set; }
  [DataType(DataType.Currency)]
  public decimal UnitCost { get; set; }
  [DataType(DataType.Currency)]
  public decimal CurrentPrice { get; set; }
  public int UnitsInStock { get; set; }
  [Required]
  public int CategoryId { get; set; }
  [JsonIgnore, ForeignKey(nameof(CategoryId))]
  public Category CategoryNavigation { get; set; }
  [InverseProperty(nameof(ShoppingCartRecord.ProductNavigation))]
  public List<ShoppingCartRecord> ShoppingCartRecords { get; set; }
      = new List<ShoppingCartRecord>();
  [InverseProperty(nameof(OrderDetail.ProductNavigation))]
  public List<OrderDetail> OrderDetails { get; set; } = new List<OrderDetail>();
  [NotMapped]
  public string CategoryName => CategoryNavigation?.CategoryName;
}
```

NOTES:

- The `NotMapped` attribute isolates the decorated property or attribute from the database. In this example, it is used to expose the `CategoryName` at the top level so it's available when serialized into JSON.

Step 7: Update the ShoppingCartRecord Entity Classes

The `ShoppingCartRecord` entity has an additional base class beyond `EntityBase` that will be used by the View Models.

Update the ShoppingCartRecodBase Class

1) Update the code for the `ShoppingCartRecordBase.cs` class to the following:

```
public class ShoppingCartRecordBase : EntityBase
{
  [DataType(DataType.Date), Display(Name = "Date Created")]
  public DateTime? DateCreated { get; set; }
  [Required]
  public int CustomerId { get; set; }
  [Required]
  public int Quantity { get; set; }
  [DataType(DataType.Currency), Display(Name = "Line Total")]
  public decimal LineItemTotal { get; set; }
  [Required]
  public int ProductId { get; set; }
}
```

Update the ShoppingCartRecord Entity Class

1) Update the code for the `ShoppingCartRecord.cs` class to the following:

```
[Table("ShoppingCartRecords", Schema = "Store")]
public class ShoppingCartRecord : ShoppingCartRecordBase
{
  [JsonIgnore]
  [ForeignKey(nameof(CustomerId))]
  public Customer CustomerNavigation { get; set; }
  [JsonIgnore]
  [ForeignKey(nameof(ProductId))]
  public Product ProductNavigation { get; set; }
}
```

Part 2: Update 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 `StoreContext`.

NOTE: Copy the StoreContext.cs and StoreContextFactory.cs files from the Lab2/Assets/EfStructures folder. This part of the lab will update those two files.

Step 1: Update the StoreContext

1) Add a property to hold the `Customer Id` for the global query filters:

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

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

```
public DbSet<Category> Categories { get; set; }
public DbSet<Customer> Customers { get; set; }
public DbSet<Order> Orders { get; set; }
public DbSet<OrderDetail> OrderDetails { get; set; }
public DbSet<Product> Products { get; set; }
public DbSet<ShoppingCartRecord> ShoppingCartRecords { get; set; }
```

3) Add the following code into the `OnModelCreating` method:

   a) Add a unique index for the `EmailAddress` property of the `Customer` table:

```
modelBuilder.Entity<Customer>(entity =>
{
  entity.HasIndex(e => e.EmailAddress).HasName("IX_Customers").IsUnique();
});
```

   b) Set the global query filter for the table to limit records to the current `Customer Id`. Then set the SQL Server data type and the default value for the `OrderDate` and `ShipDate` properties of the `Order` table:

```
modelBuilder.Entity<Order>().HasQueryFilter(x => x.CustomerId == CustomerId);
modelBuilder.Entity<Order>(entity =>
{
  entity.Property(e => e.OrderDate).HasColumnType("datetime").HasDefaultValueSql("getdate()");
  entity.Property(e => e.ShipDate).HasColumnType("datetime").HasDefaultValueSql("getdate()");
});
```

   c) Set the SQL Server data type for the `UnitCost` property:

```
modelBuilder.Entity<OrderDetail>(entity =>
{
  entity.Property(e => e.UnitCost).HasColumnType("money");
});
```

d) Set the SQL Server datatypes for the `Product` properties `UnitCost` and `CurrentPrice` to "money". Set the column names for the `Owned` entity explicitly:

```
modelBuilder.Entity<Product>(entity =>
{
  entity.Property(e => e.UnitCost).HasColumnType("money");
  entity.Property(e => e.CurrentPrice).HasColumnType("money");
  entity.OwnsOne(o => o.Details,
    pd =>
      {
        pd.Property(p => p.Description).HasColumnName(nameof(ProductDetails.Description));
        pd.Property(p => p.ModelName).HasColumnName(nameof(ProductDetails.ModelName));
        pd.Property(p => p.ModelNumber).HasColumnName(nameof(ProductDetails.ModelNumber));
        pd.Property(p => p.ProductImage).HasColumnName(nameof(ProductDetails.ProductImage));
        pd.Property(p => p.ProductImageLarge)
                        .HasColumnName(nameof(ProductDetails.ProductImageLarge));
        pd.Property(p => p.ProductImageThumb)
                        .HasColumnName(nameof(ProductDetails.ProductImageThumb));
      });
});
```

e) Set the global query filter for the `ShoppingCartRecord` entity, then create a unique index for the `ProductId` and `CustomerId` fields for the `ShoppingCartRecord` table and set the default values for the `DateCreated` and the `Quantity` fields.
NOTE: Complex indices can only be set using the FluentAPI in EF Core.

```
modelBuilder.Entity<ShoppingCartRecord>().HasQueryFilter(x => x.CustomerId == CustomerId);
modelBuilder.Entity<ShoppingCartRecord>(entity =>
{
  entity.HasIndex(e => new { ShoppingCartRecordId = e.Id, e.ProductId, e.CustomerId })
    .HasName("IX_ShoppingCart").IsUnique();
  entity.Property(e => e.DateCreated).HasColumnType("datetime").HasDefaultValueSql("getdate()");
  entity.Property(e => e.Quantity).HasDefaultValue(1);
});
```

Step 2: Update the DbContextFactory

The EF Core Tools Migrate and Database Commands must be able to create a context. In prior versions of EF Core, a parameterless constructor was used. That conflicts with the `DbContextPool` in ASP.NET Core 2. If an implementation of the `IDesignTimeDbContextFactory` interface class is found it is used by the EF Core Tools to create an instance of the `StoreContext`.

1) The interface has one method, `CreateDbContext`. Create a `DbContextOptionsBuilder` in the `CreateDbContext` method:

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

2) 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=.,6433;Database=SpyStoreHol;User ID=sa;Password=P@ssw0rd;MultipleActiveResultSets=true;";
Console.WriteLine(connectionString);

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

3) 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, rerun the configured `StoreContext` using the `DbContextOptions`.

```
optionsBuilder
  .UseSqlServer(connectionString,options => options.EnableRetryOnFailure());
return new StoreContext(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 `SpyStore.Hol.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: Installed 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 -g dotnet-ef --version 3.0.0
```

Step 1: Create and Execute the Initial Migration

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

```
cd .\SpyStore.Hol.Dal
```

1) 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
SpyStore.Hol.Dal.EfStructures.StoreContext
```
NOTE: The above lines must be entered as one line – copying and pasting from this document doesn't work

2) 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) StoreContextModelSnapshot.cs
3) 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
4) Update the database with the following command:

```
dotnet ef database update
```

5) Examine your database in SQL Server Management Studio to make sure the tables were created:

Summary

In this lab, you created the Entities, the StoreContext, and the StoreContextFactory. Finally, you created the initial migration and updated 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.