Build an EF and ASP.NET Core 2.1 App HOL

Lab 3

Welcome to the Build an Entity Framework Core and ASP.NET Core 2.1 Application in a Day Hands-on Lab. This lab walks you through creating a user defined function, assigning it to a column on the database, and adding in all of the view models.

Prior to starting this lab, you must have completed Lab 2.

Part 1: Create the User Defined Function

Step 1: Create the Migration for the UDF

1) Open Package Manager Console (View -> Other Windows -> Package Manager Console)
2) Change to the SpyStore_HOL.DAL directory:

```
cd .\SpyStore_HOL.DAL
```

3) Create an empty migration (but do **NOT** run database update):

NOTE: The following lines must be entered as one line in Package Manager Console – copying and pasting from this document doesn't work

```
dotnet ef migrations add TSQL -o EfStructures\Migrations -c
SpyStore_HOL.DAL.EfStructures.StoreContext
```

NOTE: The above lines must be entered as one line in Package Manager Console – copying and pasting from this document doesn't work

4) Open up the new migration file (named `<timestamp>_TSQL.cs`). In the Up method, add the following to create the User Defined Function:

```
string sql = @"CREATE FUNCTION Store.GetOrderTotal ( @OrderId INT )
    RETURNS MONEY WITH SCHEMABINDING
    BEGIN
    DECLARE @Result MONEY;
    SELECT @Result = SUM([Quantity]*[UnitCost]) FROM Store.OrderDetails
    WHERE OrderId = @OrderId; RETURN @Result END";
migrationBuilder.Sql(sql);
```

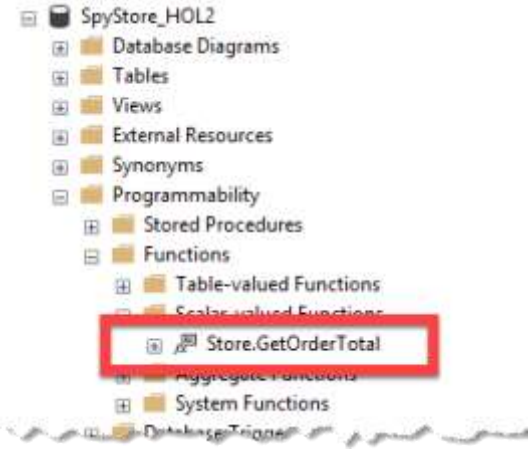
5) In the Down method, add the following code:

```
migrationBuilder.Sql("DROP FUNCTION [Store].[GetOrderTotal]");
```

6) SAVE THE MIGRATION FILE
7) Update the database by executing the migration:

```
dotnet ef database update
```

8) Check the database to make sure the function exists:



Part 2: Add the Calculated Field to the Order Table

Step 1: Update the Order Model

1) Open the Order.cs file in the Models project and add the following property:

```
[Display(Name = "Total")]
public decimal? OrderTotal { get; set; }
```

Step 2: Update the StoreContext OnModelCreating Method

1) Open the StoreContext.cs file in the DAL project, and add the following Fluent API command in the OnModelCreating method to the Order entity:

```
modelBuilder.Entity<Order>(entity =>
{
  entity.Property(e => e.OrderDate).HasColumnType("datetime").HasDefaultValueSql("getdate()");
  entity.Property(e => e.ShipDate).HasColumnType("datetime").HasDefaultValueSql("getdate()");
  entity.Property(e => e.OrderTotal).HasColumnType("money")
    .HasComputedColumnSql("Store.GetOrderTotal([Id])");
});
```

Step 3: Create the Final Migration and Update the Database

1) SAVE THE StoreContext.cs FILE

2) Create a new migration using Package Manager Console:

NOTE: The following lines must be entered as one line in Package Manager Console – copying and pasting from this document doesn't work

```
dotnet ef migrations add Final -o EfStructures\Migrations -c
SpyStore_HOL.DAL.EfStructures.StoreContext
```

NOTE: The above lines must be entered as one line in Package Manager Console – copying and pasting from this document doesn't work

3) Update the database using Package Manager Console:

dotnet ef database update

Part 3: Scalar Function Mapping in EF Core

With EF Core 2, scalar SQL Server functions can be mapped to C# methods to be used in LINQ queries.

1) Open the StoreContext.cs file and ensure the following using statement is in the file:

```
using System;
```

2) Add the following static method to the StoreContext.cs file:

```
public static int GetOrderTotal(int orderId)
{
  //code in here doesn't matter
  throw new Exception();
}
```

3) Functions can be mapped using Data Annotations. To map using Data Annotations, add the DbFunction attribute:

```
[DbFunction("GetOrderTotal",Schema = "Store")]
public static int GetOrderTotal(int orderId)
{
  //code in here doesn't matter
  throw new Exception();
}
```

Part 4: Create the InvalidQuantityException

The custom InvalidQuantityException will be used later in this workshop to indicate when a user attempts to add more items into the cart than are available in stock.

Step 1: Create the Custom Exception

4) Create a new folder in the **SpyStore_HOL.DAL** project named Exceptions.

5) Add a new class to the folder named InvalidQuantityException.cs

6) Add the following using statements to the class:

```
using System;
```

7) Update the code to the following:

```
public class InvalidQuantityException : Exception
{
  public InvalidQuantityException() { }
  public InvalidQuantityException(string message) : base(message) { }
  public InvalidQuantityException(string message, Exception innerException)
    : base(message, innerException) { }
}
```

Part 5: Add the ViewModels

ViewModels are a common way to represent data from multiple tables in one class. In this lab you will create the view models for the Data Access Layer.

NOTE: The files are listed here for reference only. Since there aren't any EF Core specific concepts to explore, copy the files from Code\Complete\Lab3\SpyStore_HOL.Models\ViewModels

Step 1: Create the Base ViewModel

The base view model combines the Product and the Category classes, adding in the Display data annotations for the MVC rendering engine.

1) Create a new folder in the SpyStore_HOL.Models project named ViewModels. Create a subfolder under that named Base.
2) Add a new class to the Base folder named ProductAndCategoryBase.cs
3) Add the following using statements to the class:

```
using System.ComponentModel.DataAnnotations;
using SpyStore_HOL.Models.Entities.Base;
```

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

```
public class ProductAndCategoryBase : EntityBase
{
  public int CategoryId { get; set; }
  [Display(Name = "Category")]
  public string CategoryName { get; set; }
  public int ProductId { get; set; }
  [MaxLength(3800)]
  public string Description { get; set; }
  [MaxLength(50)]
  [Display(Name = "Model")]
  public string ModelName { get; set; }
  [Display(Name="Is Featured Product")]
  public bool IsFeatured { get; set; }
  [MaxLength(50)]
  [Display(Name = "Model Number")]
  public string ModelNumber { get; set; }
  [MaxLength(150)]
  public string ProductImage { get; set; }
  [MaxLength(150)]
  public string ProductImageLarge { get; set; }
  [MaxLength(150)]
  public string ProductImageThumb { get; set; }
  [DataType(DataType.Currency), Display(Name = "Cost")]
  public decimal UnitCost { get; set; }
  [DataType(DataType.Currency), Display(Name = "Price")]
  public decimal CurrentPrice { get; set; }
  [Display(Name="In Stock")]
  public int UnitsInStock { get; set; }
}
```

Step 2: Create the CartRecordWithProductInfo Model

1) Add a new class to the ViewModels folder named CartRecordWithProductInfo.cs
2) Update the using statements to the following:

```
using System;
```

```
using System.ComponentModel.DataAnnotations;
using SpyStore_HOL.Models.ViewModels.Base;
```

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

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

Step 3: Create the OrderDetailWithProductInfo Model

1) Add a new class to the ViewModels folder named OrderDetailWithProductInfo.cs
2) Add the following using statements to the class:

```
using System;
using System.ComponentModel.DataAnnotations;
using SpyStore_HOL.Models.ViewModels.Base;
```

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

```
public class OrderDetailWithProductInfo : ProductAndCategoryBase
{
  public int OrderId { get; set; }
  [Required]
  public int Quantity { get; set; }
  [DataType(DataType.Currency), Display(Name = "Total")]
  public decimal? LineItemTotal { get; set; }
}
```

Step 4: Create the OrderWithDetailsAndProductInfo Model

1) Add a new class to the ViewModels folder named OrderWithDetailsAndProductInfo.cs
2) Add the following using statements to the class:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using SpyStore_HOL.Models.Entities.Base;
```

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

```
public class OrderWithDetailsAndProductInfo : EntityBase {
  public int CustomerId { get; set; }
  [DataType(DataType.Currency), Display(Name = "Total")]
  public decimal? OrderTotal { get; set; }
  [DataType(DataType.Date)]
  [Display(Name = "Date Ordered")]
  public DateTime OrderDate { get; set; }
  [DataType(DataType.Date)]
  [Display(Name = "Date Shipped")]
```

```
  public DateTime ShipDate { get; set; }
  public IList<OrderDetailWithProductInfo> OrderDetails { get; set; }
}
```

Part 6: Make Internal Methods Visible to the Unit Tests

1) Add an AssemblyInfo.cs file to the SpyStore_HOL.DAL project. Clear out the default code, and replace it with this:

```
using System.Runtime.CompilerServices;
[assembly: InternalsVisibleTo("SpyStore_HOL.Tests")]
```

Summary

This lab created the user defined function and its related C# mapping, assigned it to a computed columns, and updated the database. Then you added in the view models.

Next steps

In the next part of this tutorial series, you will create the repositories.