Build an EF and ASP.NET Core 2.0 App HOL

Welcome to the Build an Entity Framework Core and ASP.NET Core 2.0 Application in a Day Hands-on Lab. This lab walks you through creating the core of the data access library.

Prior to starting this lab, you must have completed Lab 1.

All labs and files are available at https://github.com/skimedic/dotnetcore_hol.

SpyStore_HOL.DAL Project and Database Updates

Part 1: Create the InvalidQuantityException

The InvalidQuantity is used to indicate when more items than are available in stock are added to the shopping cart.

Step 1: Create the Custom Exception

1) Create a new folder in the **SpyStore_HOL.DAL** project named Exceptions.
2) Add a new class to the folder named InvalidQuantityException.cs
3) Add the following using statements to the class:

```
using System;
```

4) 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) { }
}
```

Note: If the project can't find any of the files you just created, close VS and reopen it (that's the problem with brand new software). This has largely been resolved in Visual Studio v15.5.

Part 2: Create the DbContext (SpyStore_HOL.DAL)

Step 1: Create the DbContext

1) Create a new folder in the SpyStore_HOL.DAL project named EfStructures.
2) Add a new class to the folder named StoreContext.cs.
3) Add the following using statements to the class:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Diagnostics;
using SpyStore_HOL.Models.Entities;
```

4) 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 StoreContext : DbContext
{
  public StoreContext(DbContextOptions options) : base(options) { }
}
```

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

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

6) Add the override for OnModelCreating.
Note: The OnModelCreating allows for additional shaping of the database using the FluentAPI.

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

7) 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();
});
```

8) Set the SQL Server Data type and the default value for the OrderDate and ShipDate properties of the Order table:

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

9) The LineItemTotal is a computed column. (You set the DataAnnotation in Part 1 of this lab). The specific computation is Quantity*UnitCost, and must be set using the Fluent API. The SQL Server database is set to “money” for the LineItemTotal and UnitCost fields.
NOTE: The Data Annotation on the property in the model class is unnecessary. I add it for clarity since it doesn’t cause any issues.

```
modelBuilder.Entity<OrderDetail>(entity =>
{
  entity.Property(e => e.LineItemTotal).HasColumnType("money")
    .HasComputedColumnSql("[Quantity]*[UnitCost]");
  entity.Property(e => e.UnitCost).HasColumnType("money");
});
```

10) Set the UnitCost and CurrentPrice SQL Server types to “money”:

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

11) Create a unique index for the ProductId and CustomerId fields for the ShoppingCartRecord table. 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>(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: Create 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. The DesignTimeDbContextFactory class (is found) is used by the EF Core Tools to create the DbContext.

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

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Design;
using Microsoft.EntityFrameworkCore.Diagnostics;
```

3) Make the class public and implement IDesignTimeDbContextFactory<StoreContext>:

```
public class StoreContextFactory : IDesignTimeDbContextFactory<StoreContext>
{
}
```

4) The interface has one method, CreateDbContext.
NOTE: At the time of this writing (and EF Core 2), the args argument is not used.

```
public StoreContext CreateDbContext(string[] args)
{
}
```

5) In this method, create a new instance of DbContextOptionsBuilder types for the StoreContext and create a variable to hold the connection string (update as necessary):

```
var optionsBuilder = new DbContextOptionsBuilder<StoreContext>();
var connectionString =
@"Server=(localdb)\mssqllocaldb;Database=SpyStore_HOL2;Trusted_Connection=True;
  MultipleActiveResultSets=true;";
```

6) Opt-in to using SQL Server, 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, reurn the configured StoreContext using the DbContextOptions.

```
optionsBuilder
  .UseSqlServer(connectionString,options => options.EnableRetryOnFailure())
  .ConfigureWarnings(warnings => warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
return new StoreContext(optionsBuilder.Options);
```

Part 3: Update the Database and Add the UDF

Step 1: Create and Execute the Initial Migration

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

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

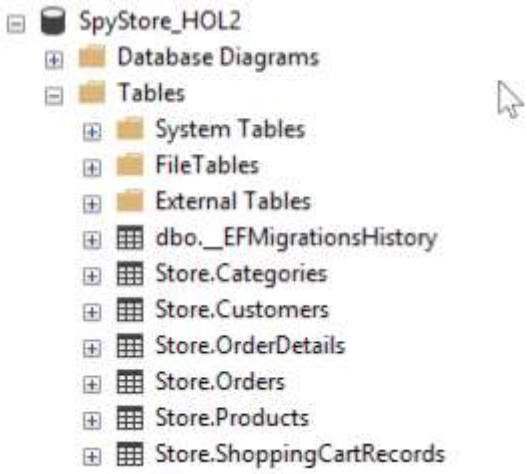
3) Create the initial migration with the following command (-o = output directory, -c = Context File):

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

4) 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 to make sure the tables were created:



Step 2: Create the Migration for the UDF and update the database

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

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

2) 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);
```

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

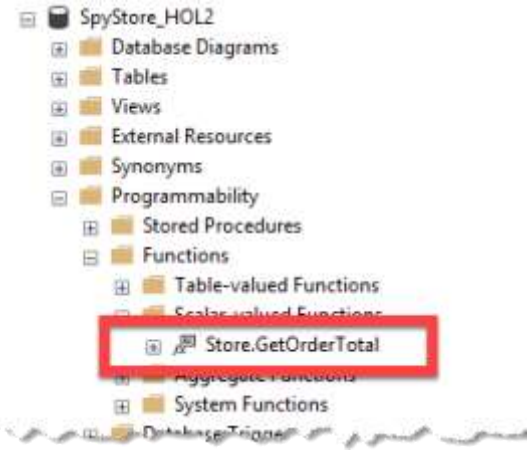
```
migrationBuilder.Sql("DROP FUNCTION [Store].[GetOrderTotal]");
```

4) SAVE THE MIGRATION FILE

5) Update the database by executing the migration:

```
dotnet ef database update
```

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



Part 4: 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:

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

3) Update the database using Package Manager Console:

dotnet ef database update

Part 5: 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) In the StoreContext.cs class, add the following static method:

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

2) Functions can be mapped using Data Annotations or the Fluent API. To map using Data Annotations, add the DbFunction attribute:

```
[DbFunction("GetOrderTotal",Schema = "Store")]
public static int GetOrderTotal(int orderId)
```

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 DbContext, updated the computed columns, and created the migrations to sync the database with the EF model.

Next steps

In the next part of this tutorial series, you will create the repositories and the data initialization code.