Build an ASP.NET Core Service, and App with Core 2.2 Two-Day Hand-On Lab

Lab 3

This lab walks you through creating a SQL Server objects (stored procs, a view, and a user defined function), computed columns on the entities, the view models, and the query types. Prior to starting this lab, you must have completed Lab 2.

Part 1: Create the SQL Server Objects

All of the SQL Server objects will be created using the EF Core migration framework. This enables a single call to the EF Core command line to update the database to the necessary state.

All of the SQL calls are added into migrations using static helper classes that leverages an instance of the `MigrationBuilder` class. The `MigrationBuilder` `Sql` method executes raw SQL against the target database.

Step 1: Create the Helper Classes

1) Create a new folder named `MigrationHelpers` under the `EfStructures` folder in the `SpyStore.Hol.Dal` project. In this folder, add three classes: `FunctionsHelper.cs`, `SprocsHelper.cs`, `ViewsHelper.cs`.
2) Make each class `public` and `static`, and add the following using statements to the top of each class:

```
using Microsoft.EntityFrameworkCore.Migrations;
```

Step 2: Create the User Defined Function Helper

1) In the `FunctionsHelper` class, add the following code to create a function that adds up the cost of each order detail record for a specific order. This will be called in the `Up` method of the migration:

```
public static void CreateOrderTotalFunction(MigrationBuilder migrationBuilder)
{
  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);
}
```

2) Add another method to drop the function. This will be called by the `Down` method of the migration.

```
public static void DropOrderTotalFunction(MigrationBuilder builder)
{
  builder.Sql("drop function [Store].[GetOrderTotal]");
}
```

Step 3: Create the Stored Proc Helper

1) In the `SprocsHelper` class, add the following code to create a stored procedure that converts the `ShoppingCartRecords` into `Orders` and `OrderDetails` records. This will be called in the `Up` method of the migration:

```
public static void CreatePurchaseSproc(MigrationBuilder migrationBuilder)
{
  var sql = @"
    CREATE PROCEDURE [Store].[PurchaseItemsInCart](@customerId INT = 0, @orderId INT OUTPUT) AS
    BEGIN
      SET NOCOUNT ON;
      INSERT INTO Store.Orders (CustomerId, OrderDate, ShipDate)
        VALUES(@customerId, GETDATE(), GETDATE());
      SET @orderId = SCOPE_IDENTITY();
      DECLARE @TranName VARCHAR(20);SELECT @TranName = 'CommitOrder';
      BEGIN TRANSACTION @TranName;
      BEGIN TRY
       INSERT INTO Store.OrderDetails (OrderId, ProductId, Quantity, UnitCost)
       SELECT @orderId, scr.ProductId, scr.Quantity, p.CurrentPrice
       FROM Store.ShoppingCartRecords scr
         INNER JOIN Store.Products p ON p.Id = scr.ProductId
       WHERE scr.CustomerId = @customerId;
       DELETE FROM Store.ShoppingCartRecords WHERE CustomerId = @customerId;
       COMMIT TRANSACTION @TranName;
      END TRY
      BEGIN CATCH
        ROLLBACK TRANSACTION @TranName;
        SET @OrderId = -1;
      END CATCH;
    END;";
  migrationBuilder.Sql(sql);
}
```

2) Add another method to drop the stored procedure. This will be called by the `Down` method of the migration.

```
public static void DropPurchaseSproc(MigrationBuilder migrationBuilder)
{
  migrationBuilder.Sql("DROP PROCEDURE [Store].[PurchaseItemsInCart]");
}
```

Step 4: Create the Views Helper

1) In the `ViewsHelper` class, add the following code to create two views, one for the `Orders` and `OrderDetails` tables (and related `Product` records), and another one for the `ShoppingCartRecords` table (and related `Product` records). These will be called in the `Up` method of the migration:

```
public static void CreateOrderDetailWithProductInfoView(MigrationBuilder builder)
{
  builder.Sql(@"
    CREATE VIEW [Store].[OrderDetailWithProductInfo]
    AS
    SELECT od.Id, od.TimeStamp, od.OrderId, od.ProductId, od.Quantity, od.UnitCost,
        od.Quantity * od.UnitCost AS LineItemTotal, p.ModelName, p.Description, p.ModelNumber,
        p.ProductImage, p.ProductImageLarge, p.ProductImageThumb, p.CategoryId, p.UnitsInStock,
        p.CurrentPrice, c.CategoryName
FROM  Store.OrderDetails od INNER JOIN Store.Orders o ON o.Id = od.OrderId
INNER JOIN Store.Products AS p ON od.ProductId = p.Id INNER JOIN
 Store.Categories AS c ON p.CategoryId = c.id");
}
public static void CreateCartRecordWithProductInfoView(MigrationBuilder builder)
{
  builder.Sql(@"
    CREATE VIEW [Store].[CartRecordWithProductInfo]
    AS
    SELECT scr.Id, scr.TimeStamp, scr.DateCreated, scr.CustomerId, scr.Quantity,
      scr.LineItemTotal,
      scr.ProductId, p.ModelName, p.Description,
      p.ModelNumber, p.ProductImage,
      p.ProductImageLarge, p.ProductImageThumb,
      p.CategoryId, p.UnitsInStock, p.CurrentPrice, c.CategoryName
    FROM Store.ShoppingCartRecords scr
      INNER JOIN Store.Products p ON p.Id = scr.ProductId
      INNER JOIN Store.Categories c ON c.Id = p.CategoryId");
}
```

2) Add another method to drop the function. This will be called by the `Down` method of the migration.

```
public static void DropOrderDetailWithProductInfoView(MigrationBuilder builder)
{
  builder.Sql("drop view [Store].[OrderDetailWithProductInfo]");
}
public static void DropCartRecordWithProductInfoView(MigrationBuilder builder)
{
  builder.Sql("drop view [Store].[CartRecordWithProductInfo]");
}
```

Step 5: Create the Migration for the SQL Server Objects

Even if nothing has changed in the model, migrations can still be created. The `Up` and `Down` methods will be empty. To execute custom SQL, that is exactly what is needed.

1) Open a command prompt or Package Manager Console in the `SpyStore.Hol.Dal` directory.
2) Create an empty migration (but do **NOT** run `dotnet ef database update`) by running the following command. Note that the output directory will be the same as previous migrations for the same derived DbContext:

```
dotnet ef migrations add TSQL -c SpyStore.Hol.Dal.EfStructures.StoreContext
```

3) Open up the new migration file (named `<timestamp>_TSQL.cs`). Note that the `Up` and `Down` methods are empty. Add the following using statement to the top of the file:

```
using SpyStore.Hol.Dal.EfStructures.MigrationHelpers;
```

4) Change the Up method to the following:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
  ViewsHelper.CreateOrderDetailWithProductInfoView(migrationBuilder);
  ViewsHelper.CreateCartRecordWithProductInfoView(migrationBuilder);
  FunctionsHelper.CreateOrderTotalFunction(migrationBuilder);
  SprocsHelper.CreatePurchaseSproc(migrationBuilder);
}
```

5) Change the Down method to the following code:

```
protected override void Down(MigrationBuilder migrationBuilder)
{
  ViewsHelper.DropOrderDetailWithProductInfoView(migrationBuilder);
  ViewsHelper.DropCartRecordWithProductInfoView(migrationBuilder);
  FunctionsHelper.DropOrderTotalFunction(migrationBuilder);
  SprocsHelper.DropPurchaseSproc(migrationBuilder);
}
```

6) SAVE THE MIGRATION FILE BEFORE RUNNING THE MIGRATION

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 Fields

Step 1: Add the OrderTotal to the Order Entity

1) Open the `OrderBase.cs` file in the `SpyStore.Hol.Models` project and add the following property:

```
[Display(Name = "Total"),DataType(DataType.Currency)]
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public decimal? OrderTotal { get; set; }
```

2) Open the `StoreContext.cs` file in the `SpyStore.Hol.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])");
});
```

NOTES:

- The `DatabaseGenerated` attribute is overruled by the Fluent API. I keep the attribute for transparency.

Step 2: Add the LineItemTotal to the OrderDetail Entity

1) Open the `OrderDetailBase.cs` file in the `SpyStore.Hol.Models` project and add the following property:

```
[DataType(DataType.Currency), Display(Name = "Total")]
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public decimal? LineItemTotal { get; set; }
```

2) Open the `StoreContext.cs` file in the `SpyStore.Hol.Dal` project, and add the following Fluent API command in the `OnModelCreating` method to the `Order` entity:

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

Step 3: Create the Final Migration and Update the Database

1) SAVE THE ALL FILES, INCLUDING OrderBase.cs, OrderDetailBase.cs, and StoreContext.cs

2) Create a new migration and update the database:

```
dotnet ef migrations add Final -c SpyStore.Hol.Dal.EfStructures.StoreContext
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:

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

NOTE:

- Functions can be mapped using Data Annotations or the Fluent API

Part 4: Add the Custom Exceptions

A common pattern in exception handling is to wrap system exceptions with custom exceptions. The SpyStore Data Access Layer uses five (5) custom exceptions with a base custome exception.

Step 1: Create the Base Exception

1) Create a new folder in the `SpyStore.Hol.Dal` project named `Exceptions`.
2) Add a new class to the folder named `SpyStoreException.cs`
3) Update the code to the following:

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

Step 2: Create the Remaining Exceptions

1) Add five more files to the directory: `SpyStoreConcurrencyException.cs`, `SpyStoreInvalidCustomerException.cs`, `SpyStoreInvalidProductException.cs`, `SpyStoreInvalidQuantityException.cs`, `SpyStoreRetryLimitExceededException.cs`.
2) Update the each of the exceptions to the following:

```
public class SpyStoreConcurrencyException : SpyStoreException
{
  public SpyStoreConcurrencyException() { }
  public SpyStoreConcurrencyException(string message) : base(message) { }
  public SpyStoreConcurrencyException(string message, Exception innerException)
    : base(message, innerException) { }
}
public class SpyStoreInvalidCustomerException : SpyStoreException
{
  public SpyStoreInvalidCustomerException() { }
  public SpyStoreInvalidCustomerException(string message) : base(message) { }
  public SpyStoreInvalidCustomerException(string message, Exception innerException)
    : base(message, innerException) { }
}
```

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

public class SpyStoreInvalidQuantityException : SpyStoreException
{
  public SpyStoreInvalidQuantityException() { }
  public SpyStoreInvalidQuantityException(string message) : base(message) { }
  public SpyStoreInvalidQuantityException(string message, Exception innerException)
    : base(message, innerException) { }
}

public class SpyStoreRetryLimitExceededException : SpyStoreException
{
  public SpyStoreRetryLimitExceededException() { }
  public SpyStoreRetryLimitExceededException(string message) : base(message) { }
  public SpyStoreRetryLimitExceededException(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. They extend the `OrderBase` and `ShoppingCartRecordBase` classes created in the last lab.

1) Create a new folder in the `SpyStore.Hol.Models` project named `ViewModels`.

Step 1: The CartRecordWithProductInfo ViewModel

This view model extends the `ShoppingCartRecordBase` class with `Product` and `Category` properties.

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

```
using System.ComponentModel.DataAnnotations;
using SpyStore.Hol.Models.Entities.Base;
```

3) Update the code to the following:

```
public class CartRecordWithProductInfo : ShoppingCartRecordBase
{
    public new int Id { get; set; }
    //Not supported at this time
    //public ProductDetails Details { get; set; }
    public string Description { get; set; }
    [Display(Name="Model Number")]
    public string ModelNumber { get; set; }
    [Display(Name = "Name")]
    public string ModelName { get; set; }
    public string ProductImage { get; set; }
    public string ProductImageLarge { get; set; }
    public string ProductImageThumb { get; set; }
    [Display(Name = "In Stock")]
    public int UnitsInStock { get; set; }
    [Display(Name = "Price"),DataType(DataType.Currency)]
    public decimal CurrentPrice { get; set; }
    public int CategoryId { get; set; }
    [Display(Name = "Category")]
    public string CategoryName { get; set; }
}
```

NOTES:

- Owned types are not yet support for query types, so the ProductDetails class can't be used.

Step 2: The CartWithCustomerInfo ViewModel

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

```
using System.Collections.Generic;
using SpyStore.Hol.Models.Entities;
```

3) Update the code to the following:

```
public class CartWithCustomerInfo
{
  public Customer Customer { get; set; }
  public IList<CartRecordWithProductInfo> CartRecords { get; set; }
    = new List<CartRecordWithProductInfo>();
}
```

Step 3: 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.ComponentModel.DataAnnotations;
using SpyStore.Hol.Models.Entities.Base;
```

3) Update the code to the following:

```
public class OrderDetailWithProductInfo : OrderDetailBase
{
  public new int Id { get; set; }
  public string Description { get; set; }
  [Display(Name = "Model Number")]
  public string ModelNumber { get; set; }
  [Display(Name = "Name")]
  public string ModelName { get; set; }
  public string ProductImage { get; set; }
  public string ProductImageLarge { get; set; }
  public string ProductImageThumb { get; set; }
  [Display(Name = "In Stock")]
  public int UnitsInStock { get; set; }
  [Display(Name = "Price"), DataType(DataType.Currency)]
  public decimal CurrentPrice { get; set; }
  public int CategoryId { get; set; }
  [Display(Name = "Category")]
  public string CategoryName { 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.Collections.Generic;
using System.Linq;
using AutoMapper;
using SpyStore.Hol.Models.Entities;
using SpyStore.Hol.Models.Entities.Base;
```

3) Update the code to the following:

```
public class OrderWithDetailsAndProductInfo : OrderBase
{
    public Customer Customer { get; set; }
    public IList<OrderDetailWithProductInfo> OrderDetails { get; set; }
}
```

4) Automapper is used to create an instance of this class from an `Order`, `Customer`, and list of `OrderWithDetailsAndProductInfo` records. Automapper must be configured with the `from` and the `to` types and any specific instructions (such as ignoring properties). Create a `static` class variable to hold the configuration, and a `static` constructor that creates the configuration. The navigation properties must be ignored, since the `OrderDetails` of the `Order` class is of a different type than the `OrderDetails` type of the viewmodel. Add the following code:

```
private static readonly MapperConfiguration _mapperCfg;
static OrderWithDetailsAndProductInfo()
{
  _mapperCfg = new MapperConfiguration(cfg =>
  {
    cfg.CreateMap<Order, OrderWithDetailsAndProductInfo>()
       .ForMember(record => record.OrderDetails, y => y.Ignore());
  });
}
```

5) Next, add a static method that will execute the mapping configuration to create the new instance of the viewmodel:

```
public static OrderWithDetailsAndProductInfo Create(Order order, Customer customer,
IEnumerable<OrderDetailWithProductInfo> details)
{
  var viewModel = _mapperCfg.CreateMapper().Map<OrderWithDetailsAndProductInfo>(order);
  viewModel.OrderDetails = details.ToList();
  viewModel.Customer = customer;
  return viewModel;
}
```

Part 6: Add the Query Types

Query types are used to map views or tables without primary keys to entities. Mapping query types is a two-step process. The first is to create a `DbQuery<T>` property on the `StoreContext`, then set the SQL to use. If the query type is to be used with a view, the mapping is added in the `OnModelCreating` method. Query types can also be loaded with `FromSql` calls.

1) Open the `StoreContext.cs` file and add the following using statement:

```
using SpyStore.Hol.Models.ViewModels;
```

2) Add the following properties to the StoreContext class:

```
public DbQuery<CartRecordWithProductInfo> CartRecordWithProductInfos { get; set; }
public DbQuery<OrderDetailWithProductInfo> OrderDetailWithProductInfos { get; set; }
```

3) In the `OnModelCreating` method, map the `DbQuery<T>` properties to the views created in the last lab:

```
modelBuilder.Query<CartRecordWithProductInfo>().ToView("CartRecordWithProductInfo", "Store");
modelBuilder.Query<OrderDetailWithProductInfo>().ToView("OrderDetailWithProductInfo", "Store");
```

Summary

This lab created the SQL Server views, function, and stored procedure. Then calculated fields were added to the `Order` and `OrderDetail` tables. The SQL Server user defined function was mapped to a C# method, the custom exceptions were created as well as the view models. Finally, the two views were mapped to view models using the query collection type.

Next steps

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