Build an EF and ASP.NET Core 2.2 App HOL

Lab 5

This lab walks you through creating the Data Initializer. Prior to starting this lab, you must have completed Lab 4.

NOTE: Copy the Assets/Lab5/Initialization folder into the SpyStore.Hol.Dal project.

Step 2: Update the Store Data Initializer

The `SampleData.cs` file (copied over from the Assets folder) provides sample data to load for testing/debugging. The `SampleDataInitializer.cs` file clears the database and reloads it.

1) Add a method to drop and create the database and a method to reseed the identity columns for all tables: **NOTE:** The `EnsureCreated` method builds the database from the entity model, but doesn't execute any migrations and blocks future migrations. The `Migrate` method creates the database (if needed) and runs all migrations for a DbContext.

```
public static void DropAndCreateDatabase(StoreContext context)
{
  context.Database.EnsureDeleted();
  context.Database.Migrate();
}
internal static void ResetIdentity(StoreContext context)
{
  var tables = new[] {"Categories", "Customers", "OrderDetails", "Orders", "Products",
                      "ShoppingCartRecords"};
  foreach (var itm in tables)
  {
    var rawSqlString = $"DBCC CHECKIDENT (\"Store.{itm}\", RESEED, 0);";
#pragma warning disable EF1000 // Possible SQL injection vulnerability.
    context.Database.ExecuteSqlCommand(rawSqlString);
#pragma warning restore EF1000 // Possible SQL injection vulnerability.
  }
}
```

2) The ClearData method clears all of the data then resets the identity seeds to -1.

```
public static void ClearData(StoreContext context)
{
  context.Database.ExecuteSqlCommand("Delete from Store.Categories");
  context.Database.ExecuteSqlCommand("Delete from Store.Customers");
  ResetIdentity(context);
}
```

3) The `SeedData` method loads the data from the `SampleData` class.

```
internal static void SeedData(StoreContext context)
{
  try
  {
    if (!context.Categories.Any())
    {
      context.Categories.AddRange(SampleData.GetCategories());
      context.SaveChanges();
    }
    if (!context.Customers.Any())
    {
      var prod1 = context.Categories.Include(c => c.Products)
                         .FirstOrDefault()?.Products.Skip(3).FirstOrDefault();
      var prod2 = context.Categories.Skip(2).Include(c => c.Products)
                         .FirstOrDefault()?.Products.Skip(2).FirstOrDefault();
      var prod3 = context.Categories.Skip(5).Include(c => c.Products)
                         .FirstOrDefault()?.Products.Skip(1).FirstOrDefault();
      var prod4 = context.Categories.Skip(2).Include(c => c.Products).FirstOrDefault()?
                         .Products.Skip(1).FirstOrDefault();
      context.Customers
               .AddRange(SampleData.GetAllCustomerRecords(
                        new List<Product> { prod1, prod2, prod3, prod4 }));
      context.SaveChanges();
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}
```

4) The entry point method is `InitializeData`. First execute the `Migrate` method of the `DatabaseFacade` to make sure all migrations have been executed. Then call `ClearData` to reset the database, and then `SeedData` to load it with test data:

```
public static void InitializeData(StoreContext context)
{
  context.Database.Migrate();
  ClearData(context);
  SeedData(context);
}
```

Summary

This lab created data initializer, completing the data access layer.

Next steps

In the next part of this tutorial series, you will start working with ASP.NET Core.