Build an EF and ASP.NET Core 2 App HOL

Welcome to the Build an Entity Framework Core and ASP.NET Core 2 Application in a Day Hands-on Lab. This lab walks you through creating the repositories and their interfaces for the data access library.

Prior to starting this lab, you must have completed Lab 2 Part 1.

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

Part 1: Creating the Repositories

Step 1: Create the Base Repository Interface

While the DbContext can be considered an implementation of the repository pattern, it's better to create specific repositories for the models. These will be leveraged by ASP.NET Core later today.

1) Create a new folder in the SpyStore_HOL.DAL project named Repos. Create a subfolder under that named Base.
2) Add a new interface to the Base folder named IRepo.cs
3) Add the following using statements to the interface:

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

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

```
public interface IRepo<T> where T : EntityBase
{
  int Count { get; }
  T Find(int id);
  IList<T> GetAll();
  int Add(T entity, bool persist = true);
  int AddRange(IEnumerable<T> entities, bool persist = true);
  int Update(T entity, bool persist = true);
  int UpdateRange(IEnumerable<T> entities, bool persist = true);
  int Delete(T entity, bool persist = true);
  int DeleteRange(IEnumerable<T> entities, bool persist = true);
  int Delete(int id, byte[] timeStamp, bool persist = true);
  int SaveChanges();
}
```

Step 2: Create the Base Repository

1) Add a new class to the Repos/Base folder named RepoBase.cs
2) Add the following using statements to the class:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Microsoft.EntityFrameworkCore.Storage;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.Models.Entities.Base;
```

3) Make the class public and abstract, generic, and Implement IDisposable and IRepo<T>:

```
public abstract class RepoBase<T> : IDisposable, IRepo<T> where T : EntityBase, new()
{
}
```

4) Add a Boolean flag for disposing of the context, a protected variable to represent the DbSet for the derived repo, and a public read only property to hold the StoreContext:

```
private readonly bool _disposeContext;
protected readonly DbSet<T> Table;
public StoreContext Context { get; }
```

5) Add a constructor that takes an instance of the StoreContext. This will set the Context property of the Repo, and set the Table property using the Context.Set<T>() method. This constructor is used by the DI container in ASP.NET Core.

```
protected RepoBase(StoreContext context)
{
  Context = context;
  Table = Context.Set<T>();
}
```

6) Add another constructor that uses the DesignTimeFactory to create a new instance of the repo. This constructor is used by the xUnit tests, and sets the dispose context flag to true. The flag doesn't get set for the other constructor, since the ASP.NET Core DI container takes care of disposing instances.

```
protected RepoBase() : this(new StoreContextFactory().CreateDbContext(new string[0]))
{
  _disposeContext = true;
}
```

7) Implement the Count, Find and GetAll methods:

```
public int Count => Table.Count();
public T Find(int id) => Table.Find(id);
public virtual IList<T> GetAll() => Table.ToList();
```

8) The Add[Range], Update[Range], and Delete[Range] methods all take an optional parameter to signal if SaveChanges should be called immediately or not.
Note: The EF method is Remove, but the Repo uses Delete for commonality.

```
public virtual int Add(T entity, bool persist = true)
{
  Table.Add(entity);
  return persist ? SaveChanges() : 0;
}
public virtual int AddRange(IEnumerable<T> entities, bool persist = true)
{
  Table.AddRange(entities);
  return persist ? SaveChanges() : 0;
}
public virtual int Update(T entity, bool persist = true)
{
  Table.Update(entity);
  return persist ? SaveChanges() : 0;
}
public virtual int UpdateRange(IEnumerable<T> entities, bool persist = true)
{
  Table.UpdateRange(entities);
  return persist ? SaveChanges() : 0;
}
public virtual int Delete(T entity, bool persist = true)
{
  Table.Remove(entity);
  return persist ? SaveChanges() : 0;
}
public virtual int DeleteRange(IEnumerable<T> entities, bool persist = true)
{
  Table.RemoveRange(entities);
  return persist ? SaveChanges() : 0;
}
```

9) Deleting and entity using EntityState and the primary key and concurrency columns can save a round trip to the database, but the entity must not be in the ChangeTracker. If it is, Remove is called on the entity in the DbSet.

```
internal T GetEntryFromChangeTracker(int? id)
{
  return Context.ChangeTracker.Entries<T>().Select((EntityEntry e) =>
(T)e.Entity).FirstOrDefault(x => x.Id == id);
}
```

```
public int Delete(int id, byte[] timeStamp, bool persist = true)
{
  var entry = GetEntryFromChangeTracker(id);
  if (entry != null)
  {
    if (timeStamp != null && entry.TimeStamp.SequenceEqual(timeStamp))
    {
      return Delete(entry, persist);
    }
    throw new Exception("Unable to delete due to concurrency violation.");
  }
  Context.Entry(new T { Id = id, TimeStamp = timeStamp }).State = EntityState.Deleted;
  return persist ? SaveChanges() : 0;
}
```

10) The RepoBase SaveChanges method encapsulates the Context.SaveChanges to allow for centralized error handling.

```
public int SaveChanges()
{
  try
  {
    return Context.SaveChanges();
  }
  catch (DbUpdateConcurrencyException ex)
  {
    //A concurrency error occurred and should be handled intelligently
    Console.WriteLine(ex);
    throw;
  }
  catch (RetryLimitExceededException ex)
  {
    //DbResiliency retry limit exceeded and should be handled intelligently
    Console.WriteLine(ex);
    throw;
  }
  catch (Exception ex)
  {
    //A general exception occurred and should be handled intelligently
    Console.WriteLine(ex);
    throw;
  }
}
```

11) The final method to add is the Dispose method:

```
public void Dispose()
{
  if (_disposeContext)
  {
    Context.Dispose();
  }
}
```

Step 3: Create the Model Specific Interfaces

1) Create a new folder under the Repos folder named Interfaces.
2) Create the following files in the Interfaces folder:

```
ICategoryRepo.cs
ICustomerRepo.cs
IOrderDetailRepo.cs
IOrderRepo.cs
IProductRepo.cs
IShoppingCartRepo.cs
```

Step 3a: Implement the ICategoryRepo Interface

1) Add the following using statements to the ICategoryRepo.cs class:

```
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.Models.Entities;
```

2) Update the code for the ICategoryRepo.cs class to the following:

```
public interface ICategoryRepo : IRepo<Category> { }
```

Step 3b: Implement the ICustomerRepo Interface

1) Add the following using statements to the ICustomerRepo.cs class:

```
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.Models.Entities;
```

2) Update the code for the ICustomerRepo.cs class to the following:

```
public interface ICustomerRepo : IRepo<Customer> { }
```

Step 3c: Implement the IOrderDetailRepo Interface

1) Add the following using statements to the IOrderDetailRepo.cs class:

```
using System.Collections.Generic;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Update the code for the IOrderDetailRepo.cs class to the following:

```
public interface IOrderDetailRepo : IRepo<OrderDetail>
{
  IEnumerable<OrderDetailWithProductInfo> GetSingleOrderWithDetails(int orderId);
}
```

Step 3d: Implement the IOrderRepo Interface

1) Add the following using statements to the IOrderRepo.cs class:

```
using System.Collections.Generic;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Update the code for the IOrderRepo.cs class to the following:

```
public interface IOrderRepo : IRepo<Order>
{
  IList<Order> GetOrderHistory(int customerId);
  OrderWithDetailsAndProductInfo GetOneWithDetails(int customerId, int orderId);
}
```

Step 3e: Implement the IProductRepo Interface

1) Add the following using statements to the IProductRepo.cs class:

```
using System.Collections.Generic;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels.Base;
```

2) Update the code for the IProductRepo.cs class to the following:

```
public interface IProductRepo : IRepo<Product>
{
  IList<ProductAndCategoryBase> Search(string searchString);
  IList<ProductAndCategoryBase> GetProductsForCategory(int id);
  IList<ProductAndCategoryBase> GetFeaturedWithCategoryName();
  ProductAndCategoryBase GetOneWithCategoryName(int id);
}
```

Step 3f: Implement the IShoppingCartRepo Interface

1) Add the following using statements to the IShoppingCartRepo.cs class:

```
using System.Collections.Generic;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Update the code for the IShoppingCartRepo.cs class to the following:

```
public interface IShoppingCartRepo : IRepo<ShoppingCartRecord>
{
  CartRecordWithProductInfo GetShoppingCartRecord(int customerId, int productId);
  IEnumerable<CartRecordWithProductInfo> GetShoppingCartRecords(int customerId);
  ShoppingCartRecord Find(int customerId, int productId);
  int Update(ShoppingCartRecord entity, int? quantityInStock, bool persist = true);
  int Add(ShoppingCartRecord entity, int? quantityInStock, bool persist = true);
}
```

Step 4: Create the Model Specific Repos

As an alternative to typing all of the following code, you can copy the fully implemented repos from the Assets folder.

1) Create the following files in the Repos folder:

```
CategoryRepo.cs
CustomerRepo.cs
OrderDetailRepo.cs
OrderRepo.cs
ProductRepo.cs
ShoppingCartRepo.cs
```

Step 4a: Implement the CategoryRepo Class

1) Add the following using statements to the CategoryRepo.cs class:

```
using System.Collections.Generic;
using System.Linq;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
```

2) Make the class public, inherit RepoBase<Category>, and implement ICategoryRepo. Update the code for the CategoryRepo.cs class to the following:

```
public class CategoryRepo : RepoBase<Category>, ICategoryRepo
{
  public CategoryRepo(){}
  public CategoryRepo(StoreContext context):base(context) { }
  public override IList<Category> GetAll() => Table.OrderBy(x => x.CategoryName).ToList();
}
```

Step 4b: Implement the CustomerRepo Class

1) Add the following using statements to the CustomerRepo.cs class:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
```

2) Make the class public, inherit RepoBase<Customer>, and implement ICustomerRepo. Update the code for the CustomerRepo.cs class to the following:

```
public class CustomerRepo : RepoBase<Customer>, ICustomerRepo
{
  public CustomerRepo() : base() { }
  public CustomerRepo(StoreContext context) : base(context) { }
  public override IList<Customer> GetAll() => Table.OrderBy(x => x.FullName).ToList();
}
```

Step 4c: Implement the OrderDetailRepo Class

1) Add the following using statements to the OrderDetailRepo.cs class:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Make the class public, inherit RepoBase<OrderDetail>, and implement IOrderDetailRepo. Update the code for the OrderDetailRepo.cs class to the following:

```
public class OrderDetailRepo : RepoBase<OrderDetail>, IOrderDetailRepo
{
  public OrderDetailRepo() { }
  public OrderDetailRepo(StoreContext context) : base(context) { }
}
```

3) Create an internal method that accepts an IQueryable<OrderDetail>, and then uses related data and a projection to return a list of ViewModels:

```
  internal IEnumerable<OrderDetailWithProductInfo> GetRecords(IQueryable<OrderDetail> query)
    => query.Include(x => x.Product).ThenInclude(p => p.Category)
      .Select(x => new OrderDetailWithProductInfo {
        OrderId = x.OrderId,
        ProductId = x.ProductId,
        Quantity = x.Quantity,
        UnitCost = x.UnitCost,
        LineItemTotal = x.LineItemTotal,
        Description = x.Product.Description,
        ModelName = x.Product.ModelName,
        ProductImage = x.Product.ProductImage,
        ProductImageLarge = x.Product.ProductImageLarge,
        ProductImageThumb = x.Product.ProductImageThumb,
        ModelNumber = x.Product.ModelNumber,
        CategoryName = x.Product.Category.CategoryName})
      .OrderBy(x => x.ModelName);
```

4) Create the public method that uses the internal method and returns the list of ViewModel instances:

```
  public IEnumerable<OrderDetailWithProductInfo> GetSingleOrderWithDetails(int orderId)
    => GetRecords(Table.Where(x => x.Order.Id == orderId));
```

Step 4d: Implement the OrderRepo Class

1) Add the following using statements to the OrderRepo.cs class:

```
using System.Collections.Generic;
using System.Linq;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Make the class public, inherit RepoBase<Order>, and implement IOrderRepo. This repo also needs an instance of an IOrderDetail repo, and a private variable to hold that instance. Update the class and add the constructors:

```
public class OrderRepo : RepoBase<Order>, IOrderRepo
{
  private readonly IOrderDetailRepo _orderDetailRepo;
  public OrderRepo(IOrderDetailRepo orderDetailRepo)
  {
    _orderDetailRepo = orderDetailRepo;
  }
  public OrderRepo(StoreContext context, IOrderDetailRepo orderDetailRepo) : base(context)
  {
    _orderDetailRepo = orderDetailRepo;
  }
}
```

3) Create a method to return the orders for a customer:

```
public IList<Order> GetOrderHistory(int customerId)
  => Table.Where(x => x.CustomerId == customerId).ToList();
```

4) Create a method to return a OrderWithDetailsAndProductInfo ViewModel for a Customer's order:

```
  public OrderWithDetailsAndProductInfo GetOneWithDetails(int customerId, int orderId)
    => Table
      .Where(x => x.CustomerId == customerId && x.Id == orderId)
      .Select(x => new OrderWithDetailsAndProductInfo
      {
        Id = x.Id,
        CustomerId = customerId,
        OrderDate = x.OrderDate,
        ShipDate = x.ShipDate,
        OrderDetails = _orderDetailRepo.GetSingleOrderWithDetails(orderId).ToList()
      })
      .FirstOrDefault();
```

Step 4e: Implement the ProductRepo Class

1) Add the following using statements to the ProductRepo.cs class:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels.Base;
```

2) Make the class public, inherit RepoBase<Product>, and implement IProductRepo. Add the constructors for the class and the GetAll method:

```
public class ProductRepo : RepoBase<Product>, IProductRepo
{
  public ProductRepo() { }
  public ProductRepo(StoreContext context) : base(context) { }

  public override IList<Product> GetAll() => Table.OrderBy(x => x.ModelName).ToList();
}
```

3) And an internal method that takes a Product and a Category and returns the ProductAndCategoryBase View Model:

```
internal ProductAndCategoryBase GetRecord(Product p, Category c) => new ProductAndCategoryBase()
{
  CategoryName = c.CategoryName,
  CategoryId = p.CategoryId,
  CurrentPrice = p.CurrentPrice,
  Description = p.Description,
  IsFeatured = p.IsFeatured,
  Id = p.Id,
  ModelName = p.ModelName,
  ModelNumber = p.ModelNumber,
  ProductImage = p.ProductImage,
  ProductImageLarge = p.ProductImageLarge,
  ProductImageThumb = p.ProductImageThumb,
  TimeStamp = p.TimeStamp,
  UnitCost = p.UnitCost,
  UnitsInStock = p.UnitsInStock
};
```

4) Add a method to return a list of ProductAndCategoryBase View Models by Category ID:

```
public IList<ProductAndCategoryBase> GetProductsForCategory(int id) => Table
  .Where(p => p.CategoryId == id).Include(p => p.Category).OrderBy(x => x.ModelName)
  .Select(item => GetRecord(item, item.Category))
  .ToList();
```

5) Add a method to return a list of ProductAndCategoryBase View Models for featured products:

```
public IList<ProductAndCategoryBase> GetFeaturedWithCategoryName() => Table
  .Where(p => p.IsFeatured).Include(p => p.Category). OrderBy(x => x.ModelName)
  .Select(item => GetRecord(item, item.Category))
  .ToList();
```

6) Add a method to return a single ProductAndCategoryBase for a Product Id:

```
public ProductAndCategoryBase GetOneWithCategoryName(int id) => Table
  .Where(p => p.Id == id).Include(p => p.Category)
  .Select(item => GetRecord(item, item.Category))
  .FirstOrDefault();
```

7) Add a method for the Search. This method uses the new Like operator in the LINQ query:

```
public IList<ProductAndCategoryBase> Search(string searchString) => Table
  .Where(p => EF.Functions.Like(p.Description, $"%{searchString}%")
    || EF.Functions.Like(p.ModelName, $"%{searchString}%"))
  .Include(p => p.Category).OrderBy(x => x.ModelName)
  .Select(item => GetRecord(item, item.Category))
  .ToList();
```

Step 4f: Implement the ShoppingCartRepo Class

The ShoppingCartRepo does the majority of the work for the sample application.

1) Add the following using statements to the ShoppingCartRepo.cs class:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore_HOL.DAL.EfStructures;
using SpyStore_HOL.DAL.Exceptions;
using SpyStore_HOL.DAL.Repos.Base;
using SpyStore_HOL.DAL.Repos.Interfaces;
using SpyStore_HOL.Models.Entities;
using SpyStore_HOL.Models.ViewModels;
```

2) Make the class public, inherit RepoBase<ShoppingCartRecord>, and implement IShoppingCartRepo. This repo needs an instance of the IProductRepo. Add the constructors for the class and the private variable for the IProductRepo:

```
public class ShoppingCartRepo : RepoBase<ShoppingCartRecord>, IShoppingCartRepo
{
  private readonly IProductRepo _productRepo;
  public ShoppingCartRepo(IProductRepo productRepo) : base()
  {
    _productRepo = productRepo;
 }
  public ShoppingCartRepo(StoreContext context, IProductRepo productRepo) : base(context)
  {
    _productRepo = productRepo;
  }
}
```

3) Add the override for GetAll that sorts by DateCreate descending:

```
public override IList<ShoppingCartRecord> GetAll()
  => Table.OrderByDescending(x => x.DateCreated).ToList();
```

4) Add a find method that searches by ProductId and CustomerId:

```
public ShoppingCartRecord Find(int customerId, int productId)
  => Table.FirstOrDefault(x => x.CustomerId == customerId && x.ProductId == productId);
```

5) Before updating the cart record, the quantity left in stock is checked to make sure the order can be fulfilled. If the quantity in stock is not passed into the method, it is looked up using the product repo. Add the following methods:

```
public override int Update(ShoppingCartRecord entity,bool persist = true)
  => Update(entity, _productRepo.Find(entity.ProductId)?.UnitsInStock,persist);
```

```
public int Update(ShoppingCartRecord entity, int? quantityInStock, bool persist = true)
{
  if (entity.Quantity <= 0)
  {
    return Delete(entity, persist);
  }
  if (entity.Quantity > quantityInStock)
  {
    throw new InvalidQuantityException("Can't add more product than available in stock");
  }
  return base.Update(entity,persist);
}
```

6) Before adding a new item to the cart, the quantity left in stock is checked to make sure the order can be fulfilled. If the quantity in stock is not passed into the method, it is looked up using the product repo. Add the following methods:

```
public override int Add(ShoppingCartRecord entity, bool persist = true)
  => Add(entity, _productRepo.Find(entity.ProductId)?.UnitsInStock, persist);

public int Add(ShoppingCartRecord entity, int? quantityInStock, bool persist = true)
{
  var item = Find(entity.CustomerId, entity.ProductId);
  if (item == null)
  {
    if (quantityInStock != null && entity.Quantity > quantityInStock.Value)
    {
      throw new InvalidQuantityException("Can't add more product than available in stock");
    }
    return base.Add(entity, persist);
  }
  item.Quantity += entity.Quantity;
  return item.Quantity <= 0 ? Delete(item,persist) : Update(item,quantityInStock,persist);
}
```

7) Create an internal method to build the CartRecordWithProductInfo ViewModel from the ShoppingCartRecord, Product, and Category.

```
internal CartRecordWithProductInfo GetRecord(
  int customerId, ShoppingCartRecord scr, Product p, Category c)
  => new CartRecordWithProductInfo
{
  Id = scr.Id,
  DateCreated = scr.DateCreated,
  CustomerId = customerId,
  Quantity = scr.Quantity,
  ProductId = scr.ProductId,
  Description = p.Description,
  ModelName = p.ModelName,
  ModelNumber = p.ModelNumber,
  ProductImage = p.ProductImage,
  ProductImageLarge = p.ProductImageLarge,
  ProductImageThumb = p.ProductImageThumb,
  CurrentPrice = p.CurrentPrice,
  UnitsInStock = p.UnitsInStock,
  CategoryName = c.CategoryName,
  LineItemTotal = scr.Quantity * p.CurrentPrice,
  TimeStamp = scr.TimeStamp
};
```

8) Add a method that gets a single CartRecordWithProductInfo ViewModel using the internal helper method:

```
public CartRecordWithProductInfo GetShoppingCartRecord(int customerId, int productId) => Table
  .Where(x => x.CustomerId == customerId && x.ProductId == productId)
  .Include(x => x.Product).ThenInclude(p => p.Category)
  .Select(x => GetRecord(customerId, x, x.Product, x.Product.Category))
  .FirstOrDefault();
```

1) Add a method that gets a list of CartRecordWithProductInfo ViewModels using the internal helper method:

```
public IEnumerable<CartRecordWithProductInfo> GetShoppingCartRecords(int customerId) => Table
  .Where(x => x.CustomerId == customerId)
  .Include(x => x.Product).ThenInclude(p => p.Category).OrderBy(x => x.Product.ModelName)
  .Select(x => GetRecord(customerId, x, x.Product, x.Product.Category));
```

Part 2: Creating the Data Initializer

Step 1: Create the Sample Data provider

1) Create a new folder named Initialization under the EfStructures folder in the SpyStore_HOL.DAL project
2) Copy the StoreSampleData.cs file from the Assets folder into the Initialization folder.

Step 2: Create the Store Data Initializer

1) Create a new file named StoreDataInitializer.cs.
2) Add the following using statements to the class:

```
using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
```

3) Change the class to public and static:

```
public static class StoreDataInitializer
{
}
```

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

```
public static void ClearData(StoreContext context)
{
  ExecuteDeleteSql(context, "Categories");
  ExecuteDeleteSql(context, "Customers");
  ResetIdentity(context);
}
internal static void ExecuteDeleteSql(StoreContext context, string tableName)
{
  //With 2.0, must separate string interpolation if not passing in params
  var rawSqlString = $"Delete from Store.{tableName}";
  context.Database.ExecuteSqlCommand(rawSqlString);
}
internal static void ResetIdentity(StoreContext context)
{
  var tables = new[] {"Categories","Customers",
    "OrderDetails","Orders","Products","ShoppingCartRecords"};
  foreach (var itm in tables)
  {
    //With 2.0, must separate string interpolation if not passing in params
    var rawSqlString = $"DBCC CHECKIDENT (\"Store.{itm}\", RESEED, -1);";
    context.Database.ExecuteSqlCommand(rawSqlString);
  }
}
```

5) 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);
}
```

6) The SeedData method loads the data from the StoreSampleData class.

```
internal static void SeedData(StoreContext context)
{
  try
  {
    if (!context.Categories.Any())
    {
      context.Categories.AddRange(StoreSampleData.GetCategories());
      context.SaveChanges();
    }
    if (!context.Products.Any())
    {
      context.Products.AddRange(StoreSampleData.GetProducts(context.Categories.ToList()));
      context.SaveChanges();
    }
    if (!context.Customers.Any())
    {
      context.Customers.AddRange(StoreSampleData.GetAllCustomerRecords(context));
      context.SaveChanges();
    }
    var customer = context.Customers.FirstOrDefault();
    if (!context.Orders.Any())
    {
      context.Orders.AddRange(StoreSampleData.GetOrders(customer, context));
      context.SaveChanges();
    }
    if (!context.OrderDetails.Any())
    {
      var order = context.Orders.First();
      context.OrderDetails.AddRange(StoreSampleData.GetOrderDetails(order,context));
      context.SaveChanges();
    }
    if (!context.ShoppingCartRecords.Any())
    {
      context.ShoppingCartRecords.AddRange(StoreSampleData.GetCart(customer, context));
      context.SaveChanges();
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex);
  }
}
```

Summary

The lab created all of the repositories and their interfaces and the data initializers, completing the data access layer.

Next steps

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