Build an EF and ASP.NET Core 2.2 App HOL

Lab 4

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 3.

NOTE: Copy the Repos directory (and all of its files and subdirectories) from the Assets/Lab4/Repos folder.

Part 1: Update the Repositories

While the `DbContext` can be considered an implementation of the repository pattern, it's better to create specific repositories for the entities. These repos will be added into the ASP.NET Core Dependency Injection container later today.

Step 1: Update the Base Repository

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

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

2) Add a constructor that takes an instance of the `StoreContext` that sets the `Context` and `Table` properties. A `DbSet<T>` property can be referenced using the `Context.Set<T>()` method.
NOTE: This constructor is used by the DI container in ASP.NET Core. The ASP.NET Core DI container manages lifetime, so set the flag for context disposal to false.

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

3) Add another constructor that takes in `DbContextOptions`, calls the previous constructor while creating a new StoreContext using the options. Since this is not used by DI, set the disposal flag to true.

```
protected RepoBase(DbContextOptions<StoreContext> options) : this(new StoreContext(options))
{
  _disposeContext = true;
}
```

4) Implement the `Dispose` method:

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

5) Implement the three `Find` variations show using the built-in `Find` method, the `AsNoTracking` method, as well as the `IgnoreQueryFilters` method:

```
public T Find(int? id) => Table.Find(id);
public T FindAsNoTracking(int id)
  => Table.Where(x => x.Id == id).AsNoTracking().FirstOrDefault();
public T FindIgnoreQueryFilters(int id)
  => Table.IgnoreQueryFilters().FirstOrDefault(x => x.Id == id);
```

6) `TheGetAll` methods are virtual, allowing for the derived repos to override them. The first just returns the records in database order, the second uses LINQ expressions to return the records in a specific order.

```
public virtual IEnumerable<T> GetAll() => Table;
public virtual IEnumerable<T> GetAll(Expression<Func<T, object>> orderBy)
  => Table.OrderBy(orderBy);
```

7) The `GetRange` demonstrates chunking of data using `Skip` and `Take`:

```
public IEnumerable<T> GetRange(IQueryable<T> query, int skip, int take)
  => query.Skip(skip).Take(take);
```

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 name to delete a record is `Remove`, since it is technically just removing the instance from the `DbSet<T>`. `Delete` doesn't happen until `SaveChanges` is called. I use the name `Delete` in my repos because it is clearer.

```
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) 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
  {
    throw new SpyStoreConcurrencyException("A concurrency error happened.", ex);
  }
  catch (RetryLimitExceededException ex)  //DbResiliency retry limit exceeded
  {
    throw new SpyStoreRetryLimitExceededException("There is a problem with you connection.", ex);
  }
  catch (DbUpdateException ex)
  {
    if (ex.InnerException is SqlException sqlException)
    {
      if (sqlException.Message
        .Contains("FOREIGN KEY constraint", StringComparison.OrdinalIgnoreCase))
      {
        if (sqlException.Message
           .Contains("table \"Store.Products\", column 'Id'", StringComparison.OrdinalIgnoreCase))
        {
          throw new SpyStoreInvalidProductException($"Invalid Product Id\r\n{ex.Message}", ex);
        }
        if (sqlException.Message
          .Contains("table \"Store.Customers\", column 'Id'", StringComparison.OrdinalIgnoreCase))
        {
          throw new SpyStoreInvalidCustomerException($"Invalid Customer Id\r\n{ex.Message}", ex);
        }
      }
    }
    throw new SpyStoreException("An error occurred updating the database", ex);
  }
  catch (Exception ex)
  {
    throw new SpyStoreException("An error occurred updating the database", ex);
  }
}
```

Part 2: Add the Entity Specific Interfaces

There is an interface and repo for each model that uses the base repository for the common functionality. Each specific repo extends or overwrites that base functionality as needed. The prebuilt classes are in the Interfaces folder under the Repos folder. Only some of the interfaces extend the base interface. Those will be updated now.

Step 1: Update the IOrderDetailRepo Interface

1) Add one method as follows:

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

Step 2: Update the IOrderRepo Interface

1) Add methods to get the order history as well as get a single order with details as follows:

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

Step 3: Define the IProductRepo Interface

1) Add methods for `Search`, getting all `Products` for a `Category`, get featured products with `CategoryName`, and get one `Product` with `CategoryName`, as follows:

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

Step 4: Define the IShoppingCartRepo Interface

1) Add the seven methods needed as follows:

```
public interface IShoppingCartRepo : IRepo<ShoppingCartRecord>
{
  CartRecordWithProductInfo GetShoppingCartRecord(int id);
  IEnumerable<CartRecordWithProductInfo> GetShoppingCartRecords(int customerId);
  CartWithCustomerInfo GetShoppingCartRecordsWithCustomer(int customerId);
  ShoppingCartRecord GetBy(int productId);
  int Update(ShoppingCartRecord entity, Product product, bool persist = true);
  int Add(ShoppingCartRecord entity, Product product, bool persist = true);
  int Purchase(int customerId);
}
```

Part 3: Update the Entity Specific Repos

Step 1: Update the CategoryRepo Class

1) Add an override for the `GetAll` that takes the expression for selecting `CategoryName`, as follows:

```
public class CategoryRepo : RepoBase<Category>, ICategoryRepo
{
  public CategoryRepo(StoreContext context) : base(context) { }
  internal CategoryRepo(DbContextOptions<StoreContext> options) : base(options) { }
  public override IEnumerable<Category> GetAll() => base.GetAll(x => x.CategoryName);
}
```

Step 2: Update the CustomerRepo Class

1) Add an override for the `GetAll` that takes the expression for selecting `FullName`, as follows:

```
public class CustomerRepo : RepoBase<Customer>, ICustomerRepo
{
  public CustomerRepo(StoreContext context) : base(context) { }
  internal CustomerRepo(DbContextOptions<StoreContext> options) : base(options) { }
  public override IEnumerable<Customer> GetAll() => base.GetAll(x => x.FullName).ToList();
}
```

Step 3: Update the OrderDetailRepo Class

2) Add the single method in the interface, as follows:

```
public class OrderDetailRepo : RepoBase<OrderDetail>, IOrderDetailRepo
{
  public OrderDetailRepo(StoreContext context) : base(context) { }
  internal OrderDetailRepo(DbContextOptions<StoreContext> options) : base(options) { }
  public IEnumerable<OrderDetailWithProductInfo> GetOrderDetailsWithProductInfoForOrder(
      int orderId)
    => Context
              .OrderDetailWithProductInfos
              .Where(x => x.OrderId == orderId)
              .OrderBy(x => x.ModelName);
}
```

Step 4: Update the OrderRepo Class

1) This repo also needs an instance of an `IOrderDetail` repo, and a private variable to hold that instance. Update the constructors with the addition of the `IOrderDetail` repo. Add an override for the Dispose method to handle this additional repo:

```
private readonly IOrderDetailRepo _orderDetailRepo;
public OrderRepo(StoreContext context, IOrderDetailRepo orderDetailRepo) : base(context)
{
  _orderDetailRepo = orderDetailRepo;
}
internal OrderRepo(DbContextOptions<StoreContext> options) : base(options)
{
  _orderDetailRepo = new OrderDetailRepo(Context);
}
public override void Dispose()
{
  _orderDetailRepo.Dispose();
  base.Dispose();
}
```

2) This entity has a global query filter, so `CustomerId` does not need to be included in the queries. Add the two implementations of the interface methods, as follows:

```
public IList<Order> GetOrderHistory() => GetAll(x => x.OrderDate).ToList();
public OrderWithDetailsAndProductInfo GetOneWithDetails(int orderId)
{
  var order = Table.IgnoreQueryFilters().Include(x=>x.CustomerNavigation)
    .FirstOrDefault(x => x.Id == orderId);
  if (order == null)
  {
    return null;
  }
  var orderDetailsWithProductInfoForOrder =
    _orderDetailRepo.GetOrderDetailsWithProductInfoForOrder(order.Id);
  var orderWithDetailsAndProductInfo = OrderWithDetailsAndProductInfo
       .Create(order,order.CustomerNavigation, orderDetailsWithProductInfoForOrder);
  return orderWithDetailsAndProductInfo;
}
```

Step 5: Update the ProductRepo Class

1) Add the interface implementations, as shown below. The `Search` method uses the SQL Server `Like` operator:

```
public override IEnumerable<Product> GetAll() => base.GetAll(x => x.Details.ModelName);
public IList<Product> Search(string searchString)
  => Table.Where(p => EF.Functions.Like(p.Details.Description, $"%{searchString}%")
                   || EF.Functions.Like(p.Details.ModelName, $"%{searchString}%"))
          .Include(p => p.CategoryNavigation)
          .OrderBy(x => x.Details.ModelName)
          .ToList();
```

```
public IList<Product> GetProductsForCategory(int id)
  => Table.Where(p => p.CategoryId == id)
          .Include(p => p.CategoryNavigation)
          .OrderBy(x => x.Details.ModelName)
          .ToList();
public IList<Product> GetFeaturedWithCategoryName()
  => Table.Where(p => p.IsFeatured)
          .Include(p => p.CategoryNavigation)
          .OrderBy(x => x.Details.ModelName)
          .ToList();
public Product GetOneWithCategoryName(int id)
  => Table.Where(p => p.Id == id)
          .Include(p => p.CategoryNavigation)
          .FirstOrDefault();
}
```

Step 6: Update the ShoppingCartRepo Class

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

1) This repo needs an instance of the `IProductRepo` and the `ICustomerRepo`, so add that to the standard constructors. Also override the Dispose method to handle the additional repos.

```
private readonly IProductRepo _productRepo;
private readonly ICustomerRepo _customerRepo;
public ShoppingCartRepo(StoreContext context,
  IProductRepo productRepo, ICustomerRepo customerRepo) : base(context)
{
    _productRepo = productRepo;
    _customerRepo = customerRepo;
}
internal ShoppingCartRepo(DbContextOptions<StoreContext> options)
  : base(new StoreContext(options))
{
  _productRepo = new ProductRepo(Context);
  _customerRepo = new CustomerRepo(Context);
}
public override void Dispose()
{
  _productRepo.Dispose();
  _customerRepo.Dispose();
  base.Dispose();
}
```

2) This entity has a global query filter, so the queries against `DbSet<ShoppingCartRecord>` do not need a `CustomerId`.

```
public override IEnumerable<ShoppingCartRecord> GetAll()
  => base.GetAll(x => x.DateCreated).ToList();
public ShoppingCartRecord GetBy(int productId)
  => Table.FirstOrDefault(x => x.ProductId == productId);
```

3) The queries against the query types do not have a filter, so for those methods the `CustomerId` is required unless the primary key is specified.

```
public CartRecordWithProductInfo GetShoppingCartRecord(int id)
  => Context.CartRecordWithProductInfos.FirstOrDefault(x => x.Id == id);
public IEnumerable<CartRecordWithProductInfo> GetShoppingCartRecords(int customerId)
  => Context
            .CartRecordWithProductInfos
            .Where(x => x.CustomerId == customerId)
            .OrderBy(x => x.ModelName);
public CartWithCustomerInfo GetShoppingCartRecordsWithCustomer(int customerId)
  => new CartWithCustomerInfo()
         {
           CartRecords = GetShoppingCartRecords(customerId).ToList(),
           Customer = _customerRepo.Find(customerId)
         };
```

4) The update checks the resulting quantity. If zero or less, the record is deleted.

```
public override int Update(ShoppingCartRecord entity, bool persist = true)
{
  var product = _productRepo.FindAsNoTracking(entity.ProductId);
  if (product == null)
  {
    throw new SpyStoreInvalidProductException("Unable to locate product");
  }
  return Update(entity, product, persist);
}
public int Update(ShoppingCartRecord entity, Product product, bool persist = true)
{
  if (entity.Quantity <= 0)
  {
    return Delete(entity, persist);
  }
  if (entity.Quantity > product.UnitsInStock)
  {
    throw new SpyStoreInvalidQuantityException("Can't add more product than available in stock");
  }
  var dbRecord = Find(entity.Id);
  if (entity.TimeStamp != null && dbRecord.TimeStamp.SequenceEqual(entity.TimeStamp))
  {
    dbRecord.Quantity = entity.Quantity;
    dbRecord.LineItemTotal = entity.Quantity * product.CurrentPrice;
    return base.Update(dbRecord, persist);
  }
  throw new SpyStoreConcurrencyException("Record was changed since it was loaded");
}
public override int UpdateRange(IEnumerable<ShoppingCartRecord> entities, bool persist = true)
{
  int counter = 0;
  foreach (var item in entities)
  {
    var product = _productRepo.FindAsNoTracking(item.ProductId);
    counter += Update(item, product, false);
  }
  return persist ? SaveChanges() : counter;
}
```

5) The Add methods check to make sure the product isn't already in the cart. If it is, it increases the quantity. If it isn't, it creates a new record.

```
public override int Add(ShoppingCartRecord entity, bool persist = true)
{
  var product = _productRepo.FindAsNoTracking(entity.ProductId);
  if (product == null)
  {
    throw new SpyStoreInvalidProductException("Unable to locate the product");
  }
  return Add(entity, product, persist);
}
public int Add(ShoppingCartRecord entity, Product product, bool persist = true)
{
  var item = GetBy(entity.ProductId);
  if (item == null)
  {
    if (entity.Quantity > product.UnitsInStock)
    {
      throw new SpyStoreInvalidQuantityException(
        "Can't add more product than available in stock");
    }
    entity.LineItemTotal = entity.Quantity * product.CurrentPrice;
    return base.Add(entity, persist);
  }
  item.Quantity += entity.Quantity;
  return item.Quantity <= 0 ? Delete(item, persist) : Update(item, product, persist);
}
public override int AddRange(IEnumerable<ShoppingCartRecord> entities, bool persist = true)
{
  int counter = 0;
  foreach (var item in entities)
  {
    var product = _productRepo.FindAsNoTracking(item.ProductId);
    counter += Add(item, product, false);
  }
  return persist ? SaveChanges() : counter;
}
```

6) The Purchase method calls the stored procedure to convert the shopping cart records to an order and order details.

```
public int Purchase(int customerId)
{
  var customerIdParam = new SqlParameter("@customerId", SqlDbType.Int)
  {
    Direction = ParameterDirection.Input,
    Value = customerId
  };
  var orderIdParam = new SqlParameter("@orderId", SqlDbType.Int)
  {
    Direction = ParameterDirection.Output
  };
  try
  {
    Context
     .Database
     .ExecuteSqlCommand("EXEC [Store].[PurchaseItemsInCart] @customerId, @orderid out",
        customerIdParam, orderIdParam);
  }
  catch (Exception ex)
  {
    return -1;
  }
  return (int)orderIdParam.Value;
}
```

Summary

The lab created all of the repositories and their interfaces.

Next steps

In the next part of this tutorial series, you will create a data initializer.