Cello How-To Guide

Entity Framework Data Access Layer in CelloSaaS

techcello

1 Introduction

The following documentation provides an overview of consuming the Entity Framework with CelloSaaS. This is not an extensive document that delves deep into the details of Entity framework.

1.1 Pre- requisites

This how-to guide assumes that the developer possesses a prior working experience with entity framework and is comfortable with the concepts detailed here.

1.2 Entity Framework

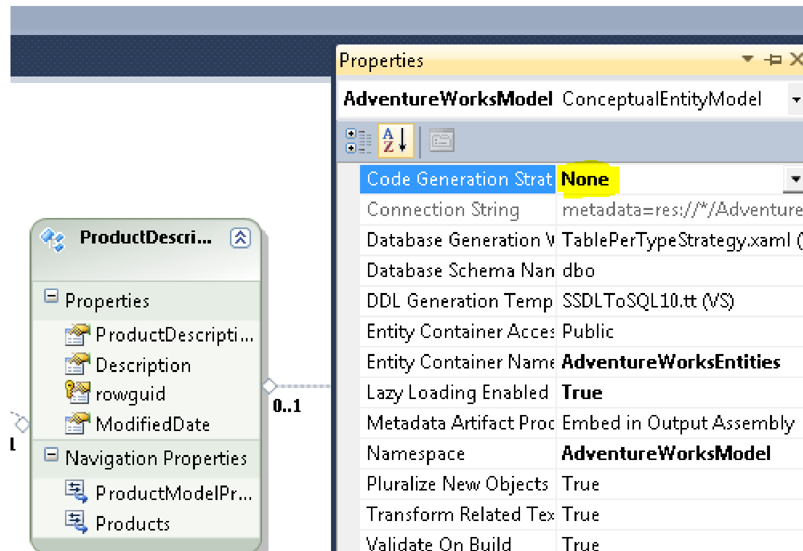
The following steps illustrate the process of building a sample application using entity framework. This sample uses the AdventureWorks database from Microsoft. The only difference with respect to the Entity Framework is in the Model and Data access layers when compared to building an application using CelloSaaS with ADO.Net.

I. Model

1. Create the required POCO classes for the product entity as shown below, or in case of using any existing POCO classes, the classes should derive from the BaseEntity and then provide the EntityIdentifier, EntityDescriptor attributes.

```
[EntityIdentifier(Name = "Product")]
[EntityDescriptor(DisplayColumnName = "ProductID", Features = "ManageProduct", IsExtensible = false,
    PrimaryKeyName = "rowguid", Privileges = "Add_Product,View_Product,Edit_Product,Delete_Product,Search_Product",
    SchemaTableConnectionStringName = CelloSaaS.Library.DAL.Constants.ApplicationConnectionString, SchemaTableName = "Product")]
[DataContract]
[Serializable]
public partial class ProductDetails : TenantTransactionalEntity
{
```

2. Generate the .edmx file for the product entity using the wizard in Visual Studio.
3. Set the namespace in the wizard to be same as that of the POCO class namespace.
4. Set the code generation strategy to "None" as shown below,



5. Now, create the context and entities for the EDMX separately by right clicking on the EDMX canvas and then choosing "Add A Code Generation Item". The purpose for doing this is to ensure that the changes if any in the database do not directly affect the above POCO classes.
6. Now, derive the Entities from the "CelloEntityObject" as shown below,

```
[EdmEntityTypeAttribute(NamespaceName = "AdventureWorksModel", Name = "Address")]
[Serializable()]
[DataContractAttribute(IsReference = true)]
public partial class Address : CelloEntityObject
{
    Factory Method
    Primitive Properties

    Navigation Properties
}
```

7. Also, the POCO classes defined above will be representing the business objects and not the database column names as such.
8. POCO can also be directly used instead of entity object if you use POCO generation with EF. In this case business object and data object need not be different and you can use POCO objects directly.
9. Also, the LazyLoading and ProxyCreation should be disabled to facilitate the use in web services [WCF]
10. In the model's application configuration file, the connection string will be generated by the wizard; this connection string name should be made available to the WebApp in the sql.config file. Hence, this can be copied and used in the WebApp sql.config file.

```
App.Config
1  <?xml version="1.0" encoding="utf-8"?>
2  <configuration>
3    <connectionStrings>
4      <add name="AdventureWorksEntities" connectionString="metadata=res://*/AdventureWorks.csdl|res:
5    </connectionStrings>
6  </configuration>
```

```
Sql.config
1  <connectionStrings>
2    <add name="ApplicationConnectionString" connectionString="Data Source=.\DbServer;Initial Catalog
3    <add name="CelloSaaSConnectionString" connectionString="Data Source=.\DBServer;Initial Catalog=C
4    <add name="UserConnectionString" connectionString="Data Source=.\dbserver;Initial Catalog=CelloE
5    <add name="WorkFlowConnectionString" connectionString="Data Source=.\dbserver;enlist=false;Initi
6    <add name="ProductAnalyticsConnectionString" connectionString="Data Source=.\dbserver;Initial Ca
7    <add name="AdventureWorksEntities" connectionString="Data Source=.\DbServer;Initial Catalog=Adve
```

11. Create an appropriate search condition that derives from the SearchCondition class from CelloSaaS.Model assembly and CelloSaaS.Model Namespace.

```
namespace EFSample.Model
{
    /// <summary> ...
    [Serializable]
    [DataContract]
    public class ProductSearchCondition : SearchCondition
    {
        /// <summary> ...
        [DataMember]
        public int ProductID...

        /// <summary> ...
        [DataMember]
        public string Name...
```

```
[DataMember]
public virtual ProductDescription ProductDescription { get; set; }
[DataMember]
public virtual ProductCategory ProductCategory { get; set; }
[DataMember]
public virtual ProductModel ProductModel { get; set; }
[DataMember]
public virtual ICollection<SalesOrderDetail> SalesOrderDetails { get; set; }
```

12. Also, create the necessary exception in this case, the ProductException that derives from the CelloSaaS.Library.Exceptions.BaseException from CelloSaaS.Library assembly and CelloSaaS.Library.Exceptions namespace.

```
namespace EFSample.Model
{
    /// <summary> ...
    [Serializable]
    public class ProductException : BaseException
    {
        /// <summary> ...
        protected ProductException(System.Runtime.Serialization.SerializationInfo info,
                                              System.Runtime.Serialization.StreamingContext context)
            : base(info, context)...

        /// <summary> ...
        public ProductException(string message, Exception innerException)
            : base(message, innerException)...

        /// <summary> ...
        public ProductException(string message)
            : base(message)...

        /// <summary> ...
        public ProductException()...
    }
```

Note:

Microsoft standard for defining custom exception is given here.

II. DAL

1. The process of creating the DAL is similar to that of ADO.Net; the interfaces should implement IEntityDAL<T>, where T will be any BaseEntity. In this case, the DAL interface will be IProductDAL

```
namespace EFSample.DAL
{
    /// <summary> ...
    public interface IProductDAL : IEntityDAL<Product>
    {
        /// <summary> ...
        int GetProductCount(DataSearchRequest dataSearchRequest);
    }
}
```

III SqlDAL

1. Create a SqlDAL class that derives from EntityDAL<T> and implements the IProductDAL. In this case it will be ProductDAL
2. Create a reference to System.Data.Entity dll for this project

```
namespace EFSample.SqlDAL
{
    /// <summary> ...
    public class ProductDAL : EntityDAL<Product>, IProductDAL
    {
        /// <summary> ...
        protected override string DoCreate(DataCreateRequest dataCreateRequest)
        {
            Product product = (Product)dataCreateRequest.Entity;
            using (AdventureWorksEntities context = new AdventureWorksEntities(_getConnection))
            {
                product.rowguid = Guid.NewGuid();
                product.ProductModel.rowguid = Guid.NewGuid();
                product.ProductDescription.rowguid = Guid.NewGuid();

                product.Identifier = product.rowguid.ToString();
                context.Products.AddObject(product);
                context.SaveChanges();
            }
            return product.Identifier;
```

3. Now, create a private method or a property that can return the Entity Connection based on the tenant identification at run time. The following figure illustrates the implementation

```
private static string _connectionStringName = "AdventureWorksEntities";
private static EntityConnection _getConnection
{
    get
    {
        var connectionString = DbMetaData.GetConnectionString(_connectionStringName);

        var connectionStringObject = new EntityConnectionStringBuilder();
        connectionStringObject.Metadata = "res://*/";
        connectionStringObject.Provider = connectionString.ProviderName;
        connectionStringObject.ProviderConnectionString = connectionString.ConnectionString;

        return new EntityConnection(connectionStringObject.ConnectionString);
    }
}
```

4. To begin with the CRUD implementations for the Product DAL create a context object and perform the CRUD operations like adding a new object, updating an existing object and deleting the object.
5. Also, ensure that there are suitable conversions being made from the database object to the business object and vice-versa.
6. The following sample shows one such operation being done

Create method implementation

7. For the autogenerated entity identifier's use Guid.NewGuid() as Entity framework 4.1 does not support default database generated id propagation in the conceptual layer.

```
protected override string DoCreate(DataCreateRequest dataCreateRequest)
{
    ProductDetails product = (ProductDetails)dataCreateRequest.Entity;
    using (AdventureWorksEntities context = new AdventureWorksEntities(_getConnection))
    {
        product.rowguid = Guid.NewGuid();
        product.ProductModel.rowguid = Guid.NewGuid();
        product.ProductDescription.rowguid = Guid.NewGuid();

        product.Identifier = product.rowguid.ToString();

        context.Products.AddObject(ObjectConverter.GetDatabaseObject(product));
        context.SaveChanges();
    }
    return product.Identifier;
```

Update method implementation

```
protected override void DoUpdate(DataUpdateRequest dataUpdateRequest)
{
    using (AdventureWorksEntities context = new AdventureWorksEntities(_getConnection))
    {
        ProductDetails productDetails = (ProductDetails)dataUpdateRequest.Entity;

        var product = ObjectConverter.GetDatabaseObject(productDetails);

        context.Products.Attach(product);
        context.ObjectStateManager.ChangeObjectState(product, System.Data.EntityState.Modified);
        context.ObjectStateManager.ChangeObjectState(product.ProductDescription, System.Data.EntityState.Modified);
        context.ObjectStateManager.ChangeObjectState(product.ProductModel, System.Data.EntityState.Modified);

        context.SaveChanges();
    }
}
```

Fetch method implementation

```
protected override ProductDetails DoFetch(DataFetchRequest dataFetchRequest)
{
    using (AdventureWorksEntities context = new AdventureWorksEntities(_getConnection))
    {
        context.Products.MergeOption = System.Data.Objects.MergeOption.NoTracking;

        var id = Guid.Parse(dataFetchRequest.Identifier);

        var product = context.Products.Include("SalesOrderDetails").Include("ProductDescription")
                                      .Include("ProductCategory").Include("ProductModel")
                                      .SingleOrDefault(x => x.rowguid == id);

        return ObjectConverter.GetBusinessObject(product);
    }
}
```

Search method implementation

8. In this implementation, use the include method to include the necessary related entities

```
protected override Dictionary<string, ProductDetails> DoSearch(DataSearchRequest dataSearchRequest)
{
    using (AdventureWorksEntities context = new AdventureWorksEntities(_getConnection))
    {
        context.Products.MergeOption = System.Data.Objects.MergeOption.NoTracking;

        IQueryable<Product> products = context.Products.Include("SalesOrderDetails").Include("ProductDescription")
                                        .Include("ProductCategory").Include("ProductModel").FilterTenant();

        if (dataSearchRequest != null)
        {
            if ((dataSearchRequest.Identifiers != null && dataSearchRequest.Identifiers.Count() > 0))
            {
                products = products.Where(p => dataSearchRequest.Identifiers.Contains(p.rowguid.ToString()));
            }
        }

        var productSearchCondition = dataSearchRequest.SearchCondition as ProductSearchCondition;

        int pageSize = !string.IsNullOrEmpty(productSearchCondition.Size) ? int.Parse(productSearchCondition.Size) : 0;

        products = products.OrderBy(x => x.Name);

        productSearchCondition.TotalCount = products.Count();

        products = products.Skip((productSearchCondition.PageNo - 1) * pageSize).Take(pageSize);

        return ObjectConverter.GetBusinessObjects(products.ToList());
    }
}
```

Delete method implementation

```
protected override void DoDelete(DataDeleteRequest dataDeleteRequest)
{
    using (AdventureWorksEntities context = new AdventureWorksEntities(_getConnection))
    {
        var productBusinessObject = dataDeleteRequest.Entity as ProductDetails;

        Product product = ObjectConverter.GetDatabaseObject(productBusinessObject);

        // 1. Attach the product
        context.Products.Attach(product);

        // 2. Set No tracking for all the required entities
        context.ProductModels.MergeOption = System.Data.Objects.MergeOption.NoTracking;
        context.ProductDescriptions.MergeOption = System.Data.Objects.MergeOption.NoTracking;
        context.Products.MergeOption = System.Data.Objects.MergeOption.NoTracking;

        // 3. Attach the child models and then delete those objects
        context.ProductDescriptions.Attach(product.ProductDescription);
        context.ProductDescriptions.DeleteObject(product.ProductDescription);

        // 4. Attach the child models and then delete those objects
        context.ProductModels.Attach(product.ProductModel);
        context.ProductModels.DeleteObject(product.ProductModel);

        // 5. Delete the parent entity at the last step
        context.Products.DeleteObject(product);
        context.SaveChanges();
    }
}
```

9. Similarly for all the entities that derive from the BaseEntity, the Identifier property should be filled in before passing the entity to the caller. This has to be followed for the Fetch, Search, Insert methods.

1 Contact Information

Any problem using this guide (or) using Cello Framework. Please feel free to contact us, we will be happy to assist you in getting started with Cello.

Email: support@techcello.com

Phone: +1(609)503-7163

Skype: techcello