PUTTING ENTITY FRAMEWORK 4 TO USE IN A BUSINESS ARCHITECTURE

– USING CODE-ONLY, INVERSION OF CONTROL & WCF AND SOME…

The intention of this document is to summarize the results of my latest posts on my blog about Entity framework 4 (EF4). I wrote these entries while I was exploring EF4 beta 2 and the new *Code-only strategy* provided by the CTP2 addition. As time went by I covered a bit more like: specific mapping scenarios; how-to setup generic object contexts so you don't need specific repositories for each aggregate root (see repository pattern, Fowler);how-to implement entity validation using the System.ComponentModel.DataAnnotations; how-to make use of client proxies and service hosts in Windows Communication Foundation (WCF) for setting up a distributed environment without the need of service references. How-to make use of Inversion-of-control (IoC) with (StructureMap) so that you easily can switch each service from being distributed or not (*that is, using WCF or not*). So this document is more the result of bringing all these pieces together to form the foundation of an architecture with a domain model and a service layer as the core principles. I will not make theoretical deep-dives in the technology. What I will do is show you a lot of code and instead describe what it does and the intentions.

The code is downloadable from here.

Note! I'm basing this document on code that was written using beta and CTP products that also might not have go-live licenses yet.

> *I hope you find this writing somewhat inspiring and if you don't already have started, that I can get you to start fiddling with these concepts on your own.*
>
> *//Daniel*

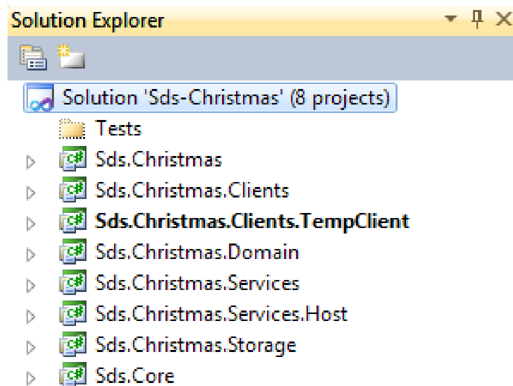
Table of contents

What's in the download? 4
Sds.Christmas 4
Sds.Christmas.Clients 4
Sds.Christmas.Clients.TemplClient 4
Sds.Christmas.Domain 4
Sds.Christmas.Services 4
Sds.Christmas.Services.Host 5
Sds.Christmas.Storage 5
Sds.Core 5
The domain 6
IEntity & Entity base class 6
User account entity 7
Wish list entity 9
Wish entity 10
Tags entity 11
The Mappings 12
Entity base class mappings (Identity & Concurrency token) 12
User account mappings 13
Wish list mappings (Relationship) 14
Wish mappings 15
Tags mappings (Mapping members that are out of scope) 16
How-to Create expression for private properties or fields 16
Other mappings 17
How-to change target names in the database table 17
How-to map a combined primary key 18
CRUD for entities 19
Why not Repositories? 19
IEntityStore & EfEntityStore 20

EfEntityContext 22
EfEntityContextBuilder 23
How-to consume the EfEntityStore 25
Inversion of Control using StructureMap 26
The Services 28
Consuming the Services 28
Basic services & WCF-services 30
ServiceResponse 31
Service engine 34
WCF Service host 36
WCF Client proxies 39
Validation of entities 41
Simple data validations 41
Custom Entity validators 42
EntityValidator 42
The end… 46

What's in the download?

There is one solution containing eight projects.



Let's go through them:

Sds.Christmas

The idea is that it should contain resources that are shared amongst the other projects. This could be constants, RESXs, definitions (interfaces), exceptions etc. For most situations, if something can be placed in a more specific assembly, it should go there instead.

Sds.Christmas.Clients

Is an assembly containing common resources that can/should be reused between different Client-implementations.

Sds.Christmas.Clients.TemplClient

This is a test-console application that I have used while exploring the API of EF4 etc. A better approach would have been to have tests for this instead.

Sds.Christmas.Domain

Contains the model and logic for the domain that we are building the system for. In this case it contains quite slim entities that are being persisted with EF4. The entities are decorated with attributes that takes care of "data validation". That is validation stating "*this field X, can't be left blank and must be a number between range Y and Z*". For more complex validation, specific validation methods are used on specific *Entity validators*. This lets me build complex validators that can be specific for certain countries etc.

Sds.Christmas.Services

All clients (whether they are consoles, Silverlight applications, Winforms or other services) that want to interact with the domain shall go via this layer. By using IoC, you can easily select if a certain service shall be a "plain service" (not distributed, running in-process) or a "WCF service" (distributed). The Services shall not contain logic that is part of the domain (see anemic domain model, Fowler), such as validation etc. The Services shall only orchestra the work (control the flow) for the task being performed. The Services doesn't expose a domain object directly, but instead incorporates this result in a ServiceResponse-instance.

Sds.Christmas.Services.Host

Contains a simple console host for the WCF services. Is configured to be using net.tcp-binding, this since I own both ends (server and client) and that both are .Net applications. Per default the TempClient is not using this host for hosting the WCF services. Instead it uses in-process self hosted WCFs, which make it easier to perform debugging and to get the samples started. If you want to use the specific host you will have to change the code in the TempClient. I will show you what to change further down in this document.

Sds.Christmas.Storage

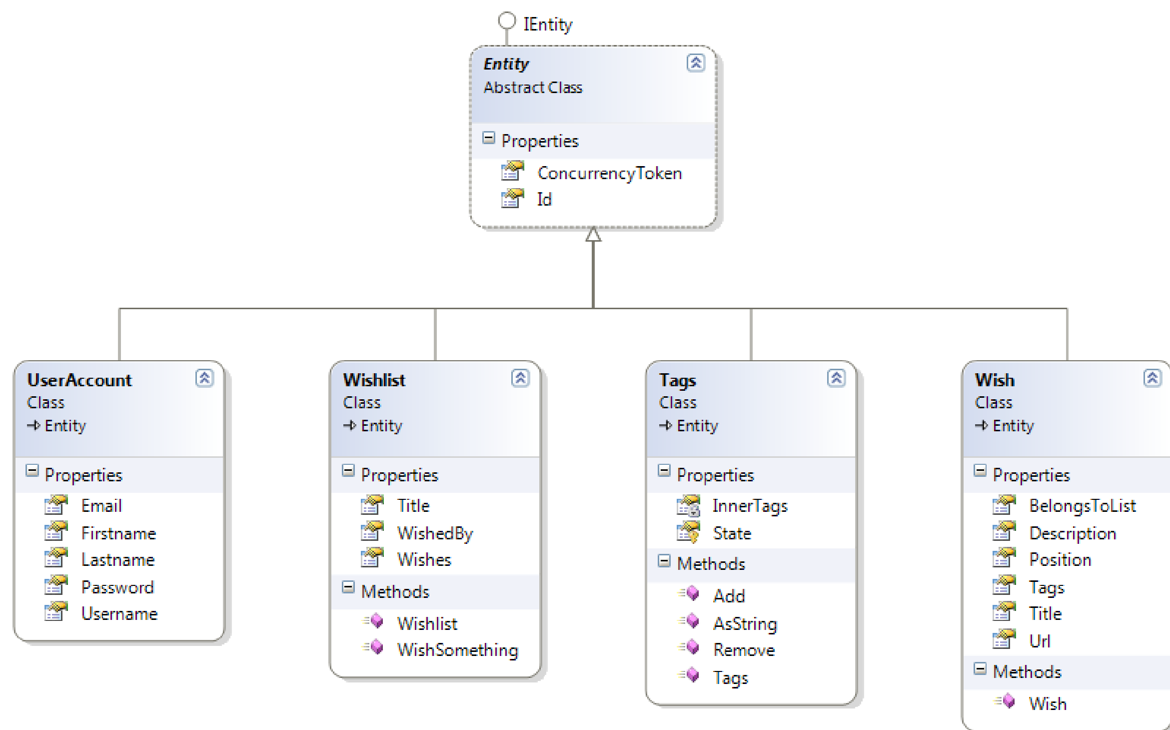
Contains code that has the responsibility of persisting and deleting entities as well as reading them back. In the example, it only contains the mappings that are used for EF. If you would, you could place a specific *EfEntityStore* in this assembly, which then is specific for storage operations within the Christmas context. You could do this to create repositories for you aggregate roots. If I would implement another storage provider, e.g. NHibernate, it will be placed under the Storage-namespace.

Sds.Core

Contains pure boilerplate, infrastructure code. Is to be seen as Microsofts Core-lib and contains functionality that isn't tied to a certain domain, but instead can be used among all "Sds.X" projects. This is the place where the *EfEntityStore* is implemented.

The domain

The domain is purely fictive and just something I played with when I was testing EF4. The idea is that a user can create a Wish list for Christmas. Each Wish list contains a prioritized list of wishes. Except priority, each Wish can be assigned Title, Description and Tags. Basically it's a quite simple domain.



The entities are used both at the client as well as in the services, hence I'm not using any DTO's for distribution, nor am I using a certain view model that is customized for my UI. The reason for this is that I'm just playing with the techniques and I would **consider** using DTOs and a custom view model if this was a real case.

IEntity & Entity base class

IEntity is currently empty and is just used to indicate that a certain class represents an entity in a domain. The interface is located in the assembly Sds.Core. The interface can be used for defining constraints when using generics or in conjunction with reflection to indicate "this is an Entity". When I find common properties or behavior, it will be added to this interface.

Code – IEntity

```
namespace Sds.Core
{
  public interface IEntity
  {
  }
}
```

For the Christmas domain I have chosen to have a base class which implements IEntity. All other entities within Christmas, extends this class.

Code – Entity

```
namespace Sds.Christmas.Domain.Model
{
    [Serializable]
    [DataContract(Namespace = DomainConstants.DataContractNamespace, IsReference = true)]
    public abstract class Entity : IEntity
    {
        [DataMember]
        public virtual int Id { get; set; }

        [DataMember]
        public virtual byte[] ConcurrencyToken { get; set; }
    }
}
```

Since I want EF to handle concurrency for me, I have implemented a *ConcurrencyToken*. Except from the auto-property above, all I have to do is to mark this property as a concurrency token in the mappings for my entities; then EF will include this concurrency token within the SQL so that optimistic concurrency checks are enforced so that "last in doesn't win".

I'm adding both Serializable and DataContract attributes, so that the DataContractSerializer (other exists: NetDataContractSerializer and XmlObjectSerializer)get more specific instructions of what to serialize. I also tell it to include information about relations to other objects in the generated xml (IsReference=true). In the Entity case, I have no relations but my sub classes do have relations and this can't be overridden in subclasses, hence I'm specifying it in the Entity base class.

User account entity

The user account is really simple and contains only some **virtual** auto properties. The reason for why I'm adding the virtual keyword is that I want EF to be able to generate dynamic proxies for my entities. The user account entity also contains some simple data annotation attributes, stating whether properties are optional or not. How long they can be etc. I have also created a custom attribute for checking if a certain property conforms to the format of a *valid email address*.

I have chosen to make use of *resx-files* for dealing with validation messages, so that I can make them localizable. When doing so you need to provide the ErrorMessageResourceName and the ErrorMessageResourceType.

Code – User account entity

```
namespace Sds.Christmas.Domain.Model
{
    [Serializable]
    [DataContract(Namespace = DomainConstants.DataContractNamespace, IsReference = true)]
    public class UserAccount
        : Entity
    {
        [Required(
            AllowEmptyStrings = false,
            ErrorMessageResourceName = "UsernameIsRequired",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [StringRange(MinLength = 5,
            MaxLength = 20,
            ErrorMessageResourceName = "UsernameHasInvalidLength",
```

```
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [DataMember]
        public virtual string Username { get; set; }

        [Required(
            AllowEmptyStrings = false,
            ErrorMessageResourceName = "PasswordIsRequired",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [StringRange(
            MinLength = 5,
            MaxLength = 20,
            ErrorMessageResourceName = "PasswordHasInvalidLength",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [DataMember]
        public virtual string Password { get; set; }

        [Required(
            AllowEmptyStrings = false,
            ErrorMessageResourceName = "EmailIsRequired",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [Email(
            ErrorMessageResourceName = "EmailHasInvalidFormat",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [StringLength(
            150,
            ErrorMessageResourceName = "EmailHasInvalidLength",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [DataMember]
        public virtual string Email { get; set; }

        [Required(
            AllowEmptyStrings = false,
            ErrorMessageResourceName = "FirstnameIsRequired",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [StringLength(
            100,
            ErrorMessageResourceName = "FirstnameHasInvalidLength",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [DataMember]
        public virtual string Firstname { get; set; }

        [Required(
            AllowEmptyStrings = false,
            ErrorMessageResourceName = "LastnameIsRequired",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [StringLength(
            100,
            ErrorMessageResourceName = "LastnameHasInvalidLength",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [DataMember]
        public virtual string Lastname { get; set; }

        public string Name { get { return "{0} {1}".Inject(Firstname, Lastname); } }
    }
}
```

Wish list entity

The Wish list is a little bit more complex than the User account. This since we now have to deal with navigation properties. I want to be able to navigate from a Wish list instance to the owner of the Wish list, which is represented by ”WishedBy”. *One downside with EF* is that I do have to expose the list of Wishes. The reason for this is that I want to be able to use lazy loading and execute the Include statement when querying EF for my wish lists. For this to work the getter have to be public whilst the setter can be private. It doesn't work to expose a read-only collection since they use the getter for population work. Hopefully this is something they will changes. This since I don't always want to expose the state of my child items. The reason for this can be as simple as the “WishSomething” method below, which contains some logic that determines the positions of the Wish being added.

Code –Wish list entity

```
namespace Sds.Christmas.Domain.Model
{
    [Serializable]
    [DataContract(Namespace = DomainConstants.DataContractNamespace, IsReference = true)]
    public class Wishlist : Entity
    {
        [Required(
            AllowEmptyStrings = false,
            ErrorMessageResourceName = "TitleIsRequired",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [StringLength(
            250,
            ErrorMessageResourceName = "TitleHasInvalidLength",
            ErrorMessageResourceType = typeof(ValidationMessages))]
        [DataMember]
        public virtual string Title { get; set; }

        [DataMember]
        public virtual UserAccount WishedBy { get; set; }

        [DataMember]
        public virtual IList<Wish> Wishes { get; private set; }

        public Wishlist()
        {
            Wishes = new List<Wish>();
        }

        public virtual void WishSomething(Wish wish)
        {
            wish.Position = Wishes.Count < 1 ? 1 : Wishes.Max(w => w.Position) + 1;
            Wishes.Add(wish);
            wish.BelongsToList = this;
        }
    }
}
```

Wish entity

The wish and wish list has bidirectional navigation properties, hence I can go from a wish to the wish list as well as from the wish list to its wishes. A wish also has a relationship to Tags.

Code – Wish entity

```
namespace Sds.Christmas.Domain.Model
{
  [Serializable]
  [DataContract(Namespace = DomainConstants.DataContractNamespace, IsReference = true)]
  public class Wish : Entity
  {
    [Required(
      AllowEmptyStrings = false,
      ErrorMessageResourceName = "TitleIsRequired",
      ErrorMessageResourceType = typeof(ValidationMessages))]
    [StringLength(
      250,
      ErrorMessageResourceName = "TitleHasInvalidLength",
      ErrorMessageResourceType = typeof(ValidationMessages))]
    [DataMember]
    public virtual string Title { get; set; }

    [DataMember]
    public virtual Wishlist BelongsToList { get; internal set; }

    [DataMember]
    public virtual int Position { get; internal set; }

    [DataMember]
    public virtual string Description { get; set; }

    [DataMember]
    public virtual Tags Tags { get; set; }

    [StringLength(
      250,
      ErrorMessageResourceName = "UrlHasInvalidLength",
      ErrorMessageResourceType = typeof(ValidationMessages))]
    [DataMember]
    public virtual string Url { get; set; }

    public Wish()
    {
      Position = 1;
      Tags = new Tags();
    }
  }
}
```

Tags entity

The only property that is persisted and handled by EF is *State*. The state is represented by a comma delimited string.

Code – Tags entity

```
namespace Sds.Christmas.Domain.Model
{
    [Serializable]
    [DataContract(Namespace = DomainConstants.DataContractNamespace, IsReference = true)]
    public class Tags : Entity
    {
        [DataMember]
        protected virtual string State
        {
            get
            {
                return string.Join(",", InnerTags.ToArray());
            }
            set
            {
                InnerTags = new HashSet<string>(value.Split(",".ToArray(), StringSplitOptions.RemoveEmptyEntries));;
            }
        }

        private ISet<string> InnerTags { get; set; }

        public Tags(params string[] tags)
        {
            InnerTags = new HashSet<string>(tags);
        }

        public virtual void Add(params string[] tags)
        {
            InnerTags.UnionWith(tags);
        }

        public virtual void Remove(params string[] tags)
        {
            InnerTags.ExceptWith(tags);
        }

        public virtual string AsString()
        {
            return State;
        }
    }
}
```

The Mappings

As all my entities in the Christmas domain is inheriting the Entity base class I have a base class for the mappings to. The base class takes care of the mappings of the members in the Entity base class. Since I want it to be easy to add new entities to work with, I have implemented a function that *auto registers the mappings for me*. For this to work more easily I have implemented an empty interface *IChristmasMapping* that acts as a marker for classes that contains mapping instructions. This interface is then searched for using reflection, and the found classes are then instantiated and added to the *EfEntityContextBuilder*. Since it is safe to reuse the *EfEntityContextBuilder*, *this auto registration is only done once*.

Entity base class mappings (Identity & Concurrency token)

I have decided that all my entities shall have an Id-property that is implemented as an identity; hence the database (DB) will have an identity column for each table. I have also added a Concurrency token to each entity, which is mapped as a *timestamp* in the database (you can also use *rowversion*). **Note!** The data type on the C# class shall be *byte[]*.

Code –EntityMapping

```
namespace Sds.Christmas.Storage.Mappings
{
   [Serializable]
   public abstract class EntityMapping<TEntity> : EntityConfiguration<TEntity>, IChristmasEntityMapping
      where TEntity : Entity
   {
      protected EntityMapping()
      {
         HasKey(u => u.Id);
         Property(u => u.Id).IsIdentity();

         Property(u => u.ConcurrencyToken)
            .IsRequired()
            .IsConcurrencyToken()
            .HasStoreType("timestamp")
            .StoreGeneratedPattern = StoreGeneratedPattern.Computed;
      }
   }
}
```

User account mappings

Since my UserAccountMapping class extends the EntityMapping class, I don't have to add the IChristmasMapping interface, nor do I have to map the Id and ConcurrencyToken properties. The mappings API is fluent and lets you perform chained calls to map a single property in one statement. The more details you specify in your mappings, the more details will be included in the DDL-script generated by EF, so the more you specify; the less default values are used when the tables are created for you.

IsRequired = Not null; IsNotUnicode = VarChar; IsUnicode = NVarChar;

Code – UserAccountMapping

```
namespace Sds.Christmas.Storage.Mappings
{
    [Serializable]
    public class UserAccountMapping : EntityMapping<UserAccount>
    {
        public UserAccountMapping()
        {
            Property(u => u.Email)
                .IsRequired()
                .IsNotUnicode()
                .MaxLength = 150;

            Property(u => u.Username)
                .IsRequired()
                .IsUnicode()
                .MaxLength = 20;

            Property(u => u.Password)
                .IsRequired()
                .IsUnicode()
                .MaxLength = 20;

            Property(u => u.Firstname)
                .IsRequired()
                .IsUnicode()
                .MaxLength = 100;

            Property(u => u.Lastname)
                .IsRequired()
                .IsUnicode()
                .MaxLength = 100;
        }
    }
}
```

Wish list mappings (Relationship)

What's new in this mapping is the simple relationship to a User account instance. This is accomplished by passing a lambda expression to the Relationship function. Note that the lambda expression states *how to get hold of the object that is at the other end of the relation*; it doesn't point to the Id of the object. I also specify that it *is required* (*Not null*). The generated database table for Wish lists will contain a *non nullable* column named "*WishedBy_Id*" which is implemented as a *foreign key* to the UserAccounts table.

Code –WishlistMapping

```
namespace Sds.Christmas.Storage.Mappings
{
   [Serializable]
   public class WishlistMapping : EntityMapping<Wishlist>
   {
      public WishlistMapping()
      {
         Property(wl => wl.Title)
            .IsRequired()
            .IsUnicode()
            .MaxLength = 250;

         Relationship<UserAccount>(wl => wl.WishedBy).IsRequired();
      }
   }
}
```

Wish mappings

The only thing that is new in this mapping is the "*IsOptional*" statement. It results in that *the foreign key can be null*.

Code –WishMapping

```
namespace Sds.Christmas.Storage.Mappings
{
    [Serializable]
    public class WishMapping : EntityMapping<Wish>
    {
        public WishMapping()
        {
            Property(w => w.Title)
                .IsRequired()
                .IsUnicode()
                .MaxLength = 250;

            Property(w => w.Url)
                .IsOptional()
                .IsUnicode()
                .MaxLength = 250;

            Property(w => w.Description)
                .IsOptional()
                .IsUnicode();

            Relationship<Wishlist>(w => w.BelongsToList).IsRequired();

            Relationship<Tags>(w => w.Tags).IsOptional();
        }
    }
}
```

Tags mappings (Mapping members that are out of scope)

The tags entity only maps one property of the Tags entity; The State property. It contains a string of tags, where each tag is separated by a comma. *The State property is private which is not supported by EF*. I have provided some simple code that lets you pass a string for the member to return an expression for, so that private members could be mapped.

Code – TagsMapping

```
namespace Sds.Christmas.Storage.Mappings
{
    [Serializable]
    public class TagsMapping : EntityMapping<Tags>
    {
        public TagsMapping()
        {
            Property(ObjectAccessor<Tags>.CreateExpression<string>("State"))
                .IsRequired()
                .IsUnicode()
                .MaxLength = 1000;
        }
    }
}
```

How-to Create expression for private properties or fields

This could is used to create an expression that can be used in conjunction with the mapping classes; so that mappings for private members can be configured.

Code – CreateExpression

```
public static Expression<Func<T, TResult>> CreateExpression<TResult>(string propertyOrFieldName)
{
    ParameterExpression param = Expression.Parameter(typeof(T), "propertyOrFieldContainer");
    Expression body = Expression.PropertyOrField(param, propertyOrFieldName);
    LambdaExpression lambda = Expression.Lambda(typeof(Func<T, TResult>), body, param);

    return (Expression<Func<T, TResult>>) lambda;
}
```

Other mappings

While doing my exploration of EF and writing about it at my blog, I got some questions about mappings. I have included these here.

How-to change target names in the database table

For a couple of days ago I got the question of “my mappings don’t work”. I looked at it and found that the database tables were generated upfront and not by the ObjectContext *(Personally, I let the context create the database and the tables, since I want the model to drive the database design)*. The column names had other names. Ok, how-do you solve this mismatch if you really do want to have separate naming? You use *MapSingleType*.

Code – The Entity

```
[Serializable]
public class Car
{
    public string LicencePlate { get; set; }
    public int CountryCode { get; set; }
}
```

Code – The Mappings

```
[Serializable]
public class CarMapping : EntityConfiguration<Car>
{
    public CarMapping()
    {
        HasKey(c => new { c.CountryCode, c.LicencePlate });
        MapSingleType(c =>
            new
            {
                TheCountryCode = c.CountryCode,
                TheLicencePlate = c.LicencePlate
            }).ToTable("dbo.TheCars");
    }
}
```

The result

TABLE: TheCars. *(If you don’t use “ToTable” the normal naming rules are applied)*.

COLUMNS:

- TheCountryCode
- TheLicencePlate

How-to map a combined primary key

Another question I have got is how-to map an entity with a primary key that is built up using two fields. The answer: *using an anonymous type*.

Code – The Entity

```
[Serializable]
public class Car
{
    public string LicencePlate { get; set; }
    public int CountryCode { get; set; }
}
```

Code – The Mappings

```
[Serializable]
public class CarMapping : EntityConfiguration<Car>
{
    public CarMapping()
    {
        HasKey(c => new { c.CountryCode, c.LicencePlate });
    }
}
```

CRUD for entities

Ok, now we have got some entities and some mappings for the entities and the next step is to look at the Sds.Core.Storage.Ef.EfEntityStore, which is used for CRUD operations for your entities. As of right now the CTP2 only supports SqlClients (Microsoft Sql Server). As the naming of the namespaces indicates, it is not tied to the Christmas domain, and is designed for being reused within several domains.

Why not Repositories?

The naming "EntityStore" is an active design decision, since the entity store is designed for dealing with numerous different entity types in the same instance; compared to a repository that shall deal with CRUD operations for only one aggregate root per repository. In the solution I have put together I have no repositories or specific entity stores. Why? I don't need them. All my entities are having very basic CRUD operations and the only thing that differs between the entities is that in some cases there are specific queries to execute. E.g. a query for determining if a username already is in use or not. This is a query that normally would be put in a repository designed for dealing with User accounts. The only thing that differs this User account repository from another repository would be the named query (a function that returns entities or data bounded to a certain entity). Instead of a custom store or repository, I like to extend the IQueryable interface and when the consumer of the store needs a certain query, he/she only have to add a using statement to gain access to the named queries.

Code – Named Queries

```
using Sds.Christmas.Domain.Queries.UserAccounts;
…
…
_entityStore.Query<UserAccount>().UsernameIsTaken(userAccount.Username);
```

The member Query returns an IQueryable of UserAccount, hence the imported extension method located under Queries.UserAccounts will be available.

```
namespace Sds.Christmas.Domain.Queries.UserAccounts
{
  public static class UserAccountQueries
  {
    public static bool UsernameIsTaken(this IQueryable<UserAccount> userAccounts, string username)
    {
      return
        userAccounts.Where(
          u =>
          u.Username.Equals(username, StringComparison.InvariantCultureIgnoreCase)).Count() > 0;
    }

    public static bool EmailIsTaken(this IQueryable<UserAccount> userAccounts, string email)
    {
      return
        userAccounts.Where(
          u =>
          u.Email.Equals(email, StringComparison.InvariantCultureIgnoreCase)).Count() > 0;
    }

    public static UserAccount GetByCredentials(this IQueryable<UserAccount> useraccounts, string username,
string password)
    {
```

```
        var match = useraccounts.Where(
          u =>
          u.Username.Equals(username, StringComparison.InvariantCultureIgnoreCase) &&
          u.Password.Equals(password, StringComparison.InvariantCultureIgnoreCase)).SingleOrDefault();

        return match;
      }
    }
}
```

IEntityStore & EfEntityStore

The EfEntityStore implements the interface IEntityStore. This interface defines the members of an entity store and lets me in e.g. create a NHibernate based entity store; or if I want to make custom EfEntityStores (by inheriting the EfEntityStore) I can handle them in an abstract manner.

Code – IEntityStore

```
namespace Sds.Core.Storage
{
  public interface IEntityStore : IDisposable
  {
    void EnsureDatabaseExists();

    void EnsureCleanDatabaseExists();

    void AttachEntityAsModified<T>(T entity) where T : class, IEntity;

    void DetachEntity<T>(T entity) where T : class, IEntity;

    void AddEntity<T>(T entity) where T : class, IEntity;

    void DeleteEntity<T>(T entity) where T : class, IEntity;

    IQueryable<T> Query<T>() where T : class, IEntity;

    IQueryable<T> Query<T>(params string[] includes) where T : class, IEntity;

    int SaveChanges();
  }
}
```

The only thing here that is quite EF specific is the definition:

```
IQueryable<T> Query<T>(params string[] includes) where T : class, IEntity;
```

It is used for telling EF to include not just the first level in the object graph but also member “X.Y.Z”.

Code – Example of include when querying EF

```
entityStore.Query<Wishlist>("WishedBy", "Wishes")
```

This instructs EF to retrieve WishedBy (User account) and the contained wishes in the result. You can add several levels like “Wishes.Tags”, which then will pull out three levels: Wish list | Wishes | Tags.

The EfEntityStore implementation is merely a wrapper around an EfEntityContext and is a storage provider that uses Entity framework to handle POCO entities. Either use it directly or Inherit from it and create your custom stores/repositories for your domain.

Note! I have chosen to use the name *Entity* instead of EF's notion of *Object*, hence I will have EfEntityContext instead of ObjectContext. Why I have chosen Entity is because I think it's more logical since the framework is about CRUD for entities and not "objects". Also, it gives a clear distinction of what are my custom classes that wraps and extends the EF; and what classes that belongs to EF.

Code – EfEntityStore

```
namespace Sds.Core.Storage.Ef
{
  public class EfEntityStore : IEntityStore
  {
    protected EfEntityContext EfEntityContext { get; private set; }

    public EfEntityStore(EfEntityContext efEntityContext)
    {
      EfEntityContext = efEntityContext;
    }

    public void Dispose()
    {
      EfEntityContext.Dispose();
    }

    public void EnsureDatabaseExists()
    {
      if (EfEntityContext.DatabaseExists())
        return;

      EfEntityContext.CreateDatabase();
    }

    public void EnsureCleanDatabaseExists()
    {
      if (EfEntityContext.DatabaseExists())
        EfEntityContext.DeleteDatabase();

      EfEntityContext.CreateDatabase();
    }

    public void AttachEntityAsModified<T>(T entity) where T : class, IEntity
    {
      EfEntityContext.EntitySet<T>().Attach(entity);
      EfEntityContext.ObjectStateManager.ChangeObjectState(entity, System.Data.EntityState.Modified);
    }

    public void DetachEntity<T>(T entity) where T : class, IEntity
    {
      EfEntityContext.EntitySet<T>().Detach(entity);
    }

    public void AddEntity<T>(T entity) where T : class, IEntity
    {
```

```
            EfEntityContext.EntitySet<T>().AddObject(entity);
        }

        public void DeleteEntity<T>(T entity) where T : class, IEntity
        {
            EfEntityContext.EntitySet<T>().DeleteObject(entity);
        }

        public IQueryable<T> Query<T>() where T : class, IEntity
        {
            return EfEntityContext.EntitySet<T>();
        }

        public IQueryable<T> Query<T>(params string[] includes) where T : class, IEntity
        {
            if (includes == null || includes.Length < 1)
                return Query<T>();

            return includes.Aggregate(EfEntityContext.EntitySet<T>() as ObjectQuery<T>, (current, include) =>
current.Include(include));
        }

        public int SaveChanges()
        {
            return EfEntityContext.SaveChanges();
        }
    }
}
```

As you can see, it only wraps calls to the EfEntityContext, which is a custom class that extends EF's ObjectContext class. If you are familiar with NHibernate, the EfEntityContext would be the Session.

EfEntityContext

The EfEntityContext is only adding one member to the existing ObjectContext in EF; and that is:

```
public ObjectSet<T> EntitySet<T>() where T : class, IEntity
```

This member lets you get rid of those ugly members that specifically returns an ObjectSet of a certain entity type. If you really want this, inherit the EfEntityStore and add the member there. This is a common scenario for repositories, but for me, it really isn't necessary (read more about this above).

Code – EfEntityContext

```
namespace Sds.Core.Storage.Ef
{
    public class EfEntityContext : ObjectContext
    {
        private readonly Dictionary<Type, object> _entitySets;

        public EfEntityContext(EntityConnection connection)
            : base(connection)
        {
            _entitySets = new Dictionary<Type, object>();
        }
```

```
public ObjectSet<T> EntitySet<T>()
        where T : class, IEntity
    {
        var t = typeof(T);
        object match;

        if (!_entitySets.TryGetValue(t, out match))
        {
            match = CreateObjectSet<T>();
            _entitySets.Add(t, match);
        }

        return (ObjectSet<T>)match;
    }
  }
}
```

EfEntityContextBuilder

The EfEntityContextBuilder is just as the name implies: "*a builder responsible for building new instances of EfEntityContexts*". If you are familiar with NHibernate, this would be the Session factory. It is created once and then reused whenever a new EfEntityContext needs to be created. It contains e.g. the mappings for the entities.

The most important function of the builder is the factory method *CreateContext*. The other methods that I have added are there for making things smoother. Like *RegisterEntity*. It Registers entity sets and entity configurations (mappings) and when doing this I make use of the PluralizationService from Microsft. It lets you pluralize names depending on a certain culture. So if I provide the culture "en" and pass it the value "UseAccount" it will give me "UserAccounts"; "Wish" will result in "Wishes". Nice, isn't it? Other useful methods in the builder are the *MapEntities* overloads. They use reflection to find mapping classes and then adds these classes as mappings. If you want you can filter this action and pass a specific interface-type that the mappings you want to register should implement. I use this method and that's why I have added *IChristmasMapping* to my mapping classes.

Note! I don't use the names that Microsoft uses. They look at mappings as configurations, hence they use names like EntityConfiguration. For me, it's about mappings.

Code – EfEntityContextBuilder

```
namespace Sds.Core.Storage.Ef
{
  public class EfEntityContextBuilder : ContextBuilder<EfEntityContext>
  {
    private readonly Type _type;
    private readonly DbProviderFactory _factory;
    private readonly ConnectionStringSettings _cnStringSettings;
    private readonly PluralizationService _pluralizer;

    public EfEntityContextBuilder(string connectionStringName)
    {
      _type = GetType();
      _cnStringSettings = ConfigurationManager.ConnectionStrings[connectionStringName];
      _factory = DbProviderFactories.GetFactory(_cnStringSettings.ProviderName);
      _pluralizer = PluralizationService.CreateService(CultureInfo.GetCultureInfo("en"));
```

```
    }

    public EfEntityContext CreateContext()
    {
        var cn = _factory.CreateConnection();
        cn.ConnectionString = _cnStringSettings.ConnectionString;

        return Create(cn);
    }

    public void MapEntity<TEntity>(EntityConfiguration<TEntity> configuration)
        where TEntity : class, IEntity
    {
        RegisterEntity(configuration);
    }

    public void MapEntities(Assembly assembly)
    {
        MapEntities(GetMappingTypes(assembly));
    }

    public void MapEntities<TFilter>()
    {
        var filterType = typeof(TFilter);
        MapEntities(GetMappingTypes(filterType.Assembly, filterType));
    }

    private IEnumerable<Type> GetMappingTypes(Assembly assembly, Type filterInterfaceType = null)
    {
        var types = assembly.GetTypes().Where(a => a.Name.EndsWith("Mapping") && a.IsClass &&
!a.IsAbstract);

        if (filterInterfaceType == null)
            return types;

        if (!filterInterfaceType.IsInterface)
            throw new ArgumentException("The sent filterInterfaceType is not of an Interface.",
"filterInterfaceType");

        return types.Where(t => t.GetInterfaces().Contains(filterInterfaceType));
    }

    private void MapEntities(IEnumerable<Type> mappingTypes)
    {
        var method = _type.GetMethod("MapEntity");

        foreach (var mappingType in mappingTypes)
        {
            var entityConfigurationType = mappingType.BaseType;
            var entityType = entityConfigurationType.GetGenericArguments()[0];
            var generic = method.MakeGenericMethod(entityType);
            generic.Invoke(this, new[] { Activator.CreateInstance(mappingType) });
        }
    }
```

```
        public void RegisterEntity<TEntity>(EntityConfiguration<TEntity> entityConfiguration = null)
            where TEntity : class, IEntity
        {
            var pluralizedSetName = _pluralizer.Pluralize(typeof(TEntity).Name);

            RegisterSet<TEntity>(pluralizedSetName);

            if (entityConfiguration != null)
                Configurations.Add(entityConfiguration);
        }
    }
}
```

How-to consume the EfEntityStore

Let's look at an example how this will look like.

1. Create an EfEntityContextBuilder that builds EfEntityContexts
2. Create an EfEntityContext and inject it to an EfEntityStore
3. Consume the EfEntityStore to perform CRUD-operations on your entities.

Code – How-to use the EfEntityStore

```
var contextBuilder = new EfEntityContextBuilder("TheConnectionStringName");
contextBuilder.MapEntities<IChristmasEntityMapping>();

//Example of how-to add
using (IEntityStore entityStore = new EfEntityStore(contextBuilder.CreateContext()))
{
    entityStore.AddEntity<UserAccount>(userAccountToAdd);
    entityStore.SaveChanges();
}

//Example of how-to attach and update
using (IEntityStore entityStore = new EfEntityStore(contextBuilder.CreateContext()))
{
    entityStore.AttachEntityAsModified<UserAccount>(userAccountToUpdate);
    entityStore.SaveChanges();
}

//Example of how-to search
using (IEntityStore entityStore = new EfEntityStore(contextBuilder.CreateContext()))
{
    var match = entityStore.Query<UserAccount>()
            .Where(u => u.Email = "daniel@wertheim.se").SingleOrDefault();
}
```

I have explicitly used IEntityStore above where I just could have used "var". Why? Because I want to make it clear that, although the implementation I have included in this document, there is nothing that keeps me from switching this out to use an implementation that uses NHibernate, e.g NHibEntityStore. For being able to do a switch like that you should at least have a factory that creates the instances of your entity stores. I will be using Inversion of Control and will be using StructureMap for this.

Inversion of Control using StructureMap

In this solution I will be using StructureMap for resolving my resources in two places: one in the client where I resolve services (either plain services or WCF based services); and one in the services where I resolve EfEntityStores for being able to perform CRUD operations for my entities.

From the consumers point of view, I will not consume StructureMap directly, but instead work against an custom Interface. Why? I want to be able to switch StructureMap for another IoC-container.

Code – Client consuming the IoC-Container

```
var securityService = ClientEngine.Instance.IoC.Resolve<ISecurityService>();
```

Within the Client context I want to access the IoC-container in the same way, hence I don't want to write code like:

```
var securityService = new ClientComponentContainer().Resolve<ISecurityService>();
```

Instead I have one central resource (ClientEngine) accessed as a Singleton, which contains client centric resources. So far it only contains the IoC-container which is returned as an interface which looks like this:

```
public interface IComponentContainer
{
    T Resolve<T>();
}
```

StructureMap obviously has a lot richer interface, but in my solution I only need this method. The *ClientComponentContainer* is the implementation of an IoC-container that I use within the context of my Clients. I have another that I consume in my services, which is *ServiceComponentContainer*. The ComponentContainers extends an abstract class *StructureMapContainer*. This is a class I have put together just the ease the work of using StructureMap in a heterogeneous manner (ensuring that StructureMap is consumed in the same way in my solution). The ClientComponentContainer is responsible for configuring which components the container should handle and how they will be created.

In the code below I have configured the IoC-container to return service-proxies that are used to call WCF-implementations of my different services.

Code - ClientComponentContainer

```
namespace Sds.Christmas.Clients
{
    public class ClientComponentContainer : StructureMapComponentContainer
    {
        protected override void BootstrapContainer()
        {
            Container.Configure(x => x
                .ForRequestedType<IApplicationService>()
                .CacheBy(InstanceScope.PerRequest)
                .TheDefault.Is.ConstructedBy(() => new
WcfServiceClientProxy<IApplicationService>().ServiceChannel));

            Container.Configure(x => x
                .ForRequestedType<ISecurityService>()
                .CacheBy(InstanceScope.PerRequest)
                .TheDefault.Is.ConstructedBy(() => new WcfServiceClientProxy<ISecurityService>().ServiceChannel));
```

```
            Container.Configure(x => x
                .ForRequestedType<IWishlistService>()
                .CacheBy(InstanceScope.PerRequest)
                .TheDefault.Is.ConstructedBy(() => new WcfServiceClientProxy<IWishlistService>().ServiceChannel));
        }
    }
}
```

If I would like to use a non distributed environment, I just need to change one line per service, e.g.:

```
.TheDefault.Is.OfConcreteType<BasicSecurityService>());
```

You can of course extend the implementation to have an overload of the Resolve method that e.g. takes a key-string so that you could specify a specific implementation, and if you use the Resolve method that takes no key, the default implementation is returned.

The Services

In this chapter I will show you how I have put together a solution that lets you consume WCF-services without needing any service references etc. But before describing the implementation of my services I want to show the consuming code.

Consuming the Services

The code below uses the IoC-container in the *ClientEngine* to get a hold of a service reference. In the downloadable code, the default configuration is configured so that client proxies are used against WCF-services. Since this is a sample app and that I own both ends of the service, and I don't have any intentions of distributing this to other .Net consumers or JAVA consumers, I'm using my entities as data contracts. This is something I normally would have replaced with pure DTO's that represents requests and responses that are specific for the operation being performed on the service.

I like to look at my services as operations that reacts to a request and returns a response. I often create specific request objects, so that I can use requests that contain all the parameters instead of having to pass them one by one into the service-operation. I also want to get customized responses that contains only the data that is assumed to be returned by the operation.

Now let's look at the consuming code.

Code – Creating an new user account

```
private static UserAccount SetupNewUserAccount()
{
   var userAccount =
      new UserAccount
      {
         Firstname = "Daniel",
         Lastname = "Wertheim",
         Username = "dannyboy",
         Password = "ilovesnow",
         Email = "none@none.com"
      };

   var securityService = ClientEngine.Instance.IoC.Resolve<ISecurityService>();

   return ServiceExecutor.Execute<ISecurityService, UserAccount>(securityService,
      () =>
      {
         var serviceResponse = securityService.SetupNewUserAccount(userAccount);

         IfFailedServiceOperationInformUser(serviceResponse);

         return serviceResponse.Result;
      });
}
```

At first it may look a bit intrusive and I could have written it a bit more simplified by not using the ServiceExecutor.Execute method. Why I'm using this method is that I want to ensure that if the injected service "securityService" is an WCF-client proxy implementation instead of a basic service, it is closed and disposed. If I would like to, I could also add aspects like logging and exception handling in the Execute method. Basically, the code could have been written like this:

```
private static UserAccount SetupNewUserAccount()
{
   var userAccount =
      new UserAccount
      {
         Firstname = "Daniel",
         Lastname = "Wertheim",
         Username = "dannyboy",
         Password = "ilovesnow",
         Email = "none@none.com"
      };

   var securityService = ClientEngine.Instance.IoC.Resolve<ISecurityService>();

   try
   {
      var serviceResponse = securityService.SetupNewUserAccount(userAccount);

      return serviceResponse.Result;
   }
   finally
   {
      var wcfProxy = securityService as ICommunicationObject;
      if (wcfProxy != null)
         wcfProxy.Close();
      Disposer.TryDispose(securityService as IDisposable);
   }
}
```

The *ServiceExecutor* that was used in the first example is quite simple:

Code – The ServiceExecutor

```
namespace Sds.Core.SvcModel.Wcf
{
   public static class ServiceExecutor
   {
      public static void Execute<TService>(TService service, Action action)
      {
         try
         {
            action.Invoke();
         }
         finally
         {
            CloseConnectionToService(service as ICommunicationObject);
            Disposer.TryDispose(service as IDisposable);
         }
      }
```

```
public static TResult Execute<TService, TResult>(TService service, Func<TResult> func)
    {
      try
      {
        return func.Invoke();
      }
      finally
      {
        CloseConnectionToService(service as ICommunicationObject);
        Disposer.TryDispose(service as IDisposable);
      }
    }

    private static void CloseConnectionToService(ICommunicationObject serviceChannel)
    {
      if (serviceChannel == null)
        return;

      serviceChannel.Close();
    }
  }
}
```

Basic services & WCF-services

All my services are defined using an interface. The interfaces are implemented something I call a "basic service" which is an implementation which isn't going to be directly hosted as a Windows Communication Foundation (WCF) service. Despite this I have selected to decorate the interface with attributes that are used by the WCF infrastructure. These are attributes like: ServiceContract, OperationContract and a custom attribute that I made *EfDataContractSerializer*. In the example solution, the basic services contains the actual implementation. The WCF-services are just inheriting from my basic services and are not adding any custom behavior. The reason is that I want to be able to add custom code to my WCF-services, e.g. custom code for authentication/authorization. A typical implementation for a basic service could look like this:

Code – Basic service

```
namespace Sds.Christmas.Services.Basic
{
  public class BasicSecurityService : Service, ISecurityService
  {
    public virtual ServiceResponse<UserAccount> SetupNewUserAccount(UserAccount userAccount)
    {
      var response = new ServiceResponse<UserAccount> { Result = userAccount };

      using (var entityStore = ServiceEngine.Instance.IoC.Resolve<IEntityStore>())
      {
        var validator = new UserAccountValidator(entityStore);
        response.Try(validator.Validate(userAccount),
          () =>
          {
            entityStore.AddEntity(userAccount);
            entityStore.SaveChanges();
          });
      }

      return response;
    }
```

```
        ...
        ...
        ...
    }
}
```

Before we go throw the code I just want to show you how simple my WCF-service implementation is:

Code – WCF service

```
namespace Sds.Christmas.Services.Wcf
{
    public class WcfSecurityService : BasicSecurityService
    {
    }
}
```

User account validator

In the code above (Basic service) I make use of a User account validator. How I perform validation of my entities will be described later in this document, the only thing you have to know at this point is that it returns an *EntityValidatonResult* object that contains any violations in the entity for the current operation.

ServiceResponse

My services that return data do this via a *ServiceResponse*. This response object contains a flag indicating *Success* or *Failure*. It also holds the actual result as well as *validation results* and a FaultException. Hence I will not intentionally throw business exceptions. Instead I include validation results in my result and I use the flag to indicate the outcome. The FaultException is used since you are not supposed to throw Exceptions over the wire. In my case I'm using FaultException<TDetails> where I'm passing a custom *FaultDetails* instance that holds some information about assembly, class, method and exception type. *This is something I actually think you should be skeptic about*. Definetly if you are building services that is open for the "public". Why? Well how much value does the consumer gain from what assembly, class, method etc. that caused the exception? None! The consumer shall only have to check for "*Did my service call succeed or did it fail?*". If it fails, the consumer shall be able to see a descriptive message and check for specific violations that the request generated. The only time I'm actually providing FaultException is when something Unhandled did occur. To ease the implementation of this, I have included a *Try-method* in my ServiceResponse class (see below). The responsibility of this method is to execute the injected service code and to catch any exceptions and to generate a FaultException of any caught exception. You can also inject an *EntityValidationResult* instance, which is then assigned to the response. Note! To get the FaultException<FaultDetails> to work (that is the WCF infrastructure doesn't cut the connections, you need to add the FaultDatails as a known type, by decorating the ServiceResponse-class with the KnownTypeAttribute.

Code – Register FaultDetails as a known type

```
[KnownType(typeof(FaultDetails))]
public class ServiceResponse : IServiceResponse
```

Code – ServiceResponse, Try-method

```
public virtual ServiceResponseStatuses Try(EntityValidationResult validationResult, Action action)
{
    ValidationResult = validationResult;

    if (Status == ServiceResponseStatuses.Failed)
        return Status;

    try
    {
        action.Invoke();
    }
    catch(Exception ex)
    {
        Exception = new FaultException<FaultDetails>(new FaultDetails(ex), ex.Message);
    }

    return Status;
}
```

The injected service code (action) will not execute if the response-status already is failed (due to validation result with violations or earlier fail). If an exception is caught, the consumer will be able to get information as shown in the picture on the next side.

Name	Value
serviceResponse	{Sds.Core.SvcModel.ServiceResponse<Sds.Christmas.Domain.Model.UserAccount>}
base	{Sds.Core.SvcModel.ServiceResponse<Sds.Christmas.Domain.Model.UserAccount>}
_status	Failed
_validationResult	{Sds.Core.Validation.EntityValidationResult}
Fault	{"The calculator does not support denominators that are zero!"}
base	{"The calculator does not support denominators that are zero!"}
Detail	{Source: Sds.Christmas.Services||Service: BasicSecurityService||Method: <SetupNewUserAccount>b__1||Exceptiontype: DivideByZeroException||Message: The calculator does not support denominators that are zero!}
ExceptionTypeName	"DivideByZeroException"
Info	"Source: Sds.Christmas.Services||Service: BasicSecurityService||Method: <SetupNewUserAccount>b__1||Exceptiontype: DivideByZeroException||Message: The calculator does not support denominators that are zero!"
Message	"The calculator does not support denominators that are zero!"
MethodName	"<SetupNewUserAccount>b__1"
ServiceName	"BasicSecurityService"
SourceName	"Sds.Christmas.Services"
Non-Public members	
Status	Failed
ValidationResult	{Sds.Core.Validation.EntityValidationResult}
HasViolations	false
Violations	{Sds.Core.Validation.Violation[0]}
Result	{Sds.Christmas.Domain.Model.UserAccount}
base	{Sds.Christmas.Domain.Model.UserAccount}
Email	"none@none.com"
Firstname	"Daniel"
Lastname	"Wertheim"
Password	"ilovesnow"
Username	"dannyboy"

Service engine

It's implemented exactly in the same way as *ClientEngine* and it is only a Singleton resource that contains resources that are bound to the Services context. Currently the only member is the IoC-container, which now is *ServiceComponentContainer* instead of *ClientComponentContainer*.

Code – ServiceComponentContainer

```
namespace Sds.Christmas.Services
{
   public class ServiceComponentContainer : StructureMapComponentContainer
   {
      protected override void BootstrapContainer()
      {
         Container.Configure(x => x
            .ForRequestedType<EfEntityContextBuilder>()
            .CacheBy(InstanceScope.Singleton)
            .TheDefault.Is.OfConcreteType<EfEntityContextBuilder>()
            .WithCtorArg("connectionStringName").EqualTo("Sds.Christmas")
            .OnCreation(builder => builder.MapEntities<IChristmasEntityMapping>()));

         Container.Configure(x => x
            .ForRequestedType<EfEntityContext>()
            .CacheBy(InstanceScope.PerRequest)
            .TheDefault.Is.ConstructedBy(() => Resolve<EfEntityContextBuilder>().CreateContext())
            .OnCreation<EfEntityContext>(ec => {
               ec.ContextOptions.LazyLoadingEnabled = true;
               ec.ContextOptions.ProxyCreationEnabled = true; }));

         Container.Configure(x => x
            .ForRequestedType<IEntityStore>()
            .CacheBy(InstanceScope.PerRequest)
            .TheDefault.Is.OfConcreteType<EfEntityStore>());
      }
   }
}
```

The first block of *Container.Configure* is responsible for configuration of how my *EfEntityContextBuilder* should be constructed. I have selected to construct it as a Singleton (CacheBy(InstanceScope.Singleton)). This, since it is a builder/factory for *EfEntityContexts*. The builder contains information that is the same for all the *EfEntityContext* within the Christmas domain, e.g. information about the entity mappings. For constructor arguments I pass a connection string name and when the builder is requested the first time, *MapEntities* will be invoked. This method will auto register all entity mapping classes that exists in the app-domain and that implements the interface *IChristmasMapping*.

The second block is responsible for the configuration of how *EfEntityContext* instances should be created. They are created using a factory method on the builder that was configured in block one. *EfEntityContext* - Is injected into the *EfEntityStore* that is configured below. Since each context represents a Unit of work, we shall get a new instance for each request, which is defined by the "CacheBy(InstanceScope.PerRequest)" statement.

The third and last block is responsible for the configuration of how *EfEntityStore* instances should be created. Since it is a wrapper around the *EfEntityContext*, each request to the IoC-container should generate a new instance.

So in theory you now could query the *ServiceComponentContainer* for instances of: *EfEntityContextBuilder*, *EfEntityContext* and *EfEntityStore*. In my solution I access it via the *ServiceEngine:*

```
var entityStore = ServiceEngine.Instance.IoC.Resolve<IEntityStore>())
```

Basically you could do it like this:

```
var container = new ServiceComponentContainer();
var es = container.Resolve<IEntityStore>();
```

This will automatically resolve an *EfEntityContext* and inject it into the *EfEntityStore* which is the default concrete implementation of *IEntityStore*. When the container is constructing the *EfEntityContext* it will resolve the singleton instance of *EfEntityContextBuilder* and call the factory method *CreateContext*.

EfDataContractSerializerAttribute

I was having trouble with serialization of my entities over WCF using net.tcp binding when letting the Entity framework's ObjectContext create dynamic proxies for me (ContextOptions.ProxyCreationEnabled = true). To solve the problem I had to extend the DataContractSerializerOperationBehavior and ensure that it provide my dynamically generated entity proxies to the DataContractSerializer as *"Known types"*.

I also had to create a custom attribute that I could mark my methods (service operations) with, so that my custom *EfDataContractSerializerOperationBehavior* is invoked "only when needed".

It was as simple as that. I also created a class that listens for the AppDomain.AssemblyLoad event and keeps track of the dynamic entity proxy types, to be registered as known types.

One example where this is used is in my WishlistService, where I have a method that lets you query for wishlists by username. The object graph will then contain three different types of dynamically generated entities: UserAccount, Wishlist and Wish. For this to work, the only thing I have to do is to decorate the service-operation with my custom attribute: *EfDataContractSerializerAttribute*

Code – IWishlistService

```
[ServiceContract]
public interface IWishlistService : IService
{
    [OperationContract]
    [EfDataContractSerializer]
    ServiceResponse<Wishlist> SetupNewWishlist(Wishlist wishlist);

    [OperationContract]
    [EfDataContractSerializer]
    ServiceResponse<Wishlist[]> GetWishlistsForUsername(string username);
}
```

Eager loading

In the implementation of *GetWishlistsForUsername* in my *BasicWishlistService* I make use of eager loading. In this method I want to return the wish list's with their associated user account and their contained wishes. Since I'm using this over WCF I want to eager load this complete object graph and can't rely on lazy loading. To accomplish this, you have to invoke Include on the ObjectQuery<T> and pass a string that defines the member names in the graph to include. In my *EfEntityStore* I have wrapped this functionality so that you just can pass these arguments to the method instead of having to chain several Include statements.

Code – WishlistService – GetWishlistsForUsername

```
public virtual ServiceResponse<Wishlist[]> GetWishlistsForUsername(string username)
{
   var response = new ServiceResponse<Wishlist[]>();

   using (var entityStore = ServiceEngine.Instance.IoC.Resolve<IEntityStore>())
   {
      response.Try(
         () => response.Result = entityStore.Query<Wishlist>("WishedBy",
"Wishes").GetForUsername(username).ToArray());
   }

   return response;
}
```

Code – ObjectQuery for Eager loading

```
public IQueryable<T> Query<T>(params string[] includes) where T : class, IEntity
{
   if (includes == null || includes.Length < 1)
      return Query<T>();

   return includes.Aggregate(EfEntityContext.EntitySet<T>() as ObjectQuery<T>, (current, include) =>
current.Include(include));
}
```

In the first code block above you can see that I'm using the *Query-overload* that lets me specify the member names to eager load. I just say: "When you are executing the query, ensure that *WishedBy* and *Wishes* are populated immediately". One can argue that strings in this scenario are ugly and that it would have been nicer with lamdas. The strings are quite compact and effective in this scenario, and if you are nervous about getting errors as a side effect from refactoring where the strings aren't kept in sync with the member name, you probably lack unit tests and really should start using them.

WCF Service host

For simplifying the demo and tests I'm hosting my services in-process. The code for this looks like this:

```
namespace Sds.Christmas.Clients.TempClient
{
   class Program
   {
      static void Main(string[] args)
      {
         //Start WCF-services inprocess. (just for simplifying demo).
         //Could also be done easier with InProcFactory from iDesign.
         //You can instead start the exe in the Sds.Christmas.Services.Host
         using (var server = new ServiceHostServer())
         {
            Console.WriteLine("Starting server...\r\n");

            server
               .AddHost<WcfApplicationService>()
               .AddHost<WcfSecurityService>()
               .AddHost<WcfWishlistService>();
```

```
            server.Open();
            …
            var securityService = ClientEngine.Instance.IoC.Resolve<ISecurityService>();
            …
        }
    }
  }
}
```

Note, that I have removed the code that consumes the services. I have just included some example code that resolves the instance that is used to communicate with a service. Remember that the actual implementation for the *ISecurityService* is determined in the configuration of the IoC-container for the client resources, the *ClientComponentContainer*. What's interesting here is that we start three hosts, where each host represents a WCF-service. The *ServiceHostServer* is just a helper that lets me control the hosts as a unit so that I can open and close them all with one line of code, e.g. "*server.Open();*". The Add method is just adding new instances of the *System.ServiceModel.ServiceHost* class. The hosts are configured using the *App.config*. Since I'm hosting my WCF-services as well as consuming them in-process in the same application, I have both service-configuration and client-configuration in the same *App.config* file.

Code – WCF-service & client configurations

```
<system.serviceModel>
  <client>
    <endpoint name="ApplicationService"
      address="net.tcp://localhost:9000/ChristmasServices/ApplicationService"
      binding="netTcpBinding"
      contract="Sds.Christmas.Services.IApplicationService" />
    <endpoint name="WishlistService"
      address="net.tcp://localhost:9001/ChristmasServices/WishlistService"
      binding="netTcpBinding"
      contract="Sds.Christmas.Services.IWishlistService" />
    <endpoint name="SecurityService"
      address="net.tcp://localhost:9002/ChristmasServices/SecurityService"
      binding="netTcpBinding"
      contract="Sds.Christmas.Services.ISecurityService" />
  </client>

  <services>
    <service name="Sds.Christmas.Services.Wcf.WcfApplicationService">
      <host>
        <baseAddresses>
          <add baseAddress = "net.tcp://localhost:9000/ChristmasServices/ApplicationService" />
        </baseAddresses>
      </host>
      <endpoint binding="netTcpBinding" contract="Sds.Christmas.Services.IApplicationService" />
    </service>

    <service name="Sds.Christmas.Services.Wcf.WcfWishlistService">
      <host>
        <baseAddresses>
          <add baseAddress = "net.tcp://localhost:9001/ChristmasServices/WishlistService" />
        </baseAddresses>
      </host>
      <endpoint binding="netTcpBinding" contract="Sds.Christmas.Services.IWishlistService" />
    </service>

    <service name="Sds.Christmas.Services.Wcf.WcfSecurityService">
      <host>
        <baseAddresses>
          <add baseAddress = "net.tcp://localhost:9002/ChristmasServices/SecurityService" />
        </baseAddresses>
      </host>
      <endpoint binding="netTcpBinding" contract="Sds.Christmas.Services.ISecurityService" />
    </service>
  </services>

  <behaviors>
    <serviceBehaviors>
      <behavior>
        <serviceMetadata httpGetEnabled="false"/>
        <serviceDebug includeExceptionDetailInFaults="false" />
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

WCF Client proxies

When I want the client to consume the hosted WCF-services I let the IoC-container (*ClientComponentContainer*) return an instance of my *WcfServiceClientProxy*. The proxy is just extending the *System.ServiceModel.ClientBase<T>* and adds some code that assumes that if no specific endpoint configuration name is specified, a default is used, which is build by using the name of the interface that is used for the generic type parameter. If the interface is named according to standard naming conventions, with a leading "I", it's removed. Hence "*ISecurityService*" → "*SecurityService*". If you want multiple endpoint configurations for a service, you have to construct the *WcfServiceClientProxy* instance by using the constructor that takes an argument for the endpoint configuration name.

Code – WcfServiceClientProxy

```
namespace Sds.Core.SvcModel.Wcf
{
  [DebuggerStepThrough()]
  public class WcfServiceClientProxy<T> : ClientBase<T>
    where T : class
  {
    public static readonly string DefaultEndpointConfigurationName;

    static WcfServiceClientProxy()
    {
      //Asume that the default endpointconfigurationname is the name of the interface
      //but remove an leading "I" if it exists.
      var serviceType = typeof (T);
      DefaultEndpointConfigurationName = serviceType.Name[0] == 'I'
        ? serviceType.Name.Substring(1) : serviceType.Name;
    }

    public WcfServiceClientProxy()
      : this(DefaultEndpointConfigurationName)
    {
    }

    public WcfServiceClientProxy(string endpointConfigurationName)
      : base(endpointConfigurationName)
    {
    }

    public T ServiceChannel
    {
      get { return Channel; }
    }
  }
}
```

So when I in the client use the IoC-container:

```
var securityService = ClientEngine.Instance.IoC.Resolve<ISecurityService>();
```

I will get an instance of “WcfServiceClientProxy<ISecurityService>”, which will communicate with the in-process hosted WCF-services at the URL: “`localhost:9002/ChristmasServices/SecurityService`”. As I described in earlier chapter, I ensure that the proxy is closed by letting the *ServiceExecutor.Execute* method wrap and invoke an *Action/Func<T>* and after the invoke of the delegate, check if the passed service is an *ICommunicationObject*; if true, it will be closed. You can read more about this above.

Validation of entities

The validation of my entities are performed using validation attributes found under the namespace System.ComponentModel.DataAnnotations. Is use these attributes to perform simple data validation, like ensuring that fields has values; that the values are in the correct range; that values conform to certain formats, e.g. emails. For more advanced and custom validation I use custom Entity validators.

Simple data validations

I have made the design decision to put my validation messages in plain *resx-files*, so that I get an easy way of making them localizable. When doing so you need to provide the ErrorMessageResourceName and the ErrorMessageResourceType. The code for setting up a decorated entity is shown "partially" below. I have already shown the complete entity under the "User account entity" section above, hence I will not show all the code for this entity here.

Code – Simple data validations using validation attributes

```
[Required(
    AllowEmptyStrings = false,
    ErrorMessageResourceName = "UsernameIsRequired",
    ErrorMessageResourceType = typeof(ValidationMessages))]
[StringRange(MinLength = 5,
    MaxLength = 20,
    ErrorMessageResourceName = "UsernameHasInvalidLength",
    ErrorMessageResourceType = typeof(ValidationMessages))]
[DataMember]
public virtual string Username { get; set; }
```

The Username is required and needs to be between 5 and 20 characters in length. Two validation messages are defined: *UsernameIsRequired* and *UsernameHasInvalidLength*.

```
[Required(
    AllowEmptyStrings = false,
    ErrorMessageResourceName = "EmailIsRequired",
    ErrorMessageResourceType = typeof(ValidationMessages))]
[Email(
    ErrorMessageResourceName = "EmailHasInvalidFormat",
    ErrorMessageResourceType = typeof(ValidationMessages))]
[StringLength(
    150,
    ErrorMessageResourceName = "EmailHasInvalidLength", ErrorMessageResourceType = 
typeof(ValidationMessages))]
[DataMember]
public virtual string Email { get; set; }
```

The Email is required and can't be longer than 150 characters. It also must conform to the format of a valid email address. Three validation messages are defined: *EmailIsRequired, EmailHasInvalidFormat* and *EmailHasInvalidLength*.

The validation messages are stored in a resx-file located in the same assembly as the entities.

ValidationMessages.resx | UserAccount.cs | UserAccountValidator.cs | EntityValidator.cs | Program.cs

Strings ▾ | Add Resource ▾ | Remove Resource | Access Modifier: Public

Name	Value
EmailHasInvalidFormat	Provided email is not of a valid format.
EmailHasInvalidLength	Email can be max 150 characters long.
EmailIsAllreadyTaken	Email is allready taken.
EmailIsRequired	You must provide an email.
FirstnameHasInvalidLength	Firstname can be max 100 characters long.
FirstnameIsRequired	You must provide a firstname.
LastnameHasInvalidLength	Lastname can be max 100 characters long.
LastnameIsRequired	You must provade a lastname.
PasswordHasInvalidLength	Provided password must have a lenght between 5 and 20 characters.
PasswordIsRequired	You must provide a password.
TitleHasInvalidLength	Title can be max 250 characters long.
TitleIsRequired	You must provide a title.
UrlHasInvalidLength	Url can be max 250 characters long.
UsernameHasInvalidLength	Provided username must have a length between 5 and 20 characters.
UsernameIsAllreadyTaken	Username is allready taken.
UsernameIsRequired	You must provide an username.

Custom Entity validators

Ok, now it's time to put the validation attributes in use. Remember the validation attributes that was applied to members on the User account entity. These are used by a Validator to generate ValidationResult items (both of them are found in the *System.ComponentModel.DataAnnotations* namespace).

EntityValidator

I have put together an *EntityValidator* that I use to validate my entities. It does not contain any state so it can be used for the same type of entity, over and over again. The responsibility is to produce an *EntityValidationResult*, which is a custom object that I'm using to contain the results of the validation of an entity. I'm not using the ValidationResult object from the *DataAnnotations* namespace, since I got problems with it when trying to serialize it over WCF. I checked it out, and it is not marked as Serializable, hence I produced a custom Violation object which simply represents a broken business rule. The Violation should be assigned a message and the name(s) of the members that generated the Violation.

Code - Violation

```
[Serializable]
public class Violation
{
   public string[] MemberNames { get; set; }

   public string ErrorMessage { get; set; }

   public Violation(string errorMessage, IEnumerable<string> memberNames = null)
   {
      ErrorMessage = errorMessage;
      var tmpMemberNames = new List<string>();

      if (memberNames != null)
         tmpMemberNames.AddRange(memberNames);

      MemberNames = tmpMemberNames.ToArray();
   }
}
```

The violations are generated in the *EntityValidator's* method *Validate*. This method simply uses Microsofts Validator to generate ValidationResults from the validation attributes. After that, Validate calls an overridable method *OnCustomValidation* that can be used to let subclasses of the EntityValidator to generate ValidationResults that are results of a custom validation, e.g. check if a username already has been taken.

I like to put the responsibility of the validation in a custom validator that is designed specifically for a certain entity, e.g. *UserAccountValidator*. This custom validator is then to be seen as part of the model, since it contains logic/rules for entities in the model. The separation of the validation rules from the entity is something one can argue about; since the validation is logic that is tied to a certain entity and therefore should be but in the entity itself. I like it since it's more clear to me to have a class that only contains validation logic and isn't a mix with other logics as well.

Code – EntityValidator

```
/// <summary>
/// Lets you perform validation on Entities.
/// Extend it and make it customizable for entities
/// that needs custom validation by overriding <see cref="OnCustomValidation"/>.
/// </summary>
/// <typeparam name="T"></typeparam>
public class EntityValidator<T> : IEntityValidator<T>
   where T : IEntity
{
   /// <summary>
   /// Validates the specified entity.
   /// </summary>
   /// <param name="entity">The entity.</param>
   /// <returns></returns>
   public EntityValidationResult Validate(T entity)
   {
      var validationResults = new List<ValidationResult>();
      var validationContext = new ValidationContext(entity, null, null);
      var isValid = Validator.TryValidateObject(entity, validationContext, validationResults, true);
      var customValidationResults = OnCustomValidation(entity);
```

```
        if(customValidationResults != null)
            validationResults.AddRange(customValidationResults);

        //Since ValidationResult is not marked as Serializable
        //a custom object "Violation" is used instead.
        var violations = validationResults.Select(ViolationMapper.ToViolation);

        return new EntityValidationResult(violations.ToList());
    }

    /// <summary>
    /// Override and implement to add custom validation of your entities.
    /// </summary>
    /// <param name="entity">The entity.</param>
    /// <returns></returns>
    protected virtual IEnumerable<ValidationResult> OnCustomValidation(T entity)
    {
        return null;
    }
}
```

An example of a custom entity validator where I override the OnCustomValidation method, is my UserAccountValidator. It only has some checks that ensures that the Email and Username isn't already represented in the application.

Code – UserAccountValidator

```
public class UserAccountValidator
    : EntityValidator<UserAccount>
{
    private readonly IEntityStore _entityStore;

    public UserAccountValidator(IEntityStore entityStore)
    {
        _entityStore = entityStore;
    }

    protected override IEnumerable<ValidationResult> OnCustomValidation(UserAccount entity)
    {
        var violations = new List<ValidationResult>();

        if (CheckIfEmailIsTaken(entity))
            violations.Add(new ValidationResult(ValidationMessages.EmailIsAllreadyTaken));

        if (CheckIfUsernameIsTaken(entity))
            violations.Add(new ValidationResult(ValidationMessages.UsernameIsAllreadyTaken));

        return violations;
    }

    private bool CheckIfUsernameIsTaken(UserAccount userAccount)
    {
        return _entityStore.Query<UserAccount>().UsernameIsTaken(userAccount.Username);
    }


 private bool CheckIfEmailIsTaken(UserAccount userAccount)
```

```
    {
        return _entityStore.Query<UserAccount>().EmailIsTaken(userAccount.Email);
    }
}
```

The resulting class EntityValidationResult only contains the violations that the validators generate. In the example solution I incorporate this within my responses from my services, in the ServiceResponse class. If the EntityValidationResult has violations, the status of the service response will indicate “Failed”. So the consumer can check the status, and if “Failed”, it can investigate if there are any violations.

Code – EntityValidationResult

```
[Serializable]
public class EntityValidationResult
{
    public Violation[] Violations { get; private set; }

    public bool HasViolations
    {
        get { return Violations.Length > 0; }
    }

    public EntityValidationResult(IEnumerable<Violation> violations = null)
    {
        Violations = (violations ?? new List<Violation>()).ToArray();
    }
}
```

In the provided example solution I check my service responses and the violations and just output them to the console.

Code – Consuming the entity validation results

```
private static void IfFailedServiceOperationInformUser(IServiceResponse serviceResponse)
{
    if (serviceResponse.Status == ServiceResponseStatuses.Failed)
        Console.WriteLine("Service failed!");

    if (!serviceResponse.ValidationResult.HasViolations)
        return;

    Console.WriteLine("Violations are:");
    foreach (var violation in serviceResponse.ValidationResult.Violations)
    {
        foreach (var member in violation.MemberNames)
            Console.WriteLine("\t+" + member);

        Console.WriteLine("\t" + violation.ErrorMessage);
    }

    Console.WriteLine();
}
```

```
Trying to create an account with the same data...
Service failed!
Violations are:
        Email is allready taken.
        Username is allready taken.
```

The end…

That's all. I hope you have found something useful. Please remember. The code is built using techniques/products that are in beta and CTP mode. I also don't think that there is a go live license yet on the CTP2 addition to Entity framework 4, so you might want to check this with Microsoft before going live….

Have fun!

//Daniel