Contents

- Install DevForce Code First support
- Apply DevForce AOP attributes
- DataContract Attributes
- Persisted properties
- Persisted simple properties
- Persisted navigation properties
- Add validation with attributes
- Add property interceptors
- Map properties explicitly
- Guid keys
- Default constructor required
- Unmapped properties
- Non-public properties
    - The only non-public property access modifier is internal
    - Make internal properties visible

You **write DevForce Code First entities using DevForce "Aspect Oriented Programming" (AOP)**. The MS Build process is modified to rewrite classes marked with DevForce AOP attributes when you install the DevForce Code First NuGet package. During class re-write, DevForce injects entity infrastructure into the class and replaces persisted properties with DevForce implementations.

Install DevForce Code First support

You should first install the DevForce Code First NuGet package to the project which will hold your code first model.

If you are working with Silverlight or a mobile application, you should also install the package to your "linked" client project.

The package will add a reference to IdeaBlade.AOP, and if the DevForce "core" assemblies are not present the DevForce 2012 Core NuGet package is automatically installed too.

The package will also install the necessary dependencies based on your environment:

- The EntityFramework NuGet package is installed to a .NET 4.5 project to add support for Entity Framework 5 or above.
- PostSharp from SharpCrafters is installed for every environment. PostSharp will display some configuration prompts during installation: see Installing PostSharp 3.0 for information.

Both the DevForce and PostSharp packages modify the MSBuild targets for the projects they are installed to. If you peek into the project file you'll see an additional import for PostSharp, and in a .NET 4.5 project another one for DevForce. For example, a .NET 4.5 C# project might look something like this:

```
<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
<Import Project="..\packages\PostSharp.3.0.31\tools\PostSharp.targets"
   Condition="Exists('..\packages\PostSharp.3.0.31\tools\PostSharp.targets')" />
<Import Project="..\packages\IdeaBlade.DevForce.Aop.7.2.0\tools\IdeaBlade.DevForce.Common.targets"
   Condition="Exists('..\packages\IdeaBlade.DevForce.Aop.7.2.0\tools\IdeaBlade.DevForce.Common.targets')" />
```

The PostSharp targets file ensures that your Code First entities are re-written and is required for Code First models in all environments.

The DevForce targets file is used in .NET 4.5 projects to fire the *EntityModelMetadataDeploy* task which builds the metadata file for your model. The metadata file is generated in the .NET 4.5 project (which also references EntityFramework) and linked into the model project in other environments.

Apply DevForce AOP attributes

You apply DevForce AOP attributes to the classes that should be rewritten.

There are only two DevForce AOP attributes to learn: ***ProvideEntityAspect*** and ***ProvideComplexAspect***. *ProvideEntityAspect* identifies **entity** classes for rewrite. The *ProvideComplexAspect* identifies **Complex Type** classes for rewrite.

You can either decorate the class directly or decorate one of its base types. You cannot do both. The AOP attribute can only appear on one of the classes in the inheritance chain. You'll get a clear, build error if the attribute appears more than once in the chain.

Here are some examples of valid class declarations:

```
[ProvideEntityAspect]
public class Category { ... }
public class Product : EntityBase { ... } // no attribute; inferred from EntityBase
[ProvideEntityAspect]
[DataContract(IsReference=true)]
public abstract class EntityBase { ... } // [ProvideEntityAspect] attribute on base class
[ProvideComplexAspect]
public class Address { ... } // a "Complex Type"
```

DataContract Attributes

DevForce uses the WCF *DataContactSerializer* (DCS) to serialize entity classes in n-tier scenarios (e.g., Silverlight and mobile apps), when importing entities from another *EntityManager*, and when storing entity data locally in offline scenarios.

So far, none of our example entity classes are decorated with *DataContract* attributes (although we do have one on the *EntityBase* class). We often omit these attributes from sample model entity classes. However, if you examine the entity classes generated by DevForce, you'll notice that they are always decorated with *[DataContract(IsReference=true)]* and all of their properties are marked with *[DataMember]*.

Explicit *DataContract* markup is not required if every property of the class is completely public and can always be serialized. Many developers prefer to omit these attributes because they are noisy clutter that reduces our ability to quickly grasp what the class does.

Beware: if any property is non-public in any way (and a property with an internal setter is not entirely public), you will have to apply explicit *DataContract* markup. If you neglect to add *DataContract* attributes, you'll get unexpected behavior such as property updates that don't write through to the database.

If you want to be absolutely safe, follow the DevForce generated class pattern by adding *DataContract* attributes to your classes. We cover this subject in a little more detail below.

Persisted properties

"Persisted properties" return values that reside in the database.

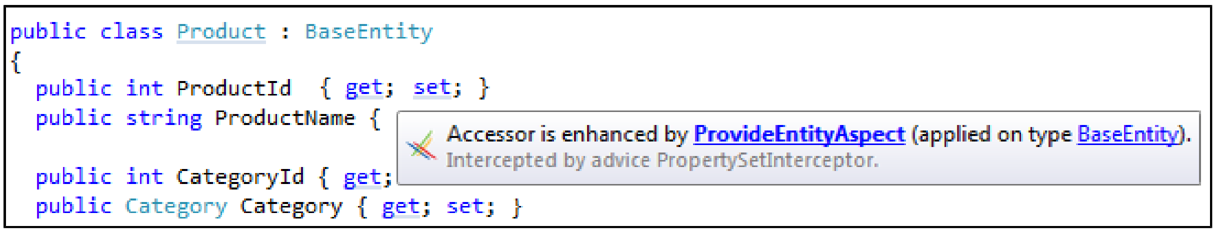
Here are two entity class definitions, *Category* and *Product*, to which we'll refer in this discussion.

```

[ProvideEntityAspect]
public class Category
{
  public int CategoryId { get; set; }
  public string CategoryName { get; set; }
  public RelatedEntityList<Product> Products { get { return null; } }
}
[ProvideEntityAspect]
public class Product
{
  public int ProductId { get; set; }
  public string ProductName { get; set; }
  public int CategoryId { get; set; } // Foreign key for "Category"
  public Category Category { get; set; }
}
```

Each class is mapped to the database entirely by convention. *Category*, for example, is mapped to a "Categories" table which has a primary key column named *CategoryId*. Entity Framework figured this out without our help.

Visual Studio IntelliSense reveals that the properties are rewritten by PostSharp; note the light grey lines under the *get* and *set*.



```
public class Product : BaseEntity
{
  public int ProductId  { get;  set; }
  public string ProductName {
  public int CategoryId { get;
  public Category Category { get; set; }
```

Accessor is enhanced by ProvideEntityAspect (applied on type BaseEntity).
Intercepted by advice PropertySetInterceptor.

Persisted simple properties

Category.CategoryName is one of the simple properties; it returns a string. Note that it is an “auto property” with no getter or setter. If the runtime class were actually implemented as we see here, *Category* would be unable to notify a UI when someone set the name of a category.

Surprisingly, the *Category* class does not appear to implement *System.DataComponent.INotifyPropertyChanged* nor does it seem to implement that interface’s *PropertyChanged* event. A listening UI control couldn’t know when a property was reset and therefore would not refresh when the name was changed. Even if the *Category* did implement that interface, the *CategoryName* setter is doing nothing to raise the *PropertyChanged* event. And yet ... it works.

After DevForce AOP rewrites the class with DevForce entity infrastructure, the *Category* class does implement *INotifyPropertyChanged* and the *CategoryName* setter does raise the *PropertyChanged* event.

The example properties shown above are written as "auto-properties", bare property definitions without implementations. You could implement a simple property with a backing field if you wished as in this example:

```
public string ProductName
{
  get { return _productName; }
  set {
    if (_productName != value) {
       _productName = value;
      // DevForce automatically raises PropertyChanged for ProductName
      // RaisePropertyChanged is the developer's custom method to fire the PropertyChanged event
      RaisePropertyChanged("SomeOtherProperty"); // notify UI that this related property has changed as well
    }
   }
}
private string _productName;
```

DevForce AOP wraps your implementation, calling the original code from within the DevForce property interception logic. Whatever you put inside your property *get* and *set* methods will be honored. For example, setting *ProductName* notifies the UI to update controls bound to both *ProductName* and *SomeOtherProperty*. You can put breakpoints on your custom code and debug as you would similar properties elsewhere in your application. We think Property Interceptors are a better way to inject logic into your properties but you are welcome to this approach as well.

Delegation to your implementation is a feature of *simple properties only*. *Navigation property* implementations are completely replaced by DevForce code. Do not put custom code inside navigation properties.

Persisted navigation properties

An entity’s navigation properties return other related entities. The two navigation properties of interest in our example are *Product.**Category*** and *Category.**Products***.

```
// Product ...
public Category Category { get; set; }  // reference navigation
public int CategoryId {get; set;} // foreign key for related Category
// Category ...
public RelatedEntityList<Product> Products { get { return null; } } //collection navigation
// ...
```

Category and *Product* have a one-to-many relationship. A *Product* has a parent category instance retrieved by means of its *Category* reference navigation property. The *Product.**CategoryId*** is the foreign key property that helps EF relate a product to its parent *Category* entity.

On the other hand, a *Category* instance can retrieve its many products by means of its *Products* collection navigation property. The two navigation properties are the inverse of each other.

Notice that the navigation properties lack substantive *get* and *set* implementations. Yet if we invoke, say, *Product.**Category***, we expect DevForce to retrieve the product’s category from cache if it can find it and from the database if there is no related category in cache. DevForce can do this because it replaces your implementation of a navigation property with its own.

Notice that neither navigation property is virtual. You do not write virtual properties with DevForce Code First (as you must do with Entity Framework Code First). DevForce doesn’t need virtual properties because it rewrites these properties with the necessary navigation logic rather than trying to override these properties in a derived proxy class.

Look again at the implementation of *Category.**Products***.

```
public RelatedEntityList<Product> Products { get { return null; } }
```

Notice that it returns a *RelatedEntityList<Product>*. Entity Framework Code First only requires that the return type be an *ICollection<T>*. DevForce honors that as well; we could have returned *ICollection<Product>*

```
public ICollection<Product> Products { get { return null; } }
```

In practice, the actual object that DevForce returns is a *RelatedEntityList<Product>* which implements *ICollection<Product>* as you would expect. Most developers prefer to declare that the property returns the more strongly typed *RelatedEntityList<T>* because it is more convenient to have access to the full *RelatedEntityList* API when you're working with this property. You are free to be more opaque if you prefer.

Notice also that *Category.**Products*** is a "ReadOnly" property; it has a *getter* but no *setter*. The developer chose to write the navigation property without a *setter* rather than as an auto-property for a good reason. The developer knows that consumers of a *Category* shouldn't be able to ***set*** the *Products* collection; they can read the collection, maybe add products to the collection, but they should not replace the collection itself. By omitting the setter, the developer ensures that no one will set a category's *Products* collection by accident. ReadOnly collection navigation properties like this one are a good practice.

Add validation with attributes

You can decorate the properties of an AOP entity with validation attributes in much the same way as you do for generated entities. The only difference – a happy difference – is that you can add the validation attributes directly to the properties in your class: you don't need a metadata "buddy" class.

Suppose the category name is required and a maximum length of 15 characters. We could enforce these constraints by adding the standard .NET validation attributes to the *Category.**CategoryName*** property:

```
[Required]
[StringLength(15)]
public string CategoryName { get; set; }
```

With Code First entities, you'll often want to stick with the .NET data annotations, since many do double duty and are used for validation, and more importantly, database mapping. The DevForce verifiers cannot be used for mapping and cover only validation within the application. Also be sure not to overlap a .NET attribute and DevForce verifier which does the same thing, since you'll get duplicate validation messages.

Of course you can also write your own custom verifiers and verifier attributes and annotate your properties in the same way.

Add property interceptors

The re-written properties of AOP entities participate in the DevForce property interception system.

You could induce the *Product.ProductName* property to return its value in upper case by adding the following method to the *Product* class.

```
[AfterGet("ProductName")]
internal void UppercaseNameAfterGet(PropertyInterceptorArgs<Product, String> args)
{
  if (null != args.Value)
   {
      args.Value = args.Value.ToUpper();
   }
}
```

This contrived "get interceptor" returns an upper-cased version of the *ProductName* value. Notice that it can be non-public … and probably should be as there is no good reason for application code to call this method.

DevForce discovers the interceptors by reflection. The method could be *private* in a full .NET model but not in Silverlight where private reflection is forbidden. Marking it *internal* gives DevForce a chance to find it in Silverlight when you make your model assembly visible to DevForce.

Map properties explicitly

The Entity Framework Code First mapping mechanisms are at your disposal. The example code relied entirely on Entity Framework Code First naming conventions for mapping the Category class to the "Categories" table. We didn't have to tell EF that *CategoryId* is the primary key property corresponding to an integer "CategoryId" column in the "Categories" table.

But we could have made the mapping explicit either declaratively by applying Entity Framework’s mapping attributes or imperatively through configuration with the Entity Framework Fluent API. Here is the same *Category* class, decorated with mapping attributes

```
[ProvideEntityAspect]
[Table("Categories")]
public class Category
{
   [Key]
   [Column("CategoryId")]
  public int CategoryId { get; set; }
   [Column("CategoryName")]
  public string CategoryName { get; set; }
}
```

The *Table*, *Key*, and *Column* attributes were all unnecessary as the class can be mapped by convention without them. But we might apply these attributes if our class and database table defied those conventions, as we see in this example:

```
[ProvideEntityAspect]
[Table("Product.Category")]
public class Category
{
   [Key]
   [Column("CategoryID")]
  public int CategoryKey { get; set; }
   [Column("CategoryName")]
  public string Name { get; set; }
}
[ProvideEntityAspect]
public class Product
{
  public int ProductId { get; set; }
  public string ProductName { get; set; }
   [ForeignKey("Category")] // because “CategoryId" != Category.CategoryKey
  public int CategoryId { get; set; } // Foreign key for "Category"
  public Category Category { get; set; }
}
```

What a mess!

- The *Category* class maps to a table named “Product.Category”
- The *Name* property of the *Category* class maps to the “CategoryName” column
- The *Category* primary key property is named *CategoryKey* but the matching column is “CategoryID”
- *Product*’s foreign key connecting it to its parent *Category* entity has yet a third spelling, *CategoryId* (small-d). The *ForeignKey* attribute informs Entity Framework and DevForce that this is the foreign key property supporting the *Category* navigation property.

It’s a typical mess and, sadly, it could be worse.

Entity Framework Code First mapping is a big topic, beyond our ability to adequately cover in the DevForce Resource Center. Please review external sources, starting with Microsoft’s ADO.NET Entity Framework website, to learn more.

Guid keys

Guid keys are a common approach to client-side key specification. They are particularly useful in offline scenarios in which the user may create a new record but can't reach the database to save it. The application can serialize the entity to a local file temporarily until the server becomes available; DevForce supports this approach with temporary integer keys. But the developer may prefer the certainty of an assigned key that is both unique and permanent.

EF does not assign *Guid* key values by default. The developer takes care of that. While key assignment can occur anywhere, many developers will set the key in the entity constructor as seen here:

```
public Supplier()
{
   SupplierKey = Guid.NewGuid(); // ToDo: use a sequential Guid generator
}
public Guid ProductId { get; set; }
```

The “ToDo” reminds us that it is wiser to (a) call a custom generator class instead of directly calling upon the *Guid* class and (b) to generate a sequential *Guid* (a "*Guid.Comb*") instead of returning the more random result of *Guid.NewGuid()* for database performance reasons. You can write a sequential *Guid* generator yourself or use Mark Miller's implementation.

Guid keys are also popular where database replication could result in integer key collisions; different databases might assign the same integer key values. You could choose *Guid*s to forestall key value conflicts and also let EF assign these values with what appears to be a sequential *Guid* generator.

If you want Entity Framework to assign the keys, you can apply a *DatabaseGenerated* attribute to the property with the *DataBaseGeneratedOption.Identity* as follows:

```
[DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public Guid ProductId { get; set; }
```

You do not need ... or want ... the constructor assignment from the example above.

Default constructor required

Both Entity Framework and DevForce require a default constructor in order to materialize your entities from query results. You don't have to write a constructor ... because you have one implicitly. Just don't block it completely by writing a constructor with parameters and neglecting to add a parameterless constructor.

Your default constructor **does not have to be public**. You can hide it from application developers with a non-public modifier. Here are some examples:

```
public class Category {...} // no constructor defined; implicit public default constructor used by everyone
public class Product
{
  public Product() {...} //  Explicit public default constructor used by everyone
}
public class Shipper
{
  public Shipper(string Name) {...} // FAIL! The class lacks a default constructor
}
public class Order
{
  public Order(string Name="New Order") {...} // App devs can use as can DevForce and EF because of optional parameter
}
public class OrderDetail
{
  internal OrderDetail() { ... } // DevForce and EF use to materialize OrderDetails from query results
  public OrderDetail(Order parentOrder) // App devs must use; forces them to assign a parent Order.
}
```

Silverlight entity classes have only two choices for the default constructor - public or internal.

Unmapped properties

Many entity classes will be enriched with additional properties that express some business logic or make the entity easier to work with. We neither retrieve nor store such property values in the database. These properties are “unmapped” and we have to tell both DevForce AOP and EntityFramework that they shouldn’t rewrite these properties nor look for corresponding data in the database. The easy way to make these points clear is to annotate the property with the ***NotMapped*** attribute.

For example, suppose we add an *IsProductOfTheDay* property to the *Product* class. The value of *IsProductOfTheDay* is calculated; it is not retrieved from the *Products* table and it can’t be saved there either. We don’t want DevForce AOP to rewrite it. We don’t want EF to try to map it to the database. So we add the *NotMapped* attribute. The property might be written as follows:

```
[NotMapped]
public bool IsProductOfTheDay {
 get {
  return this.ProductID == GetProductOfTheDay().ProductID;
 }
}
```

Non-public properties

You can limit access to your persisted properties if you wish ... at the cost of adding WCF *DataContract* attributes.

Most DevForce applications are n-tier and all Silverlight applications are n-tier. They serialize entities across process and network boundaries. No markup is necessary when the entity classes conform to WCF requirements for default serialization. The following model classes do **NOT** require markup:

```
public class Product
{
   public int ProductId {get; set;}
   public string ProductName {get; set;}
   public int SupplierId {get; set;}
   public Supplier Supplier {get; set;}
}
public class Supplier
{
   public int SupplierId {get; set;}
   public string CompanyName {get; set;}
   public RelatedEntityList<Product> Products {get {return null; }}
    [NotMapped]
   public string UnmappedProperty {get; set;}
}
```

Note that all of the persisted simple properties (e.g., *Product.ProductName*) and reference navigation properties (*Product.Supplier*) have public getters and setters. Collection navigation properties (*Supplier.Products*) can do without setters but their getters must be public.

The *Supplier.UnmappedPoperty* is not persisted but it is serialized and its value is available on client and server.

If you prefer to hide access to some properties, you'll have to annotate the classes with *DataContract* attributes. We've redefined the example model to reduce access to some of the properties.

```
[DataContract(IsReference=true)]
public class Product
{
   // Preclude application code instantiation; require use of CreateProduct factory method
   internal Product () { } // can be private unless used in Silverlight
   public static Product CreateProduct(string secretProductName)
    {
      return new Product { SecretProductName = name };
    }
    [DataMember]
   public int ProductId {get; internal set;} // auto-generated; application code should not set.
    [DataMember]
   internal string SecretProductName {get; set;}
    [DataMember]
   public string ProductName {get; set;}
    [DataMember]
   public int SupplierId {get; set;}
    [DataMember]
   public Supplier Supplier {get; set;}
   public string GetSecretProductName()
    {
      return this.SecretProductName;
    }
}
[DataContract(IsReference=true)]
public class Supplier
{
    [DataMember]
   public int SupplierId {get; internal set;} // auto-generated; application code should not set.
    [DataMember]
   public string CompanyName {get; set;}
    [DataMember]
   public RelatedEntityList<Product> Products {get {return null; }}
    [NotMapped]
   public bool UnmappedProperty {get; set;}
}
[DataSourceKeyName("NonPublicAccessorExample")]
public class ExampleDbContext : DbContext
{
   public ExampleDbContext (string connect) : base(connect) { }
   protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Ignore<EntityAspect>();
        modelBuilder.Entity<Product>().Property(p => p.SecretProductName);
    }
}
```

Notice that:

- The *DataContract* attribute specifies "(IsReference=true)". This is essential to support serialization of entities with circular references. You have a circular reference if two entities have navigation properties to each other ... as is typical in entity models.
- If *any* property has reduced access, the entire class must be marked up.
- We had to map the completely hidden *Product.SecretProductName* property with a custom *DbContext* using the Code First Fluent API. Entity Framework maps the *public* properties with *internal* setters by convention but won't map an entirely non-public property except via the Fluent API.
- You'd have to markup the class if it contains a public property that you **do not want** to serialize. Omitting the *[DataMember]* attribute on *Supplier.UnmappedProperty* excludes it from serialization.

- You can specify either *internal* or *public* access but you cannot specify *protected* or *private*.

The only non-public property access modifier is *internal*

You can make a property *private* in Entity Framework Code First but you can't do so in DevForce. A property must either be *public* or *internal*.

There are two primary reasons for DevForce's more limited choice:

1. DevForce only rewrites properties that are *public* or *internal*. This is our design choice; there is a tiny performance cost to a rewritten property and we think you shouldn't pay that price for private properties. Nor did we think that private properties benefit from rewriting with notification, interception, and validation logic as they can't participate in the binding scenarios where those capabilities are useful.
2. DevForce can not serialize *private* properties of entities in security-restricted applications ... but it can serialize *internal* properties as explained below.

Make *internal* properties visible

Private reflection is not allowed in Silverlight and mobile apps. DevForce cannot serialize/deserialize into private properties of your entities, but can use *internal* properties **if and when** the model assembly is made visible to the serialization libraries.

Add the following attributes to the model project's *AssemblyInfo* file.

```
// Make internal entity properties accessible for serialization/deserialization and querying.
[assembly: InternalsVisibleTo("System.Core,
PublicKey=002400000480000094000000060200000024000052534131000400000100010008d56c76f9e8649383049f383c44be0ec204181822a6c31cf5eb7ef48694
[assembly: InternalsVisibleTo("System.Runtime.Serialization,
PublicKey=002400000480000094000000060200000024000052534131000400000100010008d56c76f9e8649383049f383c44be0ec204181822a6c31cf5eb7ef48694
```

```
'Make internal entity properties accessible for serialization/deserialization and querying.
<Assembly: InternalsVisibleTo("System.Core,
PublicKey=002400000480000094000000060200000024000052534131000400000100010008d56c76f9e8649383049f383c44be0ec204181822a6c31cf5eb7ef48694
<Assembly: InternalsVisibleTo("System.Runtime.Serialization,
PublicKey=002400000480000094000000060200000024000052534131000400000100010008d56c76f9e8649383049f383c44be0ec204181822a6c31cf5eb7ef48694
```

Code First entities in DevForce also contain injected methods to handle serialization and deserialization. These methods - OnSerializing, OnSerialized, OnDeserializing and OnDeserialized - have internal access. If you use *ImportEntities*, faking, or the *EntityCacheState*, you must grant DevForce access to these internal members. The assembly name will differ based on environment.

Silverlight:

```
// Make internal properties accessible to DevForce for cloning.
[assembly: InternalsVisibleTo("IdeaBlade.Core.SL,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100b3f302890eb5281a7ab39b936ad9e0eded7c4a41abb440bead71ff5a31
```

Windows Store:

```
// Make internal properties accessible to DevForce for cloning.
[assembly: InternalsVisibleTo("IdeaBlade.Core.WinRT,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100b3f302890eb5281a7ab39b936ad9e0eded7c4a41abb440bead71ff5a31
```

Windows Phone:

```
// Make internal properties accessible to DevForce for cloning.
[assembly: InternalsVisibleTo("IdeaBlade.Core.WP8,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100b3f302890eb5281a7ab39b936ad9e0eded7c4a41abb440bead71ff5a31
```