Contents

How to **access the *EntityAspect*** from a DevForce AOP entity.

Access the EntityAspect

Code First, AOP entities, and the “POCO lifestyle” encourage you to write entities that appear free of infrastructure concerns. This *Category* class is (almost) all business:

```
[ProvideEntityAspect]
public class Category
{
  public int CategoryId { get; set; }
  public string CategoryName { get; set; }
}
```

However, seasoned developers understand that they must regard such business objects as *entities* on occasion. It will become important, for example, to know if a particular *Category* object has changed.

Change tracking is an entity infrastructure concern. DevForce provides access to entity infrastructure by means of the EntityAspect property. *EntityAspect's* EntityState property indicates whether and how the entity has been changed.

You can plainly see that *EntityAspect* is not a property of the Category class as written. The post-build, re-written Category class does have an *EntityAspect* property, but neither you nor the compiler can see it while writing your application. If you want access to *EntityAspect*, you'll have to get it in one of two ways:

1. By casting to *IEntity*:
   ```
   ((IEntity) someCategory).EntityAspect.EntityState;
   ```
2. By wrapping:
   ```
   EntityAspect.Wrap(someCategory).EntityState;
   ```

The cast works because every re-written AOP entity implements *IdeaBlade.EntityModel.**IEntity***, an interface whose sole member is the *EntityAspect* property.

The static *Wrap(anyObject)* method of the *EntityAspect* class can wrap any kind of object in an instance of *EntityAspect*. The *Wrap()* method understands that AOP entities contain an *EntityAspect* and returns that inner *EntityAspect*.

If the object passed as an argument to *Wrap()* does not implement *IEntity*, *Wrap(someObject)* returns a new *EntityAspect* object whose *Entity* property returns the argument (*someObject*).

Tell EF to ignore the *EntityAspect* type

Skip this advice if you do **not** write your own *DbContext*. You **must** follow this advice if you write a *DbContext*.

All DevForce AOP entities have an *EntityAspect* property through which you gain access to your entity's internal entity capabilities. You may or may not write that property in your entity source code, but it's there after DevForce rewrites the class with DevForce entity infrastructure.

Entity Framework discovers this *EntityAspect* property as it reflects into the types in your entity model. EF sees that the (injected) property returns an *EntityAspect* type which it assumes (by convention) is an *entity* type. Well *EntityAspect* is not an *entity* type and it fails EF validation because it doesn't have a key as an entity must.

Someone has to tell EF to ignore the *EntityAspect* and all properties that return that type. The DevForce default *DbContext* does so automatically. If you only write an *EntityManager* and do **not** write your own *DbContext*, you **don't have to worry about this issue** at all.

But if you do write your own *DbContext*, then you must tell EF to ignore it, using the Entity Framework Code First Fluent API, by overriding *DbContext*'s *OnModelCreating* method as shown here:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
  modelBuilder.Ignore<EntityAspect>();
}
```

Explicit *EntityAspect* properties

You may be comfortable casting or wrapping to access the *EntityAspect*. Some of us prefer readier access to *EntityAspect* ... or at least readier access to some of the members of *EntityAspect*.

You can add an *EntityAspect* property to your entity class:

```
[ProvideEntityAspect]
public class Category
{
  // ...
  public EntityAspect EntityAspect { get { return null; } }
}
```

Now the compiler will let you write application code to access a category's *EntityAspect* directly as you've done in the past.

```
if (someCategory.EntityAspect.EntityState.IsAddedOrModified()) {...}
```

Don't worry about the "null" implementation. Later, while rewriting the *Category* class, DevForce AOP detects your *EntityAspect* property and replaces your getter with an implementation returning the true, injected *EntityAspect* instance.

You might try to apply the *NotMapped* attribute to your *EntityAspect* property, hoping that doing so will relieve you of the requirement to ignore *EntityAspect* in your *DbContext* (see above). Sorry ... that won't work. The *NotMapped* attribute tells Entity Framework Code First to ignore the *EntityAspect* **property**. We need EF to ignore the *EntityAspect* **type**. On the bright side, once EF learns to ignore the *EntityAspect* **type**, it will automatically ignore every **property** that returns *EntityAspect*.

Don't add an *EntityAspect* property

Now that we've shown you how to do add an *EntityAspect* property, we're going to ask you not to do that.

The obvious reason is that adding an *EntityAspect* property to every class is repetitive, tedious and error prone. You can workaround that by writing your own base class and relocating the *EntityAspect* property there so all derived entity classes have it.

```
public class Category : EntityBase    { ... }
public class Product : EntityBase    { ... }
// ... more classes ...
[ProvideEntityAspect]
[DataContract(IsReference = true)]
public abstract class EntityBase
{
  // ...
  public EntityAspect EntityAspect { get { return null; } }
}
```

People do that. We've done it in demos.

But **we strongly discourage it**. In our opinion, *EntityAspect* provides too much access to the DevForce persistence machinery. *EntityAspect.EntityManager*, for example, returns the *EntityManager* to which the entity is attached. With that *EntityManager* you could call *SaveChanges* or *RejectChanges* or add event handlers or perform many other operations that have sweeping effects beyond the scope of the entity in hand.

Such operations really shouldn't be executed by entity classes or by direct consumers of entity classes. They should only be executed in a few, reserved components such as a *Repository*, a *Unit-of-Work*, or a *DataService*. A code review should look for references to *EntityAspect* and *EntityManager* to make sure those objects are only invoked in approved components.

This is our opinion, an opinion grounded in long experience. You are free to disagree ... and we've just shown you how to proceed contrary to our recommendation. But we really hope you'll heed our advice and implement *EntityFacts* instead.

EntityFacts

Many of the *EntityAspect* members are harmless and useful both to authors of entity classes and to consumers of entity classes. We recommend that you write a helper class that exposes the "helpful" members and hides the "harmful" ones. Let's call that class ***EntityFacts***.

If you wrote your own entity base class (as we suggest), you could add a property that returns *EntityFacts*.

```
[ProvideEntityAspect]
[DataContract(IsReference = true)]
```

```
public abstract class EntityBase {
  /// <summary>Get facts about this entity's current state.</summary>
  [NotMapped]
   [Bindable(false), Editable(false), Display(AutoGenerateField = false)]
  public EntityFacts EntityFacts
      {
        get { return _entityFacts ?? (_entityFacts = new EntityFacts(this)); }
      }
  private EntityFacts _entityFacts;
  // ... more ...
}
```

Which members are "helpful" and which are "harmful". That's really up to you. That's why we don't ship an *EntityFacts* class in DevForce. Here is an example that we've found helpful in our practice.

```
/// <summary>
/// Encapsulate access to facts from the DevForce <see cref="EntityAspect"/>.
/// </summary>
public class EntityFacts : INotifyPropertyChanged
{
  private readonly EntityAspect _entityAspect;
  public EntityFacts(object entity)
   {
     _entityAspect = EntityAspect.Wrap(entity);
     _entityAspect.PropertyChanged +=
       (s, args) => RaiseEntityFactsPropertyChanged(string.Empty);
   }
  public EntityState EntityState
   {
     get { return _entityAspect.EntityState; }
   }
  public bool IsNullEntity
   {
     get { return _entityAspect.IsNullEntity; }
   }
  public bool IsPendingEntity
   {
     get { return _entityAspect.IsPendingEntity; }
   }
  public bool IsNullOrPendingEntity
   {
     get { return _entityAspect.IsNullOrPendingEntity; }
   }
  public bool HasErrors
   {
     get { return _entityAspect.ValidationErrors.HasErrors; }
   }
  public EntityAspect.VerifierErrorsCollection ValidationErrors
   {
     get { return _entityAspect.ValidationErrors; }
   }
  protected internal EntityAspect EntityAspect
   {
     get { return _entityAspect; }
   }
  public void RaisePropertyChanged(string propertyName)
   {
      _entityAspect.ForcePropertyChanged(new PropertyChangedEventArgs(propertyName));
   }
  public event PropertyChangedEventHandler EntityPropertyChanged
   {
     add { _entityAspect.EntityPropertyChanged += value; }
     remove { _entityAspect.EntityPropertyChanged -= value; }
   }
  event PropertyChangedEventHandler INotifyPropertyChanged.PropertyChanged
   {
     add { EntityFactsPropertyChanged += value; }
     remove { EntityFactsPropertyChanged -= value; }
   }
  protected event PropertyChangedEventHandler EntityFactsPropertyChanged;
  protected void RaiseEntityFactsPropertyChanged(string propertyName)
```

```
    {
      if (null == EntityFactsPropertyChanged) return;
      EntityFactsPropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```