Contents

A DevForce entity need not derive from a base class. But you are welcome to create a base class of your own and derive some or all of your entities from that class.

It is not a violation of POCO principles to *have* a base class; it is a violation to *require* a base class.

We think there are good reasons to create your own base class and we'll suggest how you might do that in this topic.

Entity classes generated **Database First** derive from the *IdeaBlade.EntityModel.**Entity*** class ... which you cannot **not** use in your **Code First** models. In future releases, these generated entities may no longer derive from *Entity*.

Benefits of a base entity class

Entities tend to have common capabilities, typically cross-cutting concerns such as your own validation protocol (e.g., your own *Validate* method or properties such as *CanDelete* that indicate allowed and disallowed operations).

The DevForce AOP system injects common infrastructure into your classes without adding a base class. That infrastructure is discovered by UI components at runtime but you can't refer to any of it yourself when writing your own code unless you cast first. That's ugly and error prone.

It is often easiest to locate custom shared features in a base class. DevForce is not allowed to use a base class for this purpose. But you can; it's your model.

An *EntityBase* skeleton

We recommend that you begin developing your model by writing an *EntityBase* class such as this one:

```
/// <summary>
/// My custom base class for all entity classes in my model.
/// It contains whatever I believe is useful across my entity classes.
/// </summary>
/// <remarks>
/// An entity base class is optional. This is MY BASE class.
/// <para>
/// The base class is decorated with the ProvideEntityAspect attribute,
/// so CodeFirst infrastructure is also injected into sub-classes.
/// </para>
/// </remarks>
[ProvideEntityAspect]
[DataContract(IsReference = true)]
public abstract class EntityBase {
  // Your code here
}
```

EntityBase is decorated with the DevForce *ProvideEntityAspect* attribute that identifies this class – and all of its subclasses – as an AOP Entity class which should be rewritten. Applying the attribute here means you won't have to apply it to each of your derived business entity classes.

In fact, you can't apply it to those classes; this attribute can only appear on one class in an inheritance chain.

EntityBase is ***abstract***. You should never create an instance of it. Marking it *abstract* ensures that you won't and also tells Entity Framework Code First that you do not intend to map *EntityBase* to an object in the database.

Always apply the *[DataContract(IsReference = true)]* attribute. It is harmless if all of your derived entity classes consist entirely of serialized public properties. But you'll need it on this base class **if even one** of your derived entity classes requires explicit WCF DataContract configuration.

What to put in *EntityBase*

Add members to this base class that you want all entities to share. Make them *public* or *protected* as appropriate.

Custom validation

One popular addition is a custom *Validate()* method that you can call during save. Each derived class can override it to perform its own validation logic. It might, for example, confirm that the entity can be added, modified, or deleted by the current user.

```
/// <summary>
/// Perform custom validation of this entity and add errors to the <see cref="validationErrors"/>
/// </summary>
public virtual void Validate(VerifierResultCollection validationErrors) { }
```

You might call *Validate* inside your *EntityManager.Saving* handler, once for each entity in the change-set. Each *Validate* adds errors (if any) to the *VerifierResultCollection*. After iterating through all of the entities, if there are errors in the collection, the *Saving* handler cancels the save and presents the errors collection to the UI.

EntityAspect

Some developers add a public *EntityAspect* property to make it easier to access the DevForce features made available through the *EntityAspect* class

```
public EntityAspect EntityAspect { get { return null; } }
```

Weird rule alert! We told you to put the *ProvideEntityAspect* attribute on your base class. If you follow our lead, you must ensure that the *EntityAspect* property **is not *virtual***. On the other hand, if you decide to ignore our recommendation and you omit the *ProvideEntityAspect* attribute on your base class, then the *EntityAspect* property **must** be *virtual*. Don't ask why.

We described how this works elsewhere. However, we don't like this approach because it encourages application developers to work directly with the *EntityAspect.EntityManager* which we think is a bad idea.

We do it frequently in our demos because it is easy. **Demos** do not always follow good practices.

We'd rather restrict *EntityManager* access to a designated component for that purpose ... a *Repository* or *DataService* perhaps. In short, *EntityAspect* is a little too powerful to expose in this way. Remember that the entity class author can always obtain the *EntityAspect* if she absolutely must, in either of these ways:

```
((IEntity) someCategory).EntityAspect.EntityState; // by casting
EntityAspect.Wrap(someCategory).EntityState;       // by wrapping
```

Even this mode of access should be the rare exception.

EntityFacts

There is a middle ground between no access and complete access to *EntityAspect*. You can create a property returning a custom object that exposes "the safe" members of *EntityAspect*. What "safe" means is entirely up to you. We provide one possible answer called *EntityFacts* in the topic on *EntityAspect* in Code First.

An example *EntityBase*

Here is an example of an *EntityBase* class that conforms to our recommendations:

```
/// <summary>
/// My custom base class for all entity classes in my model.
/// It contains whatever I believe is useful across my entity classes.
/// </summary>
[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;
  /// <summary>
  /// Perform custom validation of this entity and add errors to the <see cref="validationErrors"/>
```

```
    /// </summary>
    public virtual void Validate(VerifierResultCollection validationErrors) { }
}
```

You'll find an example *EntityFacts* class implementation in the topic on *EntityAspect* in Code First.

EntityBase in a separate assembly

You can put your *EntityBase* class in an assembly separate from the rest of your model. This is commonly done when you have multiple Code First models but wish to use a common base class with them all.

You should install the *DevForce Code First NuGet package* to the project holding the base class to ensure that the correct assembly references are added and PostSharp MSBuild support is enabled.

However, you don't need DevForce MSBuild support to be enabled. To disable DevForce build support you can do either of two things:

- Remove the DevForce.cf marker file - The DevForce build task will only start if the file is found.
- Edit the project file to remove the import of the *IdeaBlade.DevForce.Common.targets* file which injects the DevForce *EntityModelMetadataDeploy* task into the build pipeline.