Contents

Generated entity classes ultimately inherit from *Entity* but you can inject your own **custom entity base class** at the root of your entity model's inheritance hierarchy in order to provide common state and behaviors across the model. This topic explains how.

In many applications there is logic that every entity should share. It's natural to put this logic in a base class and have all of your entity classes inherit from it.

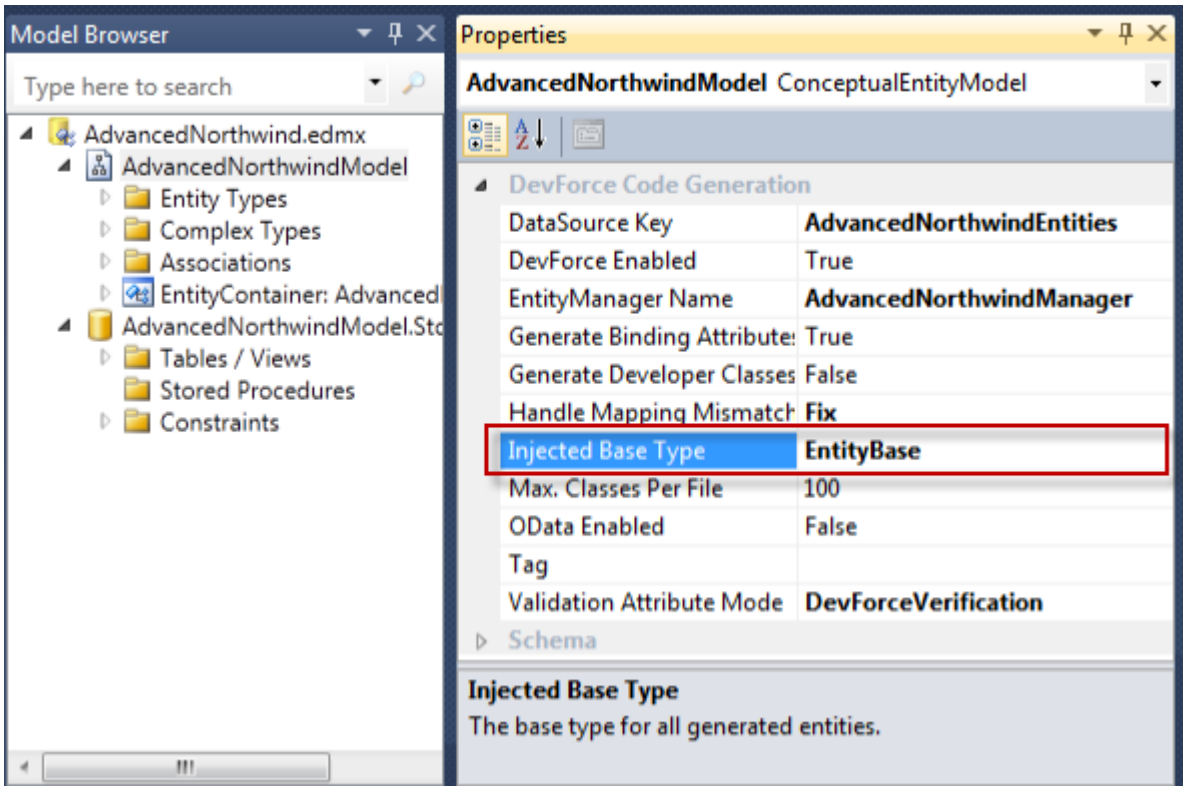
A wrong way

You happen to know that when an entity class inherits from another class, you specify the base class in the EDM Designer *Base Type* property. You consider creating an empty, abstract *EntityBase* type in the conceptual model and setting every entity's *Base Type* property to "EntityBase".

That won't work for several reasons chief among them: Entity Framework insists that **all** entity types be mapped to a store object. There is no store object for *EntityBase* and you can't create one either.

Name it in *Injected Base Type*

The DevForce EDM Designer Extension added many code generation control properties including a model-level *Injected Base Type* property. You set the *Injected Base Type* to "EntityBase" in the EDM Designer Properties Window as shown:



Base type code-generation

The DevForce code generator substitutes *EntityBase* wherever it would have specified the DevForce *Entity* class.

```
public partial class Customer : EntityBase {...}
```

```
Partial Public Class Customer Inherits EntityBase  ...
End Class
```

It also adds to the entity source code file an empty *EntityBase* partial abstract class that inherits from *Entity*:

```
using IbEm   = IdeaBlade.EntityModel;
...
[DataContract(IsReference=true)]
[IbEm.DiscoverableType(IbEm.DiscoverableTypeMode.KnownType)]
public abstract partial class EntityBase : IbEm.Entity {
}
```

```
Imports IbEm   = IdeaBlade.EntityModel
...
<DataContract(IsReference=True)>
<IbEm.DiscoverableType(IbEm.DiscoverableTypeMode.KnownType)>
Partial Public MustInherit Class EntityBase Inherits IbEm.Entity
EndClass
```

Add code to *EntityBase*

Now add the logic that you want all entities to share ... but not to the generated *EntityBase* class! Your changes would be lost the next time you re-generated the entity class file ... which happens often.

Instead, follow the same pattern for extending generated entity classes: add a partial class file called *EntityBase.cs* (or *EntityBase.vb*).

Remember to link to it in your Silverlight model project.

Here's an example with a dummy property:

```
public partial class EntityBase
{
  /// <summary>
  /// <see cref="EntityBase"/> demo property that gets the name of the concrete type.
  /// </summary>
  protected internal string EntityTypeName { get { return This.GetType().Name; } }
}
```

```
Partial Public Class EntityBase
  ''' <summary>
  ''' <see cref="EntityBase"/> demo property that gets the name of the concrete type.
  ''' </summary>
  Protected Friend ReadOnly Property EntityTypeName As String
    Get
      Return Me.GetType().Name
    End Get
  End Property
End Class
```

Base class in a different assembly

You can define the *EntityBase* class in a different assembly if you want to share it with multiple models in multiple projects. You might even use *EntityBase* in different DevForce applications.

DevForce has a naming convention for that purpose. If the name supplied to the *Injected Base Type* EDM Designer extension property contains a period (.), DevForce assumes that the named class already exists elsewhere and uses the name exactly as you specified it.

Suppose you defined *EntityBase* in a class library called *Common*. Presumably its namespace is also called *Common*. **Set the *Injected Base Type*** to "Common.EntityBase". **Add a reference** to the *Common* library to your model project. DevForce will generate entities that inherit from *Common.EntityBase*.

Imitate the generated *EntityBase* if you decide to write your own in a different project. Remember to inherit from *Entity* and include the two class-level attributes shown above.

Base class for specific entity classes

Perhaps you want a few of your entity classes to inherit from a special base class dedicated just to them.

For example, suppose that *Customer* and *Employee* have functionality in common as "actors" in the domain.

These classes have no properties in common and they do not inherit from another entity model class (i.e., they are not involved in any form of Entity Framework inheritance). For reasons known only to you they have some business logic in common. That logic is irrelevant or incorrect for all other model entities so you refuse to put that logic in the model-wide *EntityBase* class.

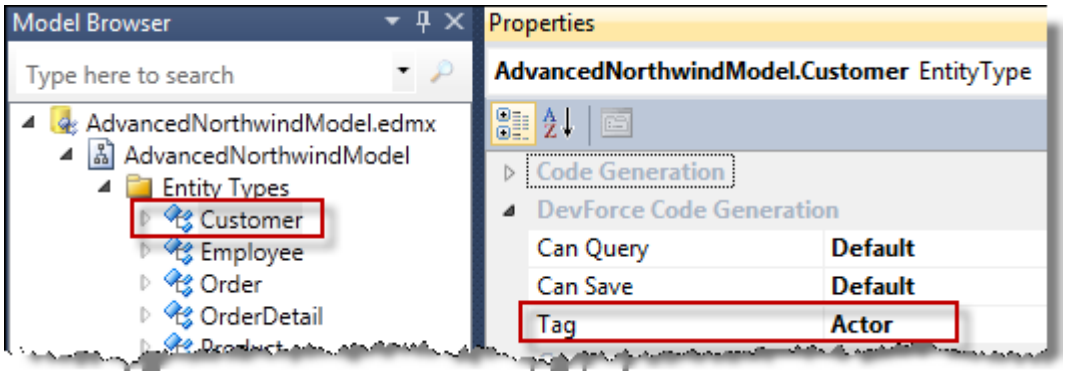
You've already defined an abstract *Actor* class to hold this logic (see below). Now you want to make *Customer* and *Employee* inherit from *Actor*.

You'll have to modify the DevForce code generation template to do it. You can't specify a custom base class for individual entity types in the designer.

You will be tempted by the EDM *Base Type* property. It won't work. That property is constrained to other types in the entity model.

Fortunately, it's not difficult.

- Follow the steps for customizing the DevForce T4 template
- Override the *DomainModelTemplate.GetEntityClassDef* method as shown below
- Set the EDM *Tag* property of both *Customer* and *Employee* to "Actor"; your custom template will know to use the *Actor* class when it sees this *Tag*.



Here is the suggested method override:

```
protected override ClassDef GetEntityClassDef
 (EntityOrComplexTypeWrapper entityOrComplexType) {
 String baseName;
 var entityType = entityOrComplexType as EntityTypeWrapper;
bool isAbstract;
if (entityType != null) {
   baseName = GetMyEntityBaseTypeName(entityType); // <-- substitute
  //baseName = GetEntityBaseTypeName(entityType); // <-- original
  isAbstract = entityType.IsAbstract;
 } else {
   baseName = "IbEm.ComplexObject";
   isAbstract = false;
 }
 var classDef = new ClassDef(FmtName(entityOrComplexType.Name),
   FmtName(baseName),
   entityOrComplexType.Accessibility)
   .SetAbstract(isAbstract)
   .SetPartial(true);
return classDef;
}
protected static string GetMyEntityBaseTypeName(EntityTypeWrapper entityType) {
 if (entityType.Tag.Contains("Actor")) return "Actor";  // ToDo: replace magic string
 return GetEntityBaseTypeName(entityType);
}
```

```
Protected Overrides Function GetEntityClassDef(ByVal entityOrComplexType As _
 EntityOrComplexTypeWrapper) As ClassDef
Dim baseName As String
Dim entityType = TryCast(entityOrComplexType, EntityTypeWrapper)
Dim isAbstract As Boolean
If entityType IsNot Nothing Then
   baseName = GetMyEntityBaseTypeName(entityType) ' <-- substitute
  'baseName = GetEntityBaseTypeName(entityType); // <-- original
  isAbstract = entityType.IsAbstract
```

```
 Else
    baseName = "IbEm.ComplexObject"
    isAbstract = False
 End If
 Dim classDef = New ClassDef(FmtName(entityOrComplexType.Name), FmtName(baseName), _
    entityOrComplexType.Accessibility).SetAbstract(isAbstract).SetPartial(True)
  Return classDef
End Function
Protected Shared Function GetMyEntityBaseTypeName(ByVal _
  entityType As EntityTypeWrapper) As String
 If entityType.Tag.Contains("Actor") Then ' ToDo: replace magic string
   Return "Actor"
 End If
 Return GetEntityBaseTypeName(entityType)
End Function
```

Notice that the *GetMyEntityBaseTypeName* method checks for the word "Actor" in the *Tag* property that DevForce added to the EDM Designer properties. If the *Tag* contains "Actor" (as it will for *Customer*), the method returns the "Actor" base name ... and *Customer* inherits from *Actor*. If it doesn't (as *Order* does not), the method returns the injected base type name - "EntityBase" in this case - and *Order* inherits directly from *EntityBase*.

The *Tag* property is a great way to tell your custom code generation method what to do on a case-by-case basis. It's easy to set from within the EDM Designer.

Important: The *Actor* class must inherit from the DevForce *Entity* class. It probably will inherit from your *EntityBase* which inherits from *Entity*.

```
[DataContract(IsReference = true)]
[IbEm.DiscoverableType(IbEm.DiscoverableTypeMode.KnownType)]
public abstract class Actor : EntityBase { ... }
```

```
<DataContract(IsReference = True)>
<IbEm.DiscoverableType(IbEm.DiscoverableTypeMode.KnownType)>
Public MustInherit Class Actor Inherits EntityBase
End Class
```

Do not insert another entity model class between *Actor* and *Entity*. The Entity Framework demands an unbroken chain of entity inheritance ***within*** the model. *Customer->Actor->EntityBase->Entity* is OK because, once you get to *Actor*, there are no more entity model ancestor classes. *Customer->Actor->**SomeModelEntity**->EntityBase->Entity* is **not OK**.