Contents

You can extend the metadata for your entity with a metadata class (aka, the "buddy" class) to specify precisely the attributes that should be applied to generated entity class properties.

Attributes are an increasingly popular means of adding information and behavior to the members of a class.

Many UI technologies, WPF and Silverlight controls in particular, inspect the data-bound entity properties, looking for UI hint attributes such as *Display*. The vocabulary of UI hint attributes is rich and growing.

Validation engines on both client and server also inspect entities for validation attributes such as *Required*; the engines turn these attributes into rules that they apply when validating entities.

The DevForce code generator adds many such attributes to the generated classes. But it probably won't get all of the ones that you care about and you may want to impose different values for some of the attributes DevForce does supply.

The DevForce EDM Designer Extension gives you some control over the generated attributes but not as much control as you might need. Once the classes have been generated, their attributes are baked in.

There is no direct way to add unanticipated attributes to the generated property. The code generator doesn't know about them. Again, once the classes have been generated, their attributes are baked in.

A property with a problem

For example, here is the *CompanyName* property generated for the *Customer* class.

```
/// <summary>Gets or sets the CompanyName. </summary>
[Bindable(true, BindingDirection.TwoWay)]
[Editable(true)]
[Display(Name="CompanyName", AutoGenerateField=true)]
[IbVal.StringLengthVerifier(MaxValue=40,
                        IsRequired=true,
                        ErrorMessageResourceName="Customer_CompanyName")]
[DataMember]
public string CompanyName {
 get { return PropertyMetadata.CompanyName.GetValue(this); }
 set { PropertyMetadata.CompanyName.SetValue(this, value); }
}
```

```
''' <summary>Gets or sets the CompanyName. </summary>
<Bindable(True, BindingDirection.TwoWay), Editable(True), _
  Display(Name:="CompanyName", AutoGenerateField:=True), _
  IbVal.StringLengthVerifier(MaxValue:=40, IsRequired:=True, _
  ErrorMessageResourceName:="Customer_CompanyName"), DataMember>
Public Property CompanyName() As String
 Get
   Return PropertyMetadata.CompanyName.GetValue(Me)
 End Get
 Set(ByVal value As String)
    PropertyMetadata.CompanyName.SetValue(Me, value)
 End Set
End Property
```

We might prefer a different label than "CompanyName" and want automatic layout controls to position the name near the front or top.

Write a metadata class

Fortunately, you can write an entity metadata class (aka, the "buddy" class) to specify precisely the attributes that should be applied to a designated property. You can add any attribute to a generated property, including custom attributes that you wrote. And you can remove or replace attributes with the help of the metadata class

The DevForce T4 code generator looks for a metadata class associated with a class it is generating. If it sees a *public* field or property name in that buddy class that matches the name of a property it would generate, it applies the attributes on that buddy class property instead of the attributes it would otherwise have generated.

```
[MetadataType(typeof(CustomerMetadata))]
```

```
[DebuggerDisplay("{ToString()}")]
public partial class Customer
{
  //...
  public static class CustomerMetadata
   {
      [Display(Name = "ID", AutoGenerateField = true, Order = 0)]
      [Editable(false)]
      public static Guid CustomerID;
      [Display(Name = "Name", AutoGenerateField = true, Order = 1)]
      [DoNothingCustom]
      public static string CompanyName;
      // Country is nullable in database but we want to require it
      [RequiredValueVerifier]
      public static string Country;

      [Bindable(true, BindingDirection.OneWay)]
      [Editable(false)]
      [Display(Name = "CustomerID_OLD", AutoGenerateField = false)]
      public static Guid CustomerID_OLD;
   }
}
```

```
<MetadataType(GetType(CustomerMetadata)), _
  DebuggerDisplay("{ToString()}")>
Partial Public Class Customer
 '...
 Public NotInheritable Class CustomerMetadata
  <Display(Name := "ID", AutoGenerateField := True, Order := 0), _
           Editable(False)>
  Public Shared CustomerID As Guid
  <Display(Name := "Name", AutoGenerateField := True, Order := 1), _
           DoNothingCustom>
  Public Shared CompanyName As String
 ' Country is nullable in database but we want to require it
  <RequiredValueVerifier>
 Public Shared Country As String
  <Bindable(True, BindingDirection.OneWay), Editable(False), _
           Display(Name := "CustomerID_OLD", AutoGenerateField := False)>
  Public Shared CustomerID_OLD As Guid
 End Class
End Class
```

Some observations:

- The ***MetadataType* attribute is essential**. It tells DevForce that a buddy class exists and where to find it.
- The metadata class is an inner static class in this example. It could stand on its own, perhaps in a separate code file.
- The metadata class is ***public*** as are its properties and fields.
- The *Editable* attributes warns the UI not to facilitate changing the key (*CustomerID*)
- The *Display* attribute specifies the desired label and relative position of the ID field if the UI will be laying out the view automatically.
- *CompanyName* gets a label change and its own custom attribute, *DoNothingCustom*. It may mean nothing to .NET controls but it might mean something to your code.
- The *Country* property is nullable in the database; the *RequiredValueVerifier* ensures that it must have a value anyway if the *Customer* is to be saved.
- The attributes on *CustomerID_OLD* instruct the UI to hide that property and forbid change if possible.

Re-running the code generator with this metadata class in place causes DevForce to produce different code for these properties than it would have otherwise. Here's the *CompanyName* property for example:

```
/// <summary>Gets or sets the CompanyName. </summary>
[Display(Name = "Name", AutoGenerateField = true, Order = 1)]
[SimpleNorthwindModel.DoNothingCustom]
[Bindable(true, BindingDirection.TwoWay)]
[Editable(true)]
[IbVal.StringLengthVerifier(MaxValue=40,
                    IsRequired=true,
                    ErrorMessageResourceName="Customer_CompanyName")]
[DataMember]
public string CompanyName {
 get { return PropertyMetadata.CompanyName.GetValue(this); }
```

```
 set { PropertyMetadata.CompanyName.SetValue(this, value); }
}
```

```
''' <summary>Gets or sets the CompanyName. </summary>
<Display(Name := "Name", AutoGenerateField := True, _
  Order := 1), SimpleNorthwindModel.DoNothingCustom, Bindable(True, _
  BindingDirection.TwoWay), Editable(True), IbVal.StringLengthVerifier(MaxValue:=40, _
  IsRequired:=True, ErrorMessageResourceName:="Customer_CompanyName"), DataMember>
Public Property CompanyName() As String
Get
Return PropertyMetadata.CompanyName.GetValue(Me)
End Get
Set(ByVal value As String)
PropertyMetadata.CompanyName.SetValue(Me, value)
End Set
End Property
```

See the revised *Display* and new *DoNothingCustom* attributes. We didn't touch the validation attributes so the templated validation remains in place.

DevForce does **not** re-generate the model automatically after changes to the metadata class. **REMEMBER** to re-run the **code generation tool** after changing the metadata class so that its attributes are merged into the generated class code.

Link to Silverlight and Windows Store projects?

In this example, the metadata class is an inner class of the custom *Customer* class located in a partial class file.

The client model project is presumably already linked to this partial class as we discussed in the partial class topic so there's no additional linking required.

What if we had put the metadata class in a separate class file? Should the client project link to it then? It depends.

There's no need if the metadata class file contains nothing but attributes adorning static fields or properties; DevForce code-generation will have baked their effects into the generated classes already; the metadata class has served its purpose.

On the other hand, if the metadata class defines other **business logic**, such as the validation rules in the *Customer* class example, then you must link to this metadata class manually in order to make that logic available on the Silverlight client.