Contents

You often delegate a portion of the entity business logic to a **helper class** defined outside the entity. The architectural forces driving this choice are too numerous to list. You will know when externalizing the logic makes sense.

It is "obvious" how to add *new* entity class members that delegate work to helper classes. It is less obvious how you delegate work from *within a generated property*. Once the class is generated, you can't change its implementation.

Fortunately, the generated properties are implemented with *PropertyMetadata* objects as seen in this *Customer.CompanyName* property

```
public string CompanyName {
 get { return PropertyMetadata.CompanyName.GetValue(this); }
 set { PropertyMetadata.CompanyName.SetValue(this, value); }
}
```

```
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
```

The *PropertyMetadata.CompanyName* objects contain "get" and "set pipelines that are responsible for numerous behaviors including validating input, executing custom actions through property interceptors, and raising the *PropertyChanged* event. In DevForce you can inject your own business logic into these pipelines and do so from outside the entity class via helper classes. Your options include:

1. Add more validation rules via a *VerifierProvider*
2. Add and remove the validation rules of a *VerifierEngine*.
3. Add and remove the *PropertyInterceptorAction*s of a *PropertyInterceptorManager*.

Add validation rules to a *VerifierProvider*

You verify the integrity of property input values with validation rules. The *PropertyMetadata* object within each generated property triggers the validation process that involves the application of the rules.

The DevForce code generator produces some rules based on information gleaned from the database schema (non-null, string length). You can add more in a companion metadata "buddy" class.

You can also add rules anywhere in your application. One approach is to define an implementation of *IVerifierProvider* that adds new rules for an entity type. DevForce uses MEF to discover these *VerifierProvider* classes and DevForce executes the automatically when it is ready to build up the rule-set for the entity type. Here's an example:

```
public class NorthwindModelVerifierProvider : IVerifierProvider
{
  public IEnumerable<Verifier> GetVerifiers(object verifierProviderContext)
  {
     var verifiers = new List<Verifier>
        {
         MakeNotEmptyGuidVerifier<Customer>(
            Customer.PropertyMetadata.CustomerID.Name),
         MakeNotEmptyGuidVerifier<Order>(
            Order.PropertyMetadata.CustomerID.Name),
        };
    return verifiers;
  }
}
```

```
Public Class NorthwindModelVerifierProvider
 Implements IVerifierProvider
 Public Function GetVerifiers(ByVal verifierProviderContext _
  As Object) As IEnumerable(Of Verifier)
  Dim verifiers = New List(Of Verifier) _
    From {MakeNotEmptyGuidVerifier(Of Customer) _
    (Customer.PropertyMetadata.CustomerID.Name), _
```

```
    MakeNotEmptyGuidVerifier(Of Order) _
    (Order.PropertyMetadata.CustomerID.Name)}
  Return verifiers
 End Function
End Class
```

The details are covered in the "Validate" topic. Here it is enough to see that, inside the *GetVerifiers* method, we're adding two rules, one each for *Customer* and *Order*, both ensuring that the *CustomerID* they both care about is *not* the empty *Guid* (the one with all zeros).

The words "verify" and "validate" mean the same thing in this discussion. DevForce uses the term *verifier* for historical reasons that are irrelevant to us.

Add and remove rules of a *VerifierEngine*

Every *EntityManager* is associated with a *VerifierEngine*. The generated properties of an entity attached to an *EntityManager* have access to the manager's *VerifierEngine* and use that engine to validate input values. You can change property validation rules by grabbing that *VerifierEngine* and adding or removing property validation rules. You would likely do that in a helper class conveniently located in the place where you configure the application's entity model.

A static *VerifierEngineCreated* event affords the opportunity to dynamically configure every *VerifierEngine* the application creates.

Add and remove property interceptors

Validations pass judgment on the validity of input. They may report problems but they don't change input values. Property interceptors give you complete control over the behavior of any generated property.

You use a property interceptor if you want to change an input value or alter the value returned by a property getter. You can also use a property interceptor to do something other than manipulate a property's data values. Property level authorization is a common use for a property interceptor. If an unauthorized user attempts to set the property, the interceptor could cancel the set or throw an exception.

You can add property interceptors to the custom partial class, a technique that locates the business logic inside the class.

You can also add property interceptors dynamically outside the entity class in a place that is convenient to your purpose. You might add interceptors dynamically if you had a data-driven authentication scheme, perhaps one that relied on a combination of the user's rights and authorization metadata imported from a database. That is the kind of thing you should create and operate outside the entity class.

Here is a much simpler example, an interceptor that removes whitespace from a *Customer.ContactName* input value.

```
private static void AddContactNameInterceptor()
{
   var act = new PropertyInterceptorAction<
             DataEntityPropertySetInterceptorArgs<Customer, string>>(
         typeof (Customer),
          Customer.EntityPropertyNames.ContactName,
          PropertyInterceptorMode.BeforeSet,
          args =>
          {
            if (null == args.Value) return;
             args.Value = Regex.Replace(args.Value, @"\s", "");
          }
       );
   PropertyInterceptorManager.CurrentInstance.AddAction(act);
}
```

```
Private Shared Sub AddContactNameInterceptor()
 Dim act = New PropertyInterceptorAction(Of DataEntityPropertySetInterceptorArgs _
   (Of Customer, String))(GetType(Customer), Customer.EntityPropertyNames.ContactName, _
   PropertyInterceptorMode.BeforeSet, Function(args)
If Nothing Is args.Value Then
 Return
End If
args.Value = Regex.Replace(args.Value, "\s", "")
   End Function)
  PropertyInterceptorManager.CurrentInstance.AddAction(act)
End Sub
```

The meat of the interceptor is in the lambda *Action* delegate.

```
...
 args =>
 {
   if (null == args.Value) return;
    args.Value = Regex.Replace(args.Value, @"\s", "");
 }
...
```

```
...
Sub(args)
 If Nothing Is args.Value Then
   Return
 End If
  args.Value = Regex.Replace(args.Value, "\s", "")
End Sub
...
```

In practice you would define this as an attributed property interceptor inside the *Customer* partial class; that's easier and there is no obvious reason to define this domain behavior outside the entity class. The contrived example does illustrate the mechanics of adding a dynamic property interceptor outside the entity class.

If you had a general rule about removing whitespace that applied to many properties of many entity types, then it makes sense to add a general purpose property interceptor to the *PropertyInterceptorManager* in a helper class near the start of the application.