Contents

DevForce ships with a number of pre-defined **verifiers that can be customized in a variety of ways**: examples include the *DateTimeRangeVerifier*, the *RegexVerifier*, the *StringLengthVerifier*, the *RequiredValueVerifier*, and a number of others. In addition, it defines a generic *DelegateVerifier<T>* that provides unlimited verification capabilities: you can use it to define any sort of rule you need or can imagine.

There are several ways to create a verifier, but in general the easiest ways involve starting with an existing one and modifying it to suit.

Creating a customized version of a standard (provided) verifier

The method *GetHireDateRangeVerifier* below illustrates the creation of a 'customized' verifier by simply creating one of the standard DevForce verifiers with specific parameters, in this case, the *DateTimeRangeVerifier*. Other provided verifiers include many others that are specific to .NET simple types, including *DecimalRangeVerifier*, *DoubleRangeVerifier*, *Int32RangeVerifier*, and so forth. There is a *ListVerifier*, a *Regex* Verifier, a *NamedRegexPattern* verifier, and a number of others. All are defined in the *IdeaBlade.Validation* namespace.

The constructor for the *DateTimeRangeVerifier* provides for all of the configuration we need. It permits us to specify the target business object type for the rule, the property on that type whose setting will trigger the running of the rule, whether a non-null value is required when that property is set, and the dates that define the valid range:

```
private static Verifier GetHireDateRangeVerifier(DateTime minHireDate, DateTime maxHireDate) {
  Verifier v = new DateTimeRangeVerifier(
   typeof(Employee), // Type of the object being verified
   Employee.PropertyMetadata.HireDate.Name, // Property trigger
   false, // Non-null value is not required
   minHireDate, true, // starting min date (inclusive)
   maxHireDate, false); // ending max date (exclusive)
 return v;
}
```

```
Private Shared Function GetHireDateRangeVerifier( _
  minHireDate as DateTime, maxHireDate as DateTime) As Verifier
 Dim v As Verifier = New DateTimeRangeVerifier( _
   GetType(Employee), _
    Employee.PropertyMetadata.HireDate.Name, _
   False, _
    minHireDate, True, _
    maxHireDate, False)
 Return v
End Function
```

If you need to improve upon the default message produced by the above verifier, you can take control of the messaging by setting the verifier's *VerifierArgs.ErrorMessageInfo.Error*. For example, if you insert the following statement right before the return statement in the above method, your message will be the one users see:

```
v.VerifierArgs.ErrorMessageInfo.ErrorMessage = String.Format(
"The HireDate must fall on or after the Company's launch date of {0} " +
"and before one month from today ({1}).",
  minHireDate.ToShortDateString(), maxHireDate.ToShortDateString());
```

```
v.VerifierArgs.ErrorMessageInfo.ErrorMessage = String.Format( _
 "The HireDate must fall on or after the Company's launch date of {0} " _
  & "and before one month from today ({1}).", _
  minHireDate.ToShortDateString(), maxHireDate.ToShortDateString())
```

Subclassing a pre-defined verifier

Another way to create a custom verifier is to subclass one of the pre-defined verifiers that are in the *IdeaBlade.Validation* namespace. In the following code samples below, we will create a custom verifier that checks that a given GUID property

does NOT equal *Guid.Empty*. We will do this by subclassing a *PropertyValueVerifier* to define a custom class called NotEmptyGuidVerifier.

Step 1: Create the custom *NotEmptyGuidVerifier* class by inheriting it from the *PropertyValueVerifier* class. The constructor takes 2 parameters, an *entityType* and a *propertyName*. This allows us to pass any entity and any Guid property.

```
public class NotEmptyGuidVerifier : PropertyValueVerifier {
 public NotEmptyGuidVerifier(Type entityType, String propertyName,
   String displayName = null, bool? shouldTreatEmptyStringAsNull = null)
   : base(new PropertyValueVerifierArgs(entityType, propertyName, false,
     displayName, shouldTreatEmptyStringAsNull)) { }
 protected override VerifierResult VerifyValue(object itemToVerify, object valueToVerify,
   TriggerContext triggerContext, VerifierContext verifierContext) {
     var aGuid = (Guid)valueToVerify;
     var result = (aGuid != Guid.Empty);
    return new VerifierResult(result);
   }
  }
}
```

```
Public Class NotEmptyGuidVerifier
 Inherits PropertyValueVerifier
 Public Sub New(ByVal entityType As Type, ByVal propertyName As String, _
  Optional ByVal displayName As String = Nothing, Optional ByVal _
   shouldTreatEmptyStringAsNull? As Boolean = Nothing)
  MyBase.New(New PropertyValueVerifierArgs(entityType, propertyName, _
    False, displayName, shouldTreatEmptyStringAsNull))
 End Sub
 Protected Overrides Function VerifyValue(ByVal itemToVerify As Object, _
  ByVal valueToVerify As Object, ByVal triggerContext_Renamed As  _
   TriggerContext, ByVal verifierContext_Renamed As VerifierContext) _
    As VerifierResult
  Dim aGuid = CType(valueToVerify, Guid)
  Dim result = (aGuid IsNot Guid.Empty)
  Return New VerifierResult(result)
 End Function
End Class
```

Step 2: Wrap the *NotEmptyGuidVerifier* class in a method that returns a Verifier. We will call this method "*CheckForEmptyGuidVerifier*".

```
private static Verifier CheckForEmptyGuidVerifier(Type entityType, string propertyName) {
  Verifier v = new EmptyGuidVerifier(entityType, propertyName);
  v.VerifierArgs.ErrorMessageInfo.ErrorMessage =
    String.Format("{0} cannot be empty", propertyName);
 return v;
}
```

```
Private Shared Function CheckForEmptyGuidVerifier(ByVal entityType _
 As Type, ByVal propertyName As String) As Verifier
 Dim v As Verifier = New EmptyGuidVerifier(entityType, propertyName)
  v.VerifierArgs.ErrorMessageInfo.ErrorMessage = String.Format( _
   "{0} cannot be empty", propertyName)
 Return v
End Function
```

Step 3: Add the method to the list of Verifiers that is defined inside the *VerifierProvider"* class that implements the *IVerifierProvider* interface. Note that in this example, we are using the verifier to validate an *Order.CustomerID* property (presumably of type GUID).

```
public class VerifierProvider : IVerifierProvider {
 public IEnumerable<Verifier> GetVerifiers(object verifierProviderContext) {
   List<Verifier> verifiers = new List<Verifier>();

   //In this example, we are passing a type Order and validating the Order.CustomerID property
   verifiers.Add(CheckForEmptyGuidVerifier(typeof(Order),
     Order.PropertyMetadata.CustomerID.Name));
   return verifiers;
  }
}
```

```
Public Class VerifierProvider
 Implements IVerifierProvider
```

```
 Public Function GetVerifiers(ByVal verifierProviderContext As _
   Object) As IEnumerable(Of Verifier)
   Dim verifiers As New List(Of Verifier)()
   'In this example, we are passing a type Order and validating
   ' the Order.CustomerID property
   verifiers.Add(CheckForEmptyGuidVerifier(GetType(Order), _
     Order.PropertyMetadata.CustomerID.Name))
   Return verifiers
 End Function
End Class
```

Step 4: And finally, let's test our newly created verifier.

```
public void GenericVerifierForGUIDEmpty() {
  var mgr = new NorthwindIBEntityManager();
  var anOrder = mgr.Orders.FirstOrNullEntity();
  anOrder.CustomerID = Guid.Empty;
  Assert.IsTrue(anOrder.EntityAspect.ValidationErrors.Count > 0,
   "CustomerID is assigned to GUID.Empty");
```

```
Public Sub GenericVerifierForGUIDEmpty()
 Dim mgr = New NorthwindIBEntityManager()
 Dim anOrder = mgr.Orders.FirstOrNullEntity()
  anOrder.CustomerID = Guid.Empty
  Assert.IsTrue(anOrder.EntityAspect.ValidationErrors.Count > 0, _
   "CustomerID is assigned to GUID.Empty")
End Sub
```

Now let's recap what we just did.

1. We created our generic verifier by subclassing a *PropertyValueVerifier* class that is pre-defined in the *IdeaBlade.Validation* namespace.
2. We wrap up the class inside a method that returns a Verifier. This is a required signature to add the verifier to the *VerifierEngine*.
3. We add the method to the list of Verifiers that is defined inside a class that implements an *IVerifierProvider* interface. In our example, this class is called *VerifierProvider*.
4. Using a *TestMethod*, we test the Verifier by retrieving *anOrder* and assigning its *CustomerID* to *Guid.Empty*. The *Assert.IsTrue* method should succeed as *anOrder.EntityAspect.ValidationErrors.Count* should be more than 0.

Creating a completely custom verifier using the DelegateVerifier

Despite the imposing name, the *DelegateVerifier* is really just a completely custom verifier. With this verifier you define the type of the object to verify, when and how to apply the verifier, and the validation to be performed. The *DelegateVerifier* is particularly useful for verifying multiple properties on a type, and for cross-type validation.

When you create the custom verifier you provide a *VerifierCondition*, which is really just a function which performs the actual validation wanted. The function returns a *VerifierResult*, indicating whether the validation succeeded or not.

You can also provide the constructor with the an *ApplicabilityConstraint*. Another imposing term, but it's just a function which given an object and some "context", determines whether the verifier should be run, in other words, whether it's applicable.

As with other verifiers, you can also set the VerifierOptions. These options control when the verifier will be executed, and what to do when a validations fails.

The verifier also needs one or more "triggers", one or more members, either on the type to be verified or another type, which "trigger" validation. Usually a trigger is the setter for a property, but it can also be a method call. If the custom validation isn't actually triggered by a property set but instead only on the object as a whole, the triggers can indicate the properties of interest.

Triggers

Consider a constraint that specifies that the HireDate for an Employee must be later than the Employee's BirthDate. That seems a pretty reasonable requirement, but suppose when the user enters a HireDate of today, this rule is found to have been violated. But the Employee really *was* hired today, and the problem is that the BirthDate previously entered for the Employee was in error, specifying the year 2015 when it should have said 1985. If we prevent the new HireDate value from being entered into that property, we'll subject the user to a lot of unnecessary work. She'll have to clear the new HireDate, even though it is entirely correct, and then go fix the BirthDate value, and then come back and re-enter the new HireDate value, even though she got it right the first time. Users don't have a lot of tolerance for this sort of thing! It's confusing, irritating, or often both.

A verifier that applies to BirthDate doesn't have to be triggered by a change to that property. You can, instead – or additionally – specify that you want it called when the value of HireDate is changed. You can even set it to be triggered by a change to a property of some other type; and not just any instance of that type, but one that happens to be specifically related. For example, suppose you have a rule that says that the OrderDate on an Order taken by Employee X must be greater than or equal to that Employee's HireDate. You could define this rule (as a verifier) on the Employee type, but specify that it should be triggered not only by a change to Employee.HireDate, but equally by a change to the OrderDate property of any Order written by that Employee!

Example

With some terminology out of the way, let's look at a simple sample. Here we'll define a custom verifier for the Employee class to require that the employee's birth date precedes her hire date. The *VerifierCondition* is simple: the BirthDate must fall before the HireDate. The triggers are the "BirthDate" and "HireDate" properties on the Employee, and options here tell the engine that verification should be performed after either of these properties is set, and for instance-level validation.

```
private static Verifier GetBornBeforeHiredVerifier() {
string description = "Must be born before hired.";
  DelegateVerifier<Employee> v =
  new DelegateVerifier<Employee>(description,
      BornBeforeHiredCondition);
  v.AddTriggers(Employee.PropertyMetadata.BirthDate.Name,
      Employee.PropertyMetadata.HireDate.Name);
  v.VerifierOptions.ExecutionModes =
    VerifierExecutionModes.InstanceAndOnAfterSetTriggers;
return v;
}
private static VerifierResult BornBeforeHiredCondition(
  Employee pEmp, TriggerContext pTriggerContext,
    VerifierContext pVerifierContext) {
if (pTriggerContext != null &&
  // We are not checking the proposed value
  // because don't expect to call it preset
  pTriggerContext.Timing == TriggerTiming.BeforeSet) {
  throw new VerifierException(
    "BornBeforeHired verifier not implemented for Preset");
  }
return new VerifierResult(pEmp.BirthDate < pEmp.HireDate);
}
```

```
Private Shared Function GetBornBeforeHiredVerifier() As Verifier
 Dim description As String = "Must be born before hired."
 Dim v As New DelegateVerifier(Of Employee)(description, _
   AddressOf BornBeforeHiredCondition)
  v.AddTriggers(Employee.PropertyMetadata.BirthDate.Name, _
    Employee.PropertyMetadata.HireDate.Name)
  v.VerifierOptions.ExecutionModes = _
    VerifierExecutionModes.InstanceAndOnAfterSetTriggers
 Return v
End Function
Private Shared Function BornBeforeHiredCondition(ByVal pEmp _
 As Employee, ByVal pTriggerContext As TriggerContext, _
 ByVal pVerifierContext As VerifierContext) As VerifierResult
 ' We are not checking the proposed value
 ' because don't expect to call it preset
 If pTriggerContext IsNot Nothing AndAlso pTriggerContext.Timing = _
    TriggerTiming.BeforeSet Then
   Throw New VerifierException( _
     "BornBeforeHired verifier not implemented for Preset")
 End If
 Return New VerifierResult(pEmp.BirthDate < pEmp.HireDate)
End Function
```

In the above code we define a DelegateVerifier that returns, as always a *VerifierResult*. We add two triggers to our verifier, since it tests the relative value of two different properties, and we want a change in either to result in re-application of our validity test.

BeforeSet Versus *AfterSet* Execution and Its Interaction With *ErrorNotificationMode*

This section requires an understanding of the use of VerifierOptions to configure validation.

Note that we have also the *VerifierExecutionModes* for this verifier set to *InstanceAndOnAfterSetTriggers* . This doesn't matter so much if the *VerifierErrorNotificationMode* is set at the default of *Notify* . Under that setting a verification failure will not result in an exception that blocks the commitment of the proposed value into the business object. But if, on the other hand, the *VerifierErrorNotificationMode* were set to *ThrowException* or *NotifyAndThrowException* then running the verifier in *AfterSet* rather than *BeforeSet* mode would be essential.

To understand why, consider what would happen if a user discovered that both the *HireDate* and *BirthDate* values currently set for a given Employee were incorrect, even though valid according to the "born before hired" rule. To correct them, she needs to change both values; but, upon changing the BirthDate value, the verifier was executed and discovered that the new BirthDate value caused the *HireDate* and *BirthDate* values to be invalid relative to one another. The user planned to correct this problem by entering a new HireDate value which would bring the two values back into relative validity. But if the verifier triggered by the BirthDate change ran BeforeSet and threw an exception, the proposed new value would be prevented by the exception from entering the business object. The user might then try entering the new HireDate value first; but this might also cause a BeforeSet exception. So she would, in fact, be unable to correct the property values for the invalid Employee.

It is therefore a good idea to set the *VerifierOptions* for such a cross-property verifier so that either
1) The verifier runs *AfterSet*.
or
2) The *VerifierErrorNotificationMode* is set to *Notify*.

Thus any validation failure will not throw an exception during and before the 'set' of a temporarily invalid but necessary intermediate value.

Cross-Type Verifiers

A cross-type verifier defines a rule whose application requires the inspection of at least one business object of a type different than the business object being validated. For example, suppose you have a rule that says the OrderDate on any Order may not be earlier than the HireDate of the Employee writing the Order. Clearly the application of this rule requires the inspection of at least one Order and one Employee. If the change that triggers the verifier is to the Employee.HireDate rather than the OrderDate of an Order, then *all* Orders associated with the changed Employee must be checked.

Note that such a verifier needs to have multiple triggers associated with it. In the case of our example, it should be executed whenever either the HireDate of an Employee, or the OrderDate of an Order, is changed.

There is another issue with such a verifier: where should it be defined? DevForce initiates its discovery process for the verifiers associated with a particular type when the first instance of that type is retrieved into the cache. So if we define our Employee.HireDate / Order.OrderDate verifier in the Employee class, it will be discovered automatically when the first Employee is retrieved during a given application session. But if an Order is retrieved and changed before any Employee is downloaded, the verifier will not be discovered and will not be run. The same issue exists, in reverse, if we choose to define the verifier in the Order class. Now if we retrieve and change an Employee before retrieving any Orders, again the verifier won't be discovered, since it is not defined on the Order type.

The solution to this conundrum is to order DevForce explicitly to discover the verifiers for any type that may have verifiers involving a type other than itself. For example, to get EntityManager *_em1* to discover the verifiers for the Order type, you can use the following statement:

```
myEntityManager.VerifierEngine.DiscoverVerifiers(typeof(Order));
```

```
myEntityManager.VerifierEngine.DiscoverVerifiers(GetType(Order))
```

This should be done before there is any chance that one of the verifiers defined in the Order class might be needed for some other type.

We'll define a DelegateVerifier for the cross-type validation to be performed in our sample. We're defining the verifier for the Order class, and then defining triggers for both the OrderDate property of Order and the HireDate property of Employee. We also define the "path" from Employee to Orders. We could have inverted this setup and defined the verifier for the Employee type; it doesn't really matter to DevForce.

```
/// <summary>
/// Implement IVerifierProvider interface to give
/// DevForce the verifiers in use at run time.
/// </summary>
public class VerifierProvider : IVerifierProvider {
public IEnumerable<Verifier> GetVerifiers(
  object verifierProviderContext) {
    List<Verifier> verifiers = new List<Verifier>();
    verifiers.Add(GetOrderDateAfterHiredVerifier());
  return verifiers;
  }
```

```
}
///<summary>Get the OrderDateAfterHired Verifier.</summary>
private static Verifier GetOrderDateAfterHiredVerifier() {
string description =
  "OrderDate must be after the sales rep's HireDate.";
  DelegateVerifier<Order> v = new DelegateVerifier<Order>(
      description, OrderDateAfterHiredCondition);
  v.VerifierOptions.ExecutionModes =
    VerifierExecutionModes.InstanceAndOnAfterSetTriggers;
  v.AddTrigger(Order.PathFor(o => o.OrderDate));
  v.AddTrigger(new TriggerLink(new TriggerItem
    (typeof(Employee), Employee.PathFor(e => e.HireDate)),
    e => ((Employee)e).Orders,
  // Path from trigger (Employee) to Order
  true));   // True = that path returns multiple orders
return v;
}
///<summary>
///Implementor of the OrderDateAfterHired Verifier condition.
///</summary>
private static VerifierResult OrderDateAfterHiredCondition(
  Order target, TriggerContext triggerContext,
  VerifierContext pVerifierContext) {
if (triggerContext != null &&
    triggerContext.Timing == TriggerTiming.BeforeSet) {
  throw new VerifierException(
    "OrderDateAfterHired verifier not implemented for Preset");
  }
bool isOk =
    target.OrderDate.HasValue &&
    target.SalesRep.HireDate.HasValue &&
    target.OrderDate.Value >= pTarget.SalesRep.HireDate.Value;
return new VerifierResult(isOk);
}
```

```
'''<summary>Get the OrderDateAfterHired Verifier.</summary>
Private Shared Function GetOrderDateAfterHiredVerifier() As Verifier
 Dim description As String = _
   "OrderDate must be after the sales rep's HireDate."
 Dim v As New DelegateVerifier(Of Order)(description, _
   AddressOf OrderDateAfterHiredCondition)
  v.VerifierOptions.ExecutionModes = _
    VerifierExecutionModes.InstanceAndOnAfterSetTriggers
  v.AddTrigger(Order.PathFor(Function(o) o.OrderDate))
 ' Path from trigger (Employee) to Order
 v.AddTrigger(New TriggerLink(New TriggerItem( _
   GetType(Employee), Employee.PathFor(Function(e) e.HireDate)), _
  Function(e) (CType(e, Employee)).Orders, True)) _
   ' True = that path returns multiple orders
 Return v
End Function
'''<summary>
'''Implementor of the OrderDateAfterHired Verifier condition.
'''</summary>
Private Shared Function OrderDateAfterHiredCondition(ByVal target _
 As Order, ByVal triggerContext As TriggerContext, _
 ByVal verifierContext As VerifierContext) As VerifierResult
 If triggerContext IsNot Nothing AndAlso triggerContext.Timing = _
    TriggerTiming.BeforeSet Then
   Throw New VerifierException( _
     "OrderDateAfterHired verifier not implemented for Preset")
 End If
 Dim isOk As Boolean = target.OrderDate.HasValue AndAlso _
    pTarget.SalesRep.HireDate.HasValue AndAlso _
    pTarget.OrderDate.Value >= target.SalesRep.HireDate.Value
 Return New VerifierResult(isOk)
End Function
```

Creating a custom verifier attribute for your custom verifier

The easiest way to create your own custom attribute is to inherit from one of our pre-defined verifier attribute classes. Below is an example of inheriting from a *PropertyValueVerifierAttribute* class by extending the NotEmptyGuidVerifier above.

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field, Inherited = false)]
public class NotEmptyGuidVerifierAttribute : PropertyValueVerifierAttribute {
 protected override Verifier BuildVerifierCore(Type pType, String propertyName) {
   return new NotEmptyGuidVerifier(pType, propertyName);
  }
}
```

```
<AttributeUsage(AttributeTargets.[Property] Or AttributeTargets.Field, Inherited := False)> _
Public Class NotEmptyGuidVerifierAttribute Inherits PropertyValueVerifierAttribute
 Protected Overrides Function BuildVerifierCore(pType As Type, propertyName As String) As Verifier
   Return New NotEmptyGuidVerifier(pType, propertyName)
 End Function
End Class
```

You can then apply the attribute to the fields in your buddy class.

```
public class CustomerMetadata {
  [NotEmptyGuidVerifier]
 public static Guid CustomerID;
}
```

```
Public Class CustomerMetadata
  <NotEmptyGuidVerifier> _
 Public Shared CustomerID As Guid
End Class
```