Contents

Part 3: Extend the entity, validate, and show when DevForce is busy - In Part 3 of this series we'll refine the work we did in Part 2: we'll extend the Employee class with custom business logic, add some validation, and add a busy indicator to show when the application is busy.

- **Platform:** WPF
- **Language:** C#, VB
- **Download:** Tour of DevForce WPF

Enhanced logging

Perhaps when we are fetching orders, we'd also like to know the id of the associated employee.

First, we'll create an overload of our WriteToLog method that specifically handles queries.

```
private void WriteToLog(IEntityQuery query) {
  var returnType = query.ElementType;
  var message = (returnType == typeof(Order))
     ? "Fetching Orders from Employee " + CurrentEmployee.EmployeeID
     : "Fetching " + returnType;
  WriteToLog(message);
}
```

```
Private Sub WriteToLog(ByVal query As IEntityQuery)
 Dim returnType = query.ElementType
 Dim message = If(returnType Is GetType(Order), "Fetching Orders from Employee " _
    & CurrentEmployee.EmployeeID, "Fetching " & returnType)
  WriteToLog(message)
End Sub
```

Recall that the first overload of WriteToLog() that we wrote, shown below, simply takes a string parameter.

```
private void WriteToLog(string message) {
  Log.Insert(0, message);
}
```

```
Private Sub WriteToLog(ByVal message As String)
  Log.Insert(0, message)
End Sub
```

We'll also modify the handler for the Fetching event to use the new overload:

```
_mgr.Fetching += (s,e)=> WriteToLog(e.Query);
```

```
AddHandler _mgr.Fetching, Sub(s,e) WriteToLog(e.Query)
```

We can run the application now and see the results of our improved logging.

Extend entities

It would be a nice improvement to show the full name of the employee in the log and not just the id. Currently, we have a FirstName and a LastName, but no FullName property. We could simply concatenate the names when we create the log message, but this is an excellent excuse to show how easily we can extend our entities with custom properties and logic.

Right-click on the WpfSimpleSteps project, and select Add | Class and name it Employee. Modify the new class so that it matches the following:

```
public partial class Employee {
 public string FullName {
   get { return FirstName + " " + LastName; }
  }
}
```

```
Partial Public Class Employee
 Public ReadOnly Property FullName() As String
   Get
     Return FirstName & " " & LastName
   End Get
 End Property
End Class
```

Notice that we are using the partial keyword. This is because DevForce has already created an Employee class for us and we want to extend that existing class.

Note that the fully qualified name of the class must be the same in all of the files that constitute the partial class. In our case, the fully qualified name is WpfSimpleSteps.Employee.

Now we can modify our WriteToLog method in MainWindowViewModel to include the employee's full name:

```
var message = (returnType == typeof(Order))
  ? "Fetching Orders from Employee " + CurrentEmployee.FullName
  : "Fetching " + returnType;
```

```
Dim message = If(returnType Is GetType(Order), _
 "Fetching Orders from Employee " & CurrentEmployee.FullName, _
 "Fetching " & returnType)
```

Show busy

At this point we are assuming that you have downloaded the Extended WPF Toolkit and added a reference to your project. If not, see the Prerequisites below.

We've already pointed out that DevForce can execute queries asynchronously. In fact we are loading the initial list of employees asynchronously. It is always good to provide visual cues to the user when performing asynchronous tasks.

We'll make use of the BusyIndicator control from the WPF Toolkit to do this.

Open MainWindow.xaml and on the root element add another namespace alias:

```
xmlns:extToolkit="clr-namespace:Microsoft.Windows.Controls;assembly=WPFToolkit.Extended"
```

We're going to add the busy indicator as the child element of the Window. The current child will become the child of the busy indicator. The new structure of the XAML file will look like this:

```
<Window>
  <!-- many attributes removed for clarity -->>
  <extToolkit:BusyIndicator IsBusy="{Binding IsBusy}">
    <Grid>
    <!-- many attributes and elements removed for clarity -->
    </Grid>
  </extToolkit:BusyIndicator>
</Window>
```

When the *IsBusy* property is set to true on the busy indicator, it will overlay an animation, over the regular content.

We've already added a binding to the busy indicator, but there is no corresponding *IsBusy* property on the view model. Let's add that now.

Open MainWindowViewModel and add the *IsBusy* property as follows. Notice that we raise the *PropertyChanged* event as we did for all other properties. This will trigger WPF to display/hide the *BusyIndicator* animation whenever the *IsBusy* property changes:

```
private bool _isBusy;
public bool IsBusy {
 get { return _isBusy; }
 set {
    _isBusy = value;
    RaisePropertyChanged("IsBusy");
  }
}
```

```
Private _isBusy As Boolean
Public Property IsBusy() As Boolean
```

```
 Get
   Return _isBusy
 End Get
 Set(ByVal value As Boolean)
    _isBusy = value
    RaisePropertyChanged("IsBusy")
 End Set
End Property
```

We want to set *IsBusy* to true whenever we initiate a query and then set it to false whenever the results are returned. We can listen for the *Fetching* and *Queried* events on the *EntityManager* to help with this. Inside the constructor, we can add the following event handlers:

```
_mgr.Fetching += delegate { IsBusy = true; };
_mgr.Queried += delegate { IsBusy = false; };
```

```
AddHandler _mgr.Fetching, Sub() IsBusy = True
AddHandler _mgr.Queried, Sub() IsBusy = False
```

Run the application to see the busy indicator in action. The only place where we are loading data asynchronously is during the startup of the application. To enable async loading of navigation properties such as Orders, add the following line in the constructor.

```
Employee.PropertyMetadata.Orders.ReferenceStrategy = EntityReferenceStrategy.DefaultAsync;
```

Depending on how fast your query executes, you may not see the BusyIndicator very often.

Simple validation

Out of the box, DevForce provides us with basic validation. In generating your entity model, DevForce examined the schema for basic rules, such as a field not allowing nulls or having a maximum length. DevForce in turn generated validation attributes on these properties; so for example the LastName here is both required, and has a maximum length of 30 characters.

Default WPF validation

By default WPF listens for exceptions to notify bound UI controls of validation errors.

Let’s take a look at the binding in our XAML. You’ll notice that when we dragged the data source on to the Visual Studio Designer, it created bindings for the form fields that look something like this:

```
Text="{Binding Path=FirstName, Mode=TwoWay, ValidatesOnExceptions=true, NotifyOnValidationError=true}"
```

Notice the validation attributes it added. *ValidatesOnExceptions=true* tells WPF to look for an exception when it calls the property’s setter and interprets it as a validation error.

DevForce doesn't by default throw validation exceptions, but we can easily modify it to do so by changing the *ErrorNotificationMode* on the *VerifierEngine*.

Open MainWindowViewModel and add the following code in the constructor:

```
_mgr.VerifierEngine.DefaultVerifierOptions.ErrorNotificationMode =
  VerifierErrorNotificationMode.NotifyAndThrowException;
```

```
_mgr.VerifierEngine.DefaultVerifierOptions.ErrorNotificationMode = _
  VerifierErrorNotificationMode.NotifyAndThrowException
```

You must add the following using or Imports statement if not already there:

```
using IdeaBlade.Validation;
```

```
Imports IdeaBlade.Validation
```

Now run the application, navigate to an employee, erase the first name and tab off the field. You should now find yourself in the Visual Studio Debugger informing you that an exception occurred. Let’s ignore the debugger and just continue running the application. Notice the red border around the first name field. WPF took the exception and as instructed turned it around into a validation error.

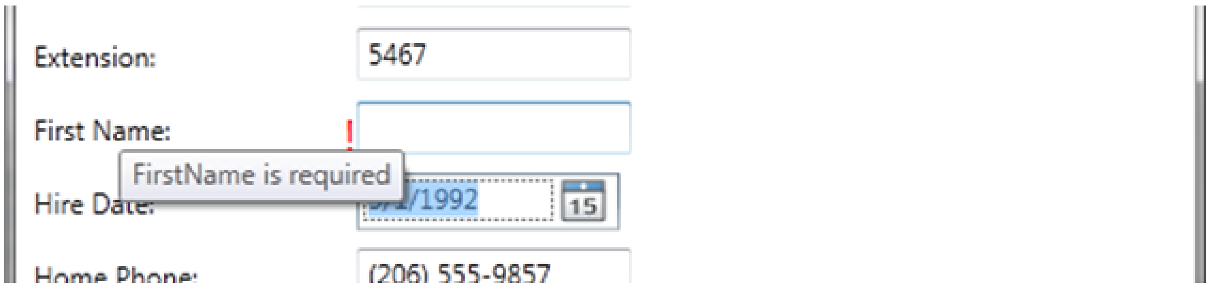
The default validation error template isn’t very exciting, though. It would be nice to get the specifics of the validation error, so we'll add an *ErrorTemplate*.

Switch back to MainWindow.xaml and add the following right below the <Window /> element:

```
<Window.Resources>
  <Style TargetType="{x:Type TextBox}">
    <Setter Property="Validation.ErrorTemplate">
      <Setter.Value>
        <ControlTemplate>
          <DockPanel>
            <TextBlock Foreground="Red" FontSize="20">!</TextBlock>
            <AdornedElementPlaceholder/>
          </DockPanel>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
    <Style.Triggers>
      <Trigger Property="Validation.HasError" Value="true">
        <Setter Property="ToolTip" Value="{Binding RelativeSource={RelativeSource Self},
          Path=(Validation.Errors)[0].ErrorContent}"/>
      </Trigger>
    </Style.Triggers>
  </Style>
</Window.Resources>
```

The above defines a style that we apply to all text boxes. Instead of the red border we are displaying a red exclamation mark and set the tooltip with the error message.

Run the application again and repeat the scenario. If you mouse over the first name field now you will see the tooltip with the validation error.



Default DevForce validation

Throwing exceptions for validation errors is not a particularly good practice, so let's look at a better way of doing it.

Let's go back to our view model and **remove** the following line of code from the constructor. We no longer want the *EntityManager* to throw an exception when a validation error occurs; instead we'll use the DevForce default mode of *Notify*.

```
_mgr.VerifierEngine.DefaultVerifierOptions.ErrorNotificationMode =
  VerifierErrorNotificationMode.NotifyAndThrowException;
```

```
_mgr.VerifierEngine.DefaultVerifierOptions.ErrorNotificationMode = _
  VerifierErrorNotificationMode.NotifyAndThrowException
```

We need to change the binding on our form elements too, so that ValidatesOnDataErrors is set to true.

Open MainWindow.xaml and switch to the XAML view. Replace all occurrences of ValidatesOnExceptions=true with ValidatesOnDataErrors=true. With this change the bindings will now listen for errors raised by an interface instead of exceptions.

```
Text="{Binding Path=FirstName, Mode=TwoWay, ValidatesOnDataErrors=true, NotifyOnValidationError=true}"
```

Now run the application, navigate to an employee, erase the first name and tab off the field. Notice how we no longer occur an unhandled exception. The validation is much cleaner and we separate data errors from application exceptions.

Next steps

This concludes the tour of DevForce WPF. We hope that this has given you a good introduction to what DevForce 2010 can do for you.

We want to remind you that this tutorial is intended to demonstrate basic features and ease-of-use. It is not production-ready code, nor is it meant to be an example of general best practices in software development.

Of course, it represents only the beginning of what you can do with DevForce. To learn more:

- Client development
- EntityManager Overview
- The query life cycle
- Adding business logic
- Asynchronous queries and saves
- DevForce entities and the UI
- Validation overview

Prerequisites

The user interface for the application built during this tour uses a BusyIndicator component supplied by the Extended WPF Toolkit. You can download the assembly from:

http://wpftoolkit.codeplex.com/

After downloading the assembly, place it anywhere in your solution directory and add a reference to your project. Because the assembly is coming from the Web, you may have to unlock it in order for Visual Studio to be able to open it. Right-click on the assembly in Windows Explorer and choose Properties. If the file is locked you will be presented with an Unlock button at the bottom of the General tab. Click the Unlock button to unlock the file.