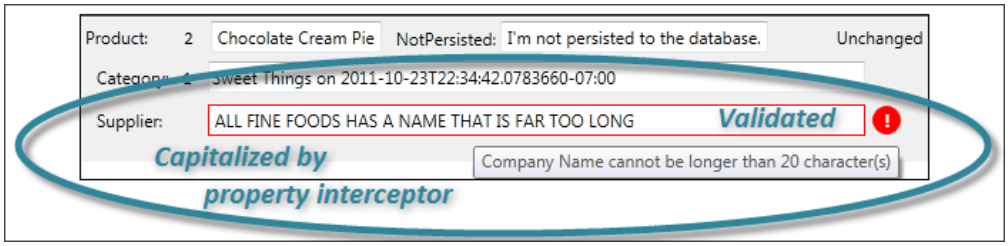
Contents

We've seen how to bind the UI to DevForce entity classes and properties thanks to infrastructure that DevForce embeds in these classes when rewriting them with AOP.

In this segment we'll see that this same infrastructure supports validation – object and property level validation on both client and server – to ensure the integrity of user input.

Entity property getters and setters can be intercepted by your custom code to perform your business logic and application magic on property values.



Finally, we'll wire up Add, Delete, and Save buttons to see how these operations resolve into calls upon entities and the *EntityManager*. We'll make visible the tacit *EntityAspect* property to implement the delete operation more intuitively and refactor some of the repetitive code into a custom base class.

- **Platform:** WPF
- **Language:** C#
- **Download:** Code First Walkthrough

Add Validation and UI Hints

If you look at properties of entities generated by DevForce from an Entity Data Model file (EDMX), you'll find a healthy number of attributes. Some of them may be useful in your Code First class:

```
// Examples of useful generated attributes
[Bindable(true, BindingDirection.TwoWay)]
 [Display(Name = "Name", AutoGenerateField = true)]
 [StringLengthVerifier(MaxValue = 20, IsRequired = true)]
```

When an entity is a data source for UI data binding, some controls configure themselves appropriately when they detect the *Bindable* and *Display* attributes, a trick that can save screen development time.

The *StringLengthVerifier* is a DevForce validation attribute that tells DevForce to both require a value and limit its length.

In Code First models you may prefer to use the *System.ComponentModel.DataAnnotations* validation attributes, since they both perform validation and control how the entity is mapped to a database table.

1. **Add usings** to the top of the **Model** class file.

```
using System.ComponentModel;
using IdeaBlade.Validation;
```

2. **Go to the Supplier class**.

3. **Add attributes** to the *Supplier.CompanyName*.

```
[Bindable(true, BindingDirection.TwoWay)]
```

```
[Display(Name = "Company Name", AutoGenerateField = true)]
[StringLengthVerifier(MaxValue = 20, IsRequired = true)]
public string CompanyName { get; set; }
```

Notice the display name is “Company Name”. A UI control could use that for a field or column label. The DevForce “verifier” will use it in validation error messages.

Light up a tooltip for validation error

We’d like to display a strong visual cue and tooltip message when the user enters invalid data.

1. **Paste the following** (**WPF only**) *TextBox* style into the ***<Application.Resources>*** tag of the application’s ***App.xaml*** file in order to see validation errors displayed in a tooltip.

```
<!--
  Validation Error Template Style for TextBox
  Courtesy Edwin Foh:
  http://codeblitz.wordpress.com/2009/05/08/wpf-validation-made-easy-with-idataerrorinfo/
-->
<Style TargetType="{x:Type TextBox}">
   <Setter Property="VerticalAlignment" Value="Center" />
   <Setter Property="Margin" Value="0,2,40,2" />
   <Setter Property="Validation.ErrorTemplate">
      <Setter.Value>
         <ControlTemplate>
            <DockPanel LastChildFill="true">
               <Border Background="Red" DockPanel.Dock="right"
                     Margin="5,0,0,0" Width="20" Height="20" CornerRadius="10"
  ToolTip="{Binding ElementName=customAdorner, Path=AdornedElement.(Validation.Errors)[0].ErrorContent}">
                  <TextBlock Text="!" VerticalAlignment="center" HorizontalAlignment="center"
                        FontWeight="Bold" Foreground="white" />
               </Border>
               <AdornedElementPlaceholder Name="customAdorner" VerticalAlignment="Center" >
                  <Border BorderBrush="Red" BorderThickness="1" />
               </AdornedElementPlaceholder>
            </DockPanel>
         </ControlTemplate>
      </Setter.Value>
   </Setter>
</Style>
```

We do not need to add this style in Silverlight where we get a validation error tooltip automatically.

We’ll demonstrate the validation error effect at the end of this lesson.

Add a Property Interceptor

A DevForce AOP entity property supports property interceptors which are custom code you write to do whatever you need when the value accessed or set.

The easy way to add a one-off property interceptor is as a method of the class, decorated with a DevForce property interceptor attribute such as *AfterGet*.

1. **Add an interceptor** for the *Supplier.CompanyName* just below that property.

```
// Example property interceptor
[AfterGet("CompanyName")]
internal void UppercaseNameAfterGet(PropertyInterceptorArgs<Product, String> args)
{
   if (null != args.Value)
   {
      args.Value = args.Value.ToUpper();
   }
}
```

Requires *using IdeaBlade.Core;*.

This contrived “get interceptor” returns an upper-cased version of the *CompanyName* value. Notice that it can be non-public … and probably should be as there is no good reason for application code to call this method.

DevForce discovers the interceptors by reflection. The method could be *private* in a full .NET model but not in Silverlight where private reflection is forbidden. Marking it *internal* gives DevForce a chance to find it in Silverlight when you make your model assembly visible to DevForce.

Add, Delete, and Save

Let's add these capabilities to our ViewModel first and then bind them to buttons on the screen.

Add AddProductAction

1. **Go** to the bottom of the *MainWindowViewModel* class.

2. **Add the *AddProductAction*** method.

```
public void AddProductAction()
{
   var newProduct = AddNewProduct();
   Log("Adding new product " + newProduct.ProductId);
   Products.Add(newProduct);
   SelectedProduct = newProduct;
}
```

The code calls *AddNewProduct* again as we did in *AddTestData()*, logs that fact, and adds the new product to the *Products IObservableCollection<Product>* so it appears in the *ListBox*.

Finally, it sets the *SelectedProduct* property which we haven't defined yet. *SelectedProduct* is how we will communicate with the View about which product is selected … either by the use or programmatically by the ViewModel.

While we're thinking about that, we realize that we want the first product added in *AddTestData()* to be the selected product when the user sees the list.

3. **Update *AddTestData*** to set *SelectedProduct* for the first product added.

```
SelectedProduct = AddNewProduct("Chocolate Cream Pie");
```

Now it's time to define *SelectedProduct*.

4. **Add the *SelectedProduct*** property; put it near the *Products* property.

```
private Product _selectedProduct;
public Product SelectedProduct
{
  get { return _selectedProduct; }
  set
   {
      _selectedProduct = value;
      RaisePropertyChanged("SelectedProduct");
   }
}
```

The Product *ListBox* in *MainWindow.xaml* is binding its *SelectedItem* property to the *ViewModel's SelectedProduct*.

```
<ListBox x:Name="productsListBox" Grid.Row="2" Margin="0,10,0,0"
        SelectedItem="{Binding SelectedProduct, Mode=TwoWay}"
```

Implement INotifyPropertyChanged

When we change the *SelectedProduct* in the ViewModel, we need to notify the View so it selects the proper product. To notify the View, must raise the *PropertyChanged* event of *INotifyPropertyChanged*.

ViewModels almost always implement *INotifyPropertyChanged* eventually.

1. **Make *MainWindowViewModel* derive from *INotifyPropertyChanged***.

```
public class MainWindowViewModel : INotifyPropertyChanged
```

Requires "*using System.ComponentModel;*"

2. **Implement *INotifyPropertyChanged*** at the bottom of MainWindowViewModel.

```
public event PropertyChangedEventHandler PropertyChanged;
private void RaisePropertyChanged(string propertyName)
{
    var pc = PropertyChanged;
   if (null == pc) return;
    pc(this, new PropertyChangedEventArgs(propertyName));
}
```

Add the SaveAction

This method mimics the similar lines in *AddTestData()*.

```
public void SaveAction()
{
    Log("Saving changes");
    Manager.SaveChanges();
}
```

Add the DeleteProductAction

```
public void DeleteProductAction()
{
    var currentProduct = SelectedProduct as Product;
   if (null == currentProduct)
    {
        Log("No selected product; nothing deleted");
    }
   else
    {
        Log("Deleting product " + currentProduct.ProductId);
        Products.Remove(currentProduct);
        currentProduct.EntityAspect.Delete();
    }
}
```

If there is no product selected, we log that fact and bail out.

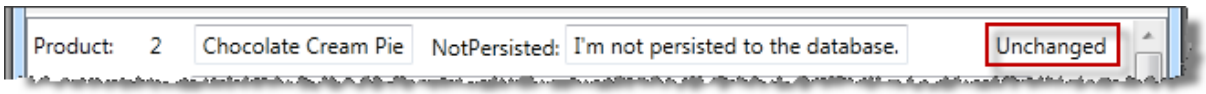
If there is a selected product, we log that fact, and remove it from the *Products IObservableCollection<Product>* so that it disappears from the *ListBox*.

Then we ask the selected product to delete itself (to schedule itself for deletion when next we save) by way of the product's *EntityAspect* property.

What is EntityAspect ?

EntityAspect is the gateway to every DevForce entity's hidden infrastructure. We didn't include an *EntityAspect* property in any of the entity classes we wrote. But it is there in every DevForce AOP entity, injected by the DevForce AOP assembly rewrite build process.

In fact, all along our view has been binding to it and displaying its current *EntityState* value.



Data binding works by runtime reflection which is why it has no trouble discovering the *EntityAspect* property.

But developers can't refer to this property at design time because it doesn't exist when the compiler is looking at the source code. *EntityAspect* is *added* after the compiler has finished compiling the entity.

That's why the critical line in the DeleteProductAction method won't compile. We could cast to get around this problem:

```
currentProduct.EntityAspect.Delete(); // won't compile yet
((IEntity) currentProduct).EntityAspect.Delete();// Works!  But we'll do it differently
```

That's ugly but acceptable … if we only write it once. However, experienced DevForce developers find themselves reaching for *EntityAspect* frequently.

Make a BaseEntity class with EntityAspect

We recommend that you make an *EntityAspect* property available to all entities by means of your own entity base class. All of your entity classes can inherit from this base class to gain access to *EntityAspect* … and any additional domain logic you think all entities should have.

1. **Go** almost to the bottom of the **Model** file.

2. **Add a *BaseEntity*** class as follows:

```
[ProvideEntityAspect]
public abstract class BaseEntity : IEntity
{
  public EntityAspect EntityAspect { get { throw new NotImplementedException(); } }
}
```

The implementation of *EntityAspect*'s getter doesn't matter; DevForce AOP will find it and replace it with the actual *EntityAspect* property when it rewrites the class. Learn more about EntityAspect and why we derived from IEntity.

3. While we're at it, we decorate the base class with the [*ProvideEntityAspect*] attribute that tells DevForce this is an AOP entity.

This optional step enables us to remove that attribute from all derived classes … and remove a step we could easily forget when we write more classes in future.

In fact, you must remove this attribute from all descendent classes because only one class in a class hierarchy is allowed to have this attribute. A build error reminds you of this point if you neglect it.

4. **Derive all entity classes from *BaseEntity*** (*Category*, *Product*, and *Supplier*).

5. **Remove [*ProvideEntityAspect*]** attribute from these entity classes.

6. **Build** the project.

Add the Buttons

1. **Open** *MainWindowViewModel.xaml*.

2. **Add** a fourth <*RowDefinition*> to the layout grid; the RowDefinitions are:

```
<Grid.RowDefinitions>
   <RowDefinition Height="40" />
   <RowDefinition Height="Auto" />
   <RowDefinition Height="*" />
   <RowDefinition Height="Auto" />
</Grid.RowDefinitions>
```

3. **Add a row of buttons** to the bottom of the layout grid, just below the products ListBox.

```
<StackPanel Grid.Row="3" Orientation="Horizontal" HorizontalAlignment="Center" VerticalAlignment="Center">
  <Button x:Name="AddButton"  MinWidth="40" Margin="0,2,8,2" Click="AddButton_Click">Add</Button>
  <Button x:Name="DeleteButton" MinWidth="40" Margin="0,2,8,2" Click="DeleteButton_Click">Delete</Button>
  <Button x:Name="SaveButton" MinWidth="40" Margin="0,2,8,2" Click="SaveButton_Click">Save</Button>
</StackPanel>
```

Each button has a click handler which we next define in the code behind.

4. **Right-Click | View Code**

5. **Paste** the following click handlers at the bottom of the class.

```
private void AddButton_Click(object sender, RoutedEventArgs e)
{
   _viewModel.AddProductAction();
}
private void DeleteButton_Click(object sender, RoutedEventArgs e)
{
   _viewModel.DeleteProductAction();
}
private void SaveButton_Click(object sender, RoutedEventArgs e)
{
   _viewModel.SaveAction();
```

```
}
```

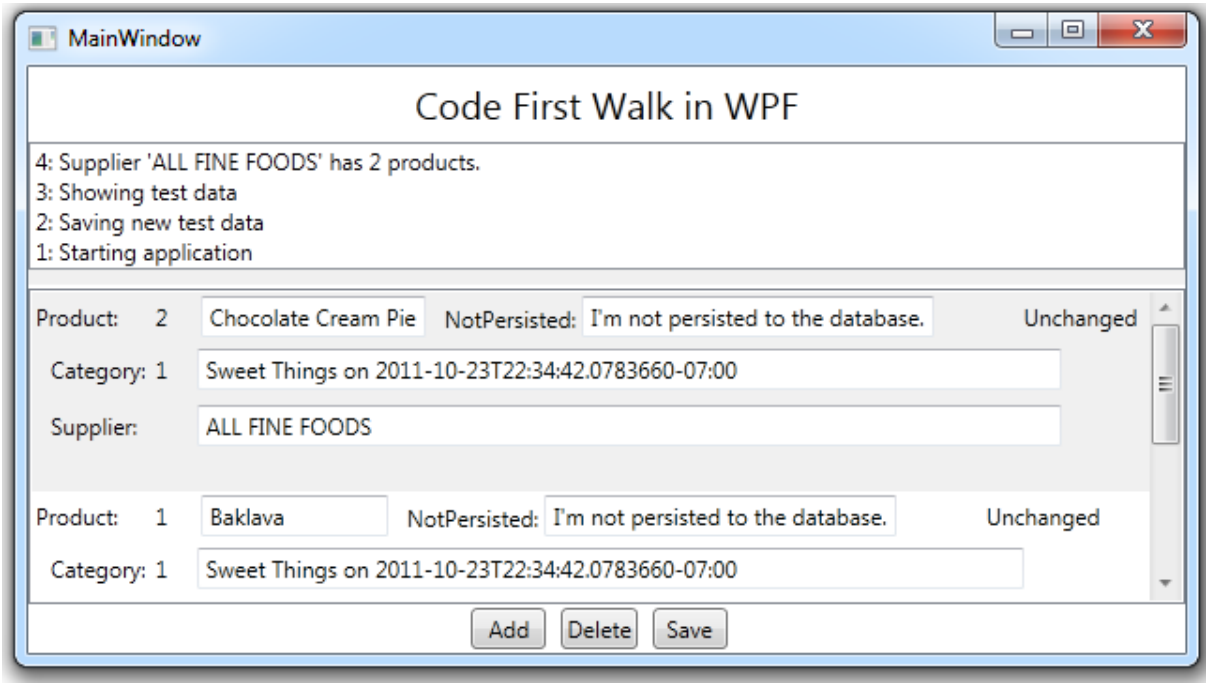
That's simple switchboard code, connecting the click to the appropriate action in the ViewModel.

6. **Add the _viewModel** private variable and **assign it in the constructor**.

```
private readonly MainWindowViewModel _viewModel;
public MainWindow()
{
    InitializeComponent();
    DataContext = _viewModel = new MainWindowViewModel();
}
```

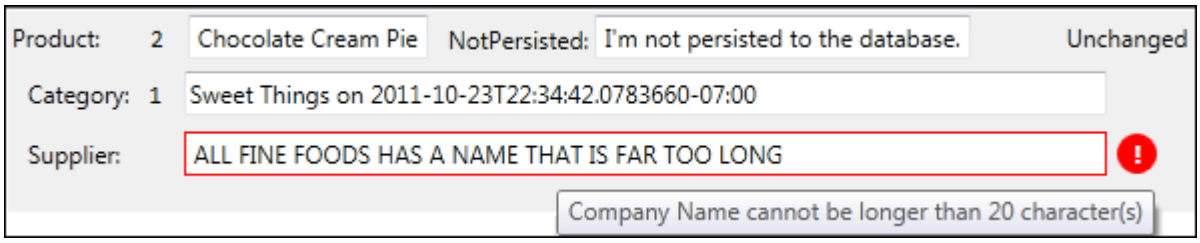
Try the UI

1. **Build and run** [F5] … you should see:

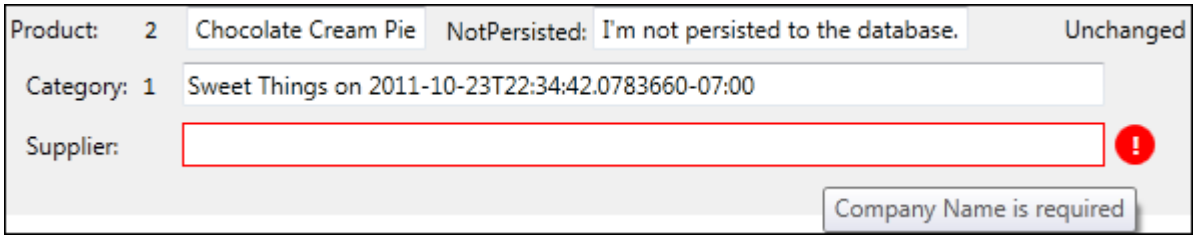


Notice that the *Supplier.CompanyName* property interceptor is capitalizing "*ALL FINE FOODS*" (which is actually "*All Fine Foods*" in the data), both in the log at the top and for each *Product* entity displayed.

The string length validation displays an error if you enter a long Supplier name:

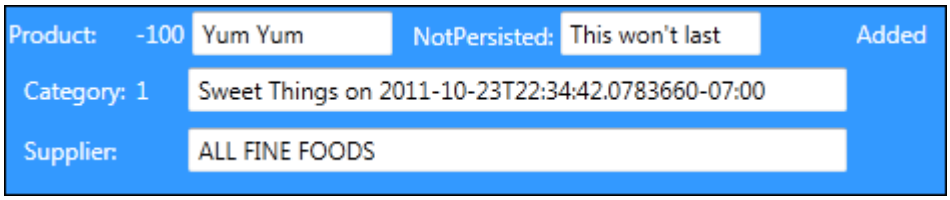


… or if you clear the name:



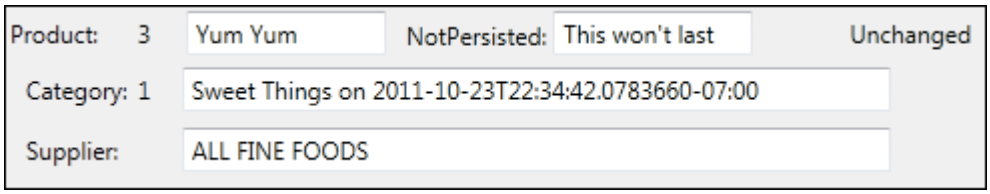
2. **Add a new Product** and scroll to the bottom.

3. **Enter** a different product name and **replace** the *NotPersisted* text with something.



Notice the temporary *ProductId* (-100) and the *EntityState* of "Added".

4. **Press the Save** button.



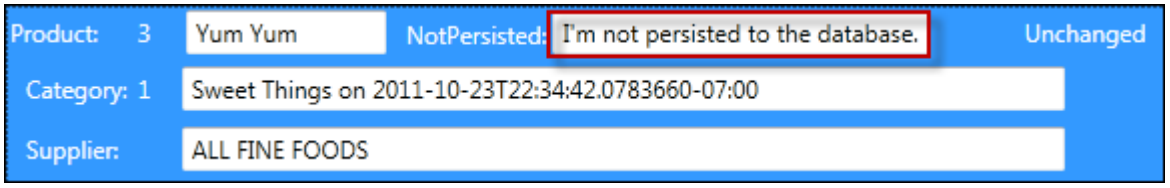
The save succeeds. The *ProductId* receives its permanent value and the *EntityState* becomes "Unchanged". The NotPersisted value remains as we last modified it; this product entity is the same physical object in cache. That won't change until we evict it or terminate the application.

5. **Terminate** the application and **re-launch** it.

   There are five products in the database:

   - The two initial products from session #1
   - The added product from session #1
   - The two new products in this session #2

6. **Scroll to product #3** and **select it**.



This time the product entity was materialized fresh from the database … which does not have a value for the *NotPersisted* property. Consequently, *NotPersisted* is its default value.

7. **Delete** product #3 and **Save**.

8. **Terminate** the application and **re-launch** it.

9. There should be six products: the two new products from session #3 and "Yum Yum" is gone.