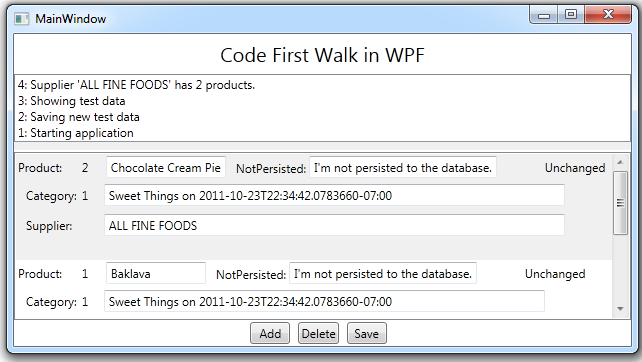
CodeFirstWalk Overview

This CodeFirstWalk sample shows you how to build a [DevForce Code First](http://drc.ideablade.com/xwiki/bin/view/Documentation/code%2Dfirst) WPF application from scratch.



The application itself is not the focus; we are not striving to make a useful application nor demonstrating best practices in UI design or development. Our purpose is to lead you through a sequence of steps that introduces the essentials of writing and consuming an entity model in DevForce Code First style.

The exercise proceeds in six stages, each of which can be completed in less than fifteen minutes, each of which leaves you with a working application.

**From “New Project” to Running** – Build a console application in WPF clothing. Define a single-entity (*Category*) model, add instances of it to a development database, query them back, format them as strings, and display them on screen as logged messages.

**Update the Model and Bind to it** – Models evolve. See that it’s easy to add a new entity type (*Product*). Replace the “*Console.Writeline*” approach with a WPF form that bind to the entities. See that DevForce AOP entities have data binding support wired in automatically and invisibly.

**Explicit Entity Mapping** – When the Entity Framework naming conventions don’t convey the database schema you need, use Entity Framework mapping configuration. We add another new type (*Supplier*) and map declaratively with attributes. Then we create our first EF *DbContext* class and map imperatively with the Fluent API.

**Validation and Property Interception** – DevForce AOP entities are wired to support object and property validation as we show by constraining the *Supplier.CompanyName* with string length validation and force it to display the name in uppercase with a property interceptor. “Add”, “Delete”, and “Save” buttons demonstrate basic operations and help explore the implications of the entity features we’ve built.

**Create a separate Model Project** – Most developers quickly relocate the model classes to a Model project that is separate from the application project as we do in this five minute stage.

From “New Project” to Running

# Create New Project

**File | New | Project | DevForce 2010 | DevForce WPF Application**

Name it “CodeFirstWalk”

# MainWindow.xaml – our first taste of WPF UI

Open *MainWindow.xaml*. Make sure the XAML pane is showing.

Make the window wider – width=800 is ok.

Title="MainWindow" Height="350" Width="800">

Add a two-row grid with a title in the top row and a *ListBox* to display messages in the second row. We’re using data binding right away, binding the *ListBox*’s *ItemSource* to a collection property called “***Messages***”.

<Grid>

    <Grid.RowDefinitions>

        <RowDefinition Height="40" />

        <RowDefinition Height="Auto" />

     </Grid.RowDefinitions>

    <TextBlock Text="Code First Walk in WPF"

FontSize="20" TextAlignment="Center" VerticalAlignment="Center"/>

    <ListBox x:Name="messages" Grid.Row="1" ItemsSource="{Binding Messages}" />

</Grid>

We’ll return to the UI soon. For now … **Close all windows** (Alt-W, L)

# Reference the Entity Framework Code First Library

[**Use NuGet**](http://drc.ideablade.com/xwiki/bin/view/Documentation/code%2Dfirst%2Dentity%2Dclasses#HAddingEntityFrameworklibrarytoyourproject) to acquire the "EntityFramework.dll", the EF Code First library, and to reference that library in the project.

# Add DevForce CF Marker File

**Add | New** Item (or [ctrl+shift+A]) | **DevForce 2010** | **DevForce CodeFirst File**

**Name** it “DevForce.cf” (although the name doesn’t matter at all)

The CF marker file enables DevForce code first processing. The template also adds references to the DevForce and .NET libraries that we'll need.

**Close** the editor window

# Create a Model

**Add | New** Item | **Class**

Name it “Model”

**Delete** the template generated “Model” class

**Add** the *Category* class below

[ProvideEntityAspect]

public class Category

{

    public int CategoryId { get; set; }

    public string CategoryName { get; set; }

}

The *ProvideEntityAspect* attribute requires us to add “*using IdeaBlade.Aop;*”

# Add custom EntityManager

We’ll start by creating it in this same Model.cs file.

You are welcome to move any or all of these classes into their own class files at any time.

**At the top** of the file, **define** the following custom “*ProductEntities*” *EntityManager* class

public class ProductEntities : EntityManager {}

The *EntityManager* class requires “*using IdeaBlade.EntityModel;*”

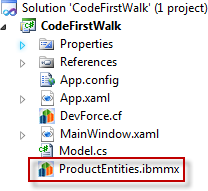
Add the “Categories” *EntityQuery* property so it’s easier for us to build queries of *Category* entities.

public EntityQuery<Category> Categories { get; set; }

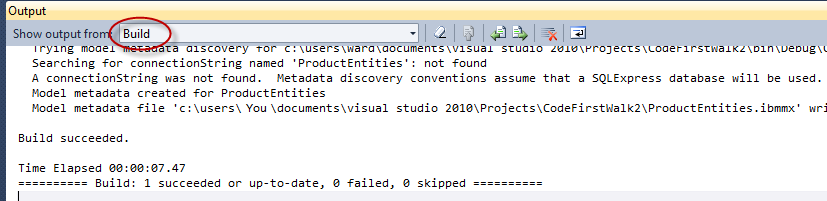
# Build First Time

Build the solution (“Ctrl-Shift-B”) … this is the first build with Code First model

Notice it generates “*ProductEntities.ibmmx*” into the project



Look at the Build output in the Output window



**You should always build soon (immediately?) after changing the model and confirm that DevForce (re)generated the metadata “.ibmmx” file.**

# Create a ViewModel

Yes, we’ll use the MVVM pattern in this example. It’s super easy.

**Add** | New Item | **Class**

Name it “MainWindowViewModel”

Make it a **public** class

**Add** a parameterless constructor

Call the following four methods:

Start();

SetManager();

AddTestData();

ShowTestData();

Let Visual Studio (or R#) create stubs for these methods (work bottom up so the stubs are created top down).

# Start()

**Delete** the line throwing a *NotImplementedException*

We’ll initialize a *Messages* collection property to hold our string messages (remember back in the UI we said we’d bind to it).

Messages = new ObservableCollection<string>();

The *ObservableCollection* class requires “*using System.Collections.ObjectModel*;”

Next call a *Log(…)* method that we’ll soon write

Log("Starting application");

Let Visual Studio (or R#) **create** the stub for the *Log* method

Add a **public** ***Messages* auto-property** just below the *Start* method.

At this point, this section looks like so:

 private void Start()

 {

     Messages = new ObservableCollection<string>();

     Log("Starting application");

 }

 public ObservableCollection<string> Messages { get; set; }

 private void Log(string startingApplication)

 {

     throw new **NotImplementedException**();

 }

Implement the *Log* method such that it **inserts** a numbered message at the front of the *Messages* collection. This gambit causes more recent messages to appear at the top our messages *ListBox*. The implementation follows:

private void Log(string message)

{

    message = String.Format("{0}: {1}", ++\_messageCounter, message);

    Console.WriteLine(message);

    Messages.Insert(0, message); // new messages in front

}

private int \_messageCounter;

We are logging to the Visual Studio *Console* window for good measure.

# SetManager()

We create and initialize your custom *EntityManager* here.

We prefer to put *EntityManager* creation and query and save logic such as you’ll see here in a *Repository* or “*DataServices*” class. Not in this demo; that’s an exercise for the future.

**Delete** the line throwing a *NotImplementedException*

**Instantiate** your ProductEntities class and assign it to a *Manager* property.

Add a **private** ***Manager* auto-property** just below the *SetManager* method.

The server may not always be kind. An exception thrown on the server will be sent to the client and surfaced here as an *EntityServerException*. We can catch them here in an *EntityServerError* event handler which will (a) indicate that we’re handling the error, (b) log the problem, and (c) undo any pending changes in the EntityManager that may be the source of the problem.

We can define the handler in a lambda. The result is as follows:

private void SetManager()

{

    Manager = new ProductEntities();

    Manager.EntityServerError += (s, e) =>

    {

        e.Handled = true; // we're dealing with it here

        Log("Server error: " + e.Exception.Message);

        Manager.RejectChanges(); // undo pending changes

    };

}

private ProductEntities Manager { get; set; }

You might want to **add breakpoint** to the first line in the lambda expression in case we get an error and want to see the entire exception.

# AddTestData()

**Delete** the line throwing a *NotImplementedException*

Create a new *Category* entity and set its name.

var cat = new Category {CategoryName = "Sweet Things on " + DateTime.Now.ToString("o")};

**Add** it to the *Manager*, **Log** the fact that we are saving now, and call ***SaveChanges***

Manager.AddEntity(cat);

Log("Saving new test data");

Manager.SaveChanges();

The *AddTestData* method should look like this:

 private void AddTestData()

 {

    var cat = new Category { CategoryName = "Sweet Things on " + DateTime.Now.ToString("o") };

    Manager.AddEntity(cat);

    Log("Saving new test data");

    Manager.SaveChanges();

 }

# ShowTestData()

**Delete** the line throwing a *NotImplementedException*

We’ll log the fact that we’re showing test data

Log("Showing test data");

We query for all Category entities. This will pick up both the one we just added and any others that are lurking in the database from previous application sessions.

**Add this query**

var cats = Manager.Categories.ToList(); // get 'em all

Log the categories we fetched, using a helper method

cats.ForEach(LogCats);

*ToList()* and *ForEach()* require “*using System.Linq;*”

Let Visual Studio (or R#) **create** the stub for the *LogCats* method

* Change the *obj* parameter name to “***cat***”
* Set **format string**: var fmt = "Category {0} Name={1}";
* **Call Log**: Log(string.Format(fmt, cat.CategoryId, cat.CategoryName));

The *ShowTestData* and *LogCats* methods should look like this

 private void ShowTestData()

 {

     Log("Showing test data");

     var cats = Manager.Categories.ToList(); // get 'em all

     cats.ForEach(LogCats);

 }

 private void LogCats(Category cat)

 {

     var fmt = "Category {0} Name={1}";

     Log(string.Format(fmt, cat.CategoryId, cat.CategoryName));

 }

# Create the ViewModel in the View

Time to tie the View (*MainWindow.xaml*) to the ViewModel (*MainWindowViewModel.cs*). We’ll do that in the view’s code behind.

That is not often the best choice but it’s good enough for our demo.

**Open** *MainWindow.xaml.cs*

**Delete** every using except “System.Windows” (optional)

In the constructor, **assign** the view’s *DataContext* with a new instance of the *MainWindowViewModel* ViewModel class. The constructor looks like this when you’re done.

public MainWindow()

{

    InitializeComponent();

    DataContext = new MainWindowViewModel();

}

# Run in Debug [F5]

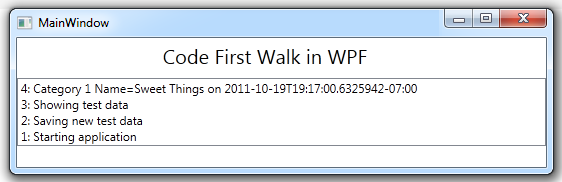
It builds again. This time the build messages in the output window end as follows:

Model metadata created for ProductEntities

Model metadata for ProductEntities.ibmmx is unchanged

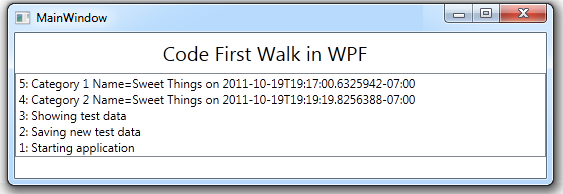
The metadata “*ibmmx*” file is unchanged. But the assembly has been changed … because now that **metadata file is embedded** in the *CodeFirstWalk.exe* assembly.

The application runs. And it works!



The messages appear top down in reverse order. At the top is the Category we just created.

Run the app again and you’ll see two Categories in the application window, the one we created in the previous session and the new one we just added.



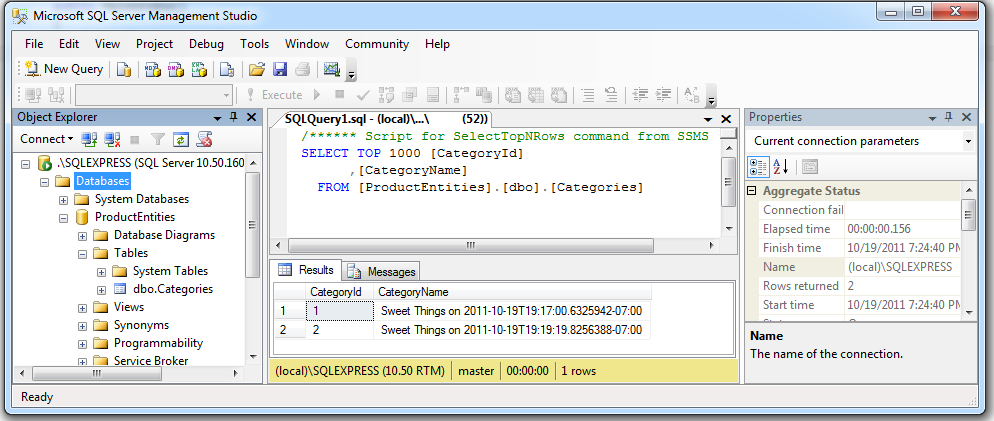
How is that possible? We haven’t defined a database. We haven’t even named a database.

In the first session, when the *AddTestData()* method asked the *EntityManager* to save the new *Category*, the *EntityManager* issued the first command that requires a database. Somewhere in the chain of events that followed, DevForce gave the Entity Framework the name of a connection string. In our example, it supplied the name of the *EntityManager* class (“*ProductEntities*”).

Because there is no “ProductEntities” connection string in the *app.config* file, the Entity Framework Code First creates a database to match our entity model in SQL Server Express and calls it “ProductEntities”.

As it stands, you must have Microsoft **SQL Server Express** installed and running or the Entity Framework can’t create the missing database for you. [You can change that default](http://drc.ideablade.com/xwiki/bin/view/Documentation/advanced%2Ddatabase%2Dconnections#HLivingwithoutSQLServerExpress).

Examine the database in SQL Server Management Studio



The database exists and has a “Categories” table with the two rows displayed in the UI.

# Summary

In this part of the “Code First Walk” we

* Created a new DevForce WPF Application
* Defined a DevForce entity model entirely in code using “Code First”
* Put our entity access code in a simple ViewModel and bound that ViewModel to the View.
* Added new entities, saved them, and queried them back from the database.
* Let the Entity Framework create a database to match our Model.

Update the Model and Bind to it

DevForce AOP entities are more than bags of data stored in a database. They can participate in client UIs. They have hidden infrastructure for caching, navigation to related entities, for data binding and validation and change tracking.

Last time we established that an entity class could be built entirely in code. Instances of that type could be created and saved. But we didn’t show related entities and we merely dumped entity property values into strings displayed on screen.

This time, we show that you can add entities to the model quickly, relate them to each other, and bind to them in XAML.

# Add a Product entity to the Model

**Open** the Model class and **add** the following *Product* class definition:

[ProvideEntityAspect]

public class Product

{

    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public int CategoryId { get; set; } // Foreign key for "Category"

    public Category Category { get; set; }

}

Notice that *Product* has a navigation property to its parent *Category*. We could add a navigation property from *Category* back to its child products … but we do not have to do so and we won’t in this example.

**Add** a convenience *Product* query property to *ProductEntities*, which is our custom *EntityManager*.

 public EntityQuery<Product> Products { get; set; }

Build again to save the model and re-generate metadata

**Tip**: build after every change to the model. Look in the Output window and confirm that the metadata “.ibmmx” file was regenerated.

# Update the View

**Open** the View, *MainWindow.xaml*. Make sure the XAML pane is showing.

**Add a row** to the grid by adding a *RowDefinition*. There should be three now.

<Grid.RowDefinitions>

    <RowDefinition Height="40" />

    <RowDefinition Height="Auto" />

    <RowDefinition Height="\*" />

</Grid.RowDefinitions>

**Add a GridSplitter** to separate the second and third rows and enable the user to show more or less of row #2.

<GridSplitter Grid.Row="2" HorizontalAlignment="Stretch"  VerticalAlignment="Top" Height="8"/>

**Add a *ListBox*** to row #2.

**Bind the *ListBox.ItemSource*** to the ***Products*** collection property in the ViewModel; we haven’t written that property yet but we will soon.

**Bind the *ListBox.SelectedItem***to another future ViewModel property, the ***SelectedProduct***, such that the user’s choice is communicated to the ViewModel.

**Bind the *ListBox.ItemTemplate*** to the soon-to-be-added ***ProductTemplate*** in order to present a well-formatted product to the user for each item in the *Products* collection.

Here’s the finished *ListBox* xaml.

<ListBox x:Name="productsListBox" Grid.Row="2" Margin="0,10,0,0"

          SelectedItem="{Binding SelectedProduct, Mode=TwoWay}"

          ItemsSource="{Binding Products}"

          ItemTemplate="{StaticResource ProductTemplate}"/>

**Scroll** nearly to the top of the xaml, just above the <Grid> tag.

**Add** the following *Product* *DataTemplate* in a *Window.Resources* tag:

<Window.Resources>

<DataTemplate x:Key="ProductTemplate">

        <Grid>

            <Grid.RowDefinitions>

                <RowDefinition Height="Auto" />

                <RowDefinition Height="Auto" MinHeight="30"/>

                <RowDefinition Height="Auto" MinHeight="30"/>

                <RowDefinition Height="20" />

            </Grid.RowDefinitions>

            <Grid.ColumnDefinitions>

                <ColumnDefinition Width="Auto" />

                <ColumnDefinition Width="Auto" />

                <ColumnDefinition Width="Auto" />

                <ColumnDefinition Width="Auto" />

                <ColumnDefinition Width="Auto" />

                <ColumnDefinition Width="Auto" />

            </Grid.ColumnDefinitions>

            <TextBlock Text="Product: " Grid.Row="0" Grid.Column="0"

VerticalAlignment="Center" />

            <TextBlock Text="{Binding ProductId}" Grid.Row="0" Grid.Column="1"

VerticalAlignment="Center" Margin="2,0,2,0" MinWidth="20"/>

            <TextBox Text="{Binding ProductName, Mode=TwoWay, ValidatesOnDataErrors=True}"

Grid.Row="0" Grid.Column="2"

VerticalAlignment="Center" Margin="2,0,2,0" MinWidth="100"/>

            <TextBlock Text="NotPersisted: " Grid.Row="0" Grid.Column="3"

VerticalAlignment="Center" Margin="8,2,0,0"/>

            <TextBox Text="{Binding NotPersisted, Mode=TwoWay, ValidatesOnDataErrors=True}"

Grid.Row="0" Grid.Column="4"

VerticalAlignment="Center" MinWidth="100"/>

            <TextBlock Text="{Binding EntityAspect.EntityState}" Grid.Row="0" Grid.Column="5"

   VerticalAlignment="Center" Margin="8,0,0,0"/>

            <TextBlock Text="Category: " Grid.Row="1" Grid.Column="0"

VerticalAlignment="Center" Margin="8,0,0,0"/>

            <TextBlock Text="{Binding CategoryId}" Grid.Row="1" Grid.Column="1"

VerticalAlignment="Center" Margin="2,0,2,0" MinWidth="20"/>

            <TextBox Text="{Binding Category.CategoryName, Mode=TwoWay, ValidatesOnDataErrors=True}"

Grid.Row="1" Grid.Column="2"

  VerticalAlignment="Center" Grid.ColumnSpan="4" />

            <TextBlock Text="Supplier: " Grid.Row="2" Grid.Column="0"

VerticalAlignment="Center" Margin="8,0,0,0" />

            <TextBox Text="{Binding Supplier.CompanyName, Mode=TwoWay, ValidatesOnDataErrors=True}"

Grid.Row="2" Grid.Column="2"

VerticalAlignment="Center" Grid.ColumnSpan="4"/>

        </Grid>

    </DataTemplate>

</Window.Resources>

Wow … that’s a lot of Xaml !

We won’t need all of it right away. If you bother to read it, you’ll see bindings to properties and types we haven’t defined yet.

There is nothing surprising in it if you know Xaml binding, no pretense that you’ll do other than paste this into your example if you choose to follow along, and no point in pasting more template xaml later.

# Revise the UI to show Products

We next create some new *Product* entities and display them.

**Open** the *MainWindowViewModel* class and **go to** the ***AddTestData*()** method

**Refactor** the category creation into a ***AddNewCategory()*** method and a ***CurrentCategory*** property.

private void AddNewCategory()

{

    CurrentCategory =

        new Category { CategoryName = "Sweet Things on " + DateTime.Now.ToString("o") };

Manager.AddEntity(CurrentCategory);

}

private Category CurrentCategory { get; set; }

**Write an *AddNewProduct*()** method

private Product AddNewProduct(string productName = "A new product")

{

    var newProduct = new Product

    {

        ProductName = productName,

        Category = CurrentCategory,

    };

    //Manager.AddEntity(newProduct); // harmless but unnecessary

    return newProduct;

}

A few noteworthy points:

* The caller can supply a new product name or the method will supply a default name
* Although the method does not add the new *Product* instance to the *Manager* explicitly, setting the *Category* to an entity already in the Manager (*CurrentCategory*) draws the new *Product* in.
* You could add it to the Manager explicitly if you wished; there’s no harm in adding it twice.

**Revise the *AddTestData*** method to use these new ViewModel members:

private void AddTestData()

{

    AddNewCategory();

    AddNewProduct("Chocolate Cream Pie");

    AddNewProduct("Baklava");

    Log("Saving new test data");

    Manager.SaveChanges();

}

**Go to *ShowTestData()*** where we’ll display the new products.

**Add a *Products* property** just below *ShowTestData*(); this the data bound collection of *Product*s to show.

public ObservableCollection<Product> Products { get; set; }

**Query for *Products***

This time instead of querying for *Categories*, we will query for *Products*. We also tell DevForce to “*Include*” … to bring along … the parent *Category* when it fetches *Product*s.

*Include* is a DevForce extension method; we require “*using IdeaBlade.EntityModel;*”

var productQuery = Manager.Products.Include("Category");

We’ve defined the query but we haven’t run it. No entities have been retrieved yet.

**Initialize *Products*** collection with the query results.

Products = new ObservableCollection<Product>(productQuery); // executes & adds to list

The constructor of *ObservableCollection<T>* sees the query as an IEnumerable<T>. It unintentionally executes the query (once) as a by-product of iterating over it.

**Go “CacheOnly”**

We’d like to prove that we really are bringing in the *Category* when we retrieve a *Product*. We don’t want the *Product*’s *Category* property, as in “*someProduct.Category*”, to do a lazy load.

One way to prevent a lazy load is to tell the *Manager* that it may no longer query the database by default. We **change** the *Manager*’s ***DefaultQueryStrategy*** to “**CacheOnly**”.

// DEMO ONLY – DO NOT DO THIS IN YOUR CODE

Manager.DefaultQueryStrategy = QueryStrategy.CacheOnly; // let's see only what's in cache

We can still query the database explicitly but no longer by default.

The final state of the *ShowTestData* method is:

private void ShowTestData()

{

    Log("Showing test data");

    var productQuery = Manager.Products.Include("Category");

    Products = new ObservableCollection<Product>(productQuery); // executes & adds to list

    // DEMO ONLY - DO NOT DO THIS IN YOUR CODE

    Manager.DefaultQueryStrategy = QueryStrategy.CacheOnly; // let's see only what's in cache

}

**Delete all logging of *Categories***, both the line in this method and the *LogCats* method.

**Review the *ProductTemplate*** in Window.xaml (optional)

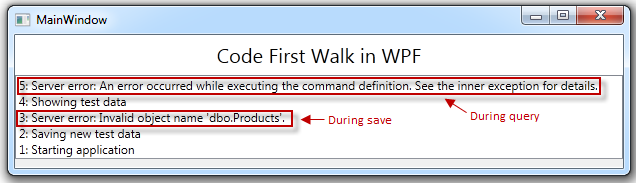
You’ll find a line in the product’s *DataTemplate* that binds to the *CategoryName*:

<TextBox Text="{Binding Category.CategoryName, Mode=TwoWay, ValidatesOnDataErrors=True}"

Notice how it navigates from *Product* (the implicit *DataContext*) to *Category* and from there to *CategoryName*. That works because (a) DevForce injects cache navigation into the *Product.Category* property and (b) the related *Category* instance is already in cache when the binding is exercised, thanks to the “*Include*” query.

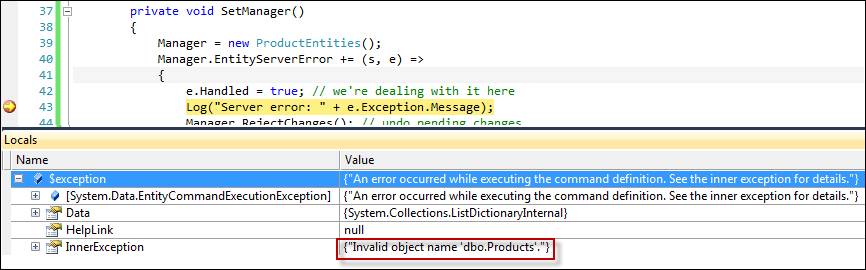
# F5 – Build and Run (fail)

The window reports failures during save and query. Good thing we’re logging failures!



The error message during the initial save tells us exactly what’s wrong: our model has a *Product* type but the database has no corresponding table. We’re trying to save *Product*s but our database doesn’t know about products.

Because we told DevForce that we’ve handled the exception, the application keeps going and fails again trying to query for products. The outer exception message isn’t useful but if we stop the application (Shift-F5), put a breakpoint on the *EntityServerError* handler, run again, and see what the inner exception has to say:

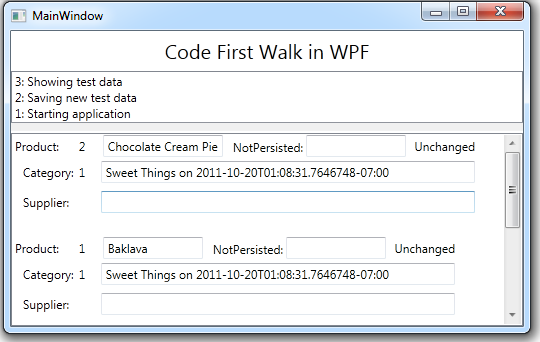


It’s the same problem. We’re trying to query for *Products* but which our database knows nothing about them.

The database is junk anyway so we can **drop the “ProductEntities” database**, using SQL Server Management Studio.

We’ll overcome this annoying necessity of dropping an out-of-date database in the next segment of this tutorial.

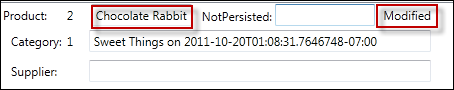
**Run again** (F5). Entity Framework creates the database this time and the application works.



The text value “Unchanged” comes from the *EntityState* property of every DevForce entity by way of its *EntityAspect* property as we see in the pertinent data binding line from the *ProductTemplate*.

<TextBlock Text="{Binding EntityAspect.EntityState}" Grid.Row="0" Grid.Column="5" .../>

It means that the *Product* hasn’t changed since we saved it. When we change “Chocolate Cream Pie” to “Chocolate Rabbit” and tab out, the displayed *EntityState* value becomes “Modified”.



DevForce infrastructure inside the *Product* entity is clearly at work:

* Change tracking detected the name change and altered the product’s *EntityState*
* Change notification caused the form to refresh its display of the *EntityState* value.

You don’t remember writing an *EntityAspect* or *EntityState* property. You don’t remember implementing change notification either. That’s because you didn’t. Review the source code for *Product*; they aren’t there.

The entity acquired these properties and capabilities when it was re-written as a consequence of DevForce AOP.

# Summary

In this part of the “Code First Walk” we

* added a new entity to the model … without using a designer or modifying the database.
* used eager loading (“Include”) to retrieve *Categories* with their *Products*.
* bound the UI to entities, leveraging DevForce AOP entity data binding support.
* displayed the state of an entity by binding to the DevForce *EntityAspect* property injected into AOP entities.

Explicit Entity Mapping

Our goal in this segment is to demonstrate explicit Code First mapping when the mapping between entity classes and database objects cannot be inferred from the class and property names. We’ll also introduce the Entity Framework DbContext class for the first time, both as a vehicle for imperative mapping with the Fluent API and as a means to re-create the database when the model changes.

We also add the new *Supplier* entity class and write a navigation property that returns a collection of related *Product* entities.

# Add a Supplier Entity

**Add reference** to *System.ComponentModel.DataAnnotations* if it’s not in your project already.

**Paste** the following at the bottom of the Model class file

[ProvideEntityAspect]

public class Supplier

{

   public Guid SupplierKey { get; set; }

   public string CompanyName { get; set; }

}

**Add the Suppliers EntityQuery** to the *ProductEntities* entity manager.

public EntityQuery<Supplier> Suppliers { get; set; }

We’ve done this before for our other entity classes to make it easier to query for them.

## Map and initialize the SupplierKey

**Return** to the *Supplier* class.

The *SupplierKey* property is the entity key; it has two unusual aspects:

1. its unconventional name
2. its *Guid* return type

The name, “SupplierKey”, does not conform to Entity Framework Code First [naming conventions](http://msdn.microsoft.com/en-us/library/hh161541%28v=VS.103%29.aspx). We must configure this critical key mapping explicitly.

We could use the [Code First Fluent API](http://msdn.microsoft.com/en-us/library/hh295844(v=VS.103).aspx) to configure the mapping imperatively (as we will later). Instead, for now, we’ll take a declarative approach and apply [mapping attributes](http://msdn.microsoft.com/en-us/library/gg197525%28v=VS.103%29.aspx). We’ll mark the *SupplierKey* property with the [*Key*] attribute

[Key] // Key for "Supplier" does not conform to conventions

public Guid SupplierKey { get; set; }

The *[Key]* attribute requires “*using System.ComponentModel.DataAnnotations;*”

The key’s *Guid* type presents a different challenge. Entity Framework maps integer keys to “Identity” columns by default; that means EF expects the database to generate the key value when new entities are inserted.

Entity Framework treats *Guid* key properties differently; it assumes that the application will assign their values. Many developers assign the key in the entity constructor; we will follow their lead by adding one here.

public Supplier()

{

    SupplierKey = Guid.NewGuid();

}

Entity Framework can generate *Guid* keys if you wish. Learn more in the [topic on *Guid* keys](http://drc.ideablade.com/xwiki/bin/view/Documentation/code%2Dfirst%2Dentity%2Dclasses#HGuidkeys).

## Set the Table Mapping

If you happened to study the database that Entity Framework generated, you perhaps noticed that the tables corresponding to the entities have plural names: “Categories” and “Products”. That is EF’s Code First convention.

We might prefer that the table names be singular like their corresponding entity class names. We can tell Code First to expect a singular table name with the *System.ComponentModel.DataAnnotations.****Table*** mapping attribute, decorating the class as follows:

[Table("Supplier")]

public class Supplier { ... }

## Associate Suppliers and Products

Suppliers will have Products and each Product will have a parent Supplier.

Products have multiple Suppliers in the real world. We could model that with a ProductSupplier association table and perhaps a many-to-many relationship between the two classes. That’s an exercise for the future. At the moment, we’ll design for a one-to-many relationship between Supplier and its Products.

**Add *SupplierKey* and *Supplier* to the *Product* class**

**Go** to the Product class and add the following *SupplierKey* and *Supplier* properties which are just like *CategoryId* and *Category*.

public Guid SupplierKey { get; set; } // Foreign key for "Supplier"

public Supplier Supplier { get; set; }

DevForce and Entity Framework can tell that *SupplierKey* holds the foreign key value uniting *Product* and *Supplier* because it has the same name as the *Supplier*’s key property. If you wanted to be explicit about the role of the *SupplierKey* property (as you’d have to be if you gave it a different name), you could decorate it with the *ForeignKey* attribute:

[ForeignKey("Supplier")] // Specifies the corresponding navigation property

public Guid SupplierKey { get; set; } // Foreign key for "Supplier"

public Supplier Supplier { get; set; }

**Add *Products* property to the *Supplier* class**

**Go** back to Supplier and add the following Products property.

public RelatedEntityList<Product> Products { get { return null; } }

The return type must be *RelatedEntityList<T>*, a derivative of *ICollection<T>* as required by EF Code First. We’ve implemented the getter but not the setter because we don’t want anyone to set the *Products* collection. Add and remove products, yes, but not change the collection itself. That’s a DevForce job.

The *Supplier* class at this point is as follows:

[ProvideEntityAspect]

[Table("Supplier")]

public class Supplier

{

public Supplier()

{

     SupplierKey = Guid.NewGuid();

}

    [Key] // Key for "Supplier" does not conform to conventions

    public Guid SupplierKey { get; set; }

    public string CompanyName { get; set; }

    public RelatedEntityList<Product> Products { get { return null; } }

}

# Add an unmapped property

Your business logic may require entity properties that do retrieve or store data in the database. Calculation fields are typical but you might have a settable property as well. Let’s add an example to the *Product* class.

// Example of non-persisted property

[NotMapped]

public string NotPersisted

{

    get { return \_notPersisted; }

    set { \_notPersisted = value; }

}

private string \_notPersisted = "I'm not persisted to the database.";

# Add a custom DbContext Class

Those of you who are familiar with Entity Framework Code First may be surprised that we have not even mentioned EF’s *DbContext* class. EF Code First requires a *DbContext*.

DevForce has been using a *DbContext* all along. It’s been using its own *DbContext*. DevForce will use your *DbContext* instead … if you write one.

Most developers write a *DbContext* in order to configure entity-database mapping imperatively using the [Code First Fluent API](http://msdn.microsoft.com/en-us/library/hh295844(v=VS.103).aspx). We’ve done quite well with just the [mapping attributes](http://msdn.microsoft.com/en-us/library/gg197525%28v=VS.103%29.aspx). But some people don’t like attributes and some mappings (e.g., certain forms of inheritance) can only be defined through the Fluent API.

Another reason to write a *DbContext* is to control how and when the Entity Framework creates and initializes the database … or to stop it from ever creating a database.

This is why we will write a DbContext. We’re tired of running our application, watching it crash because the model changed, deleting the database, and running again. In these early stages of development, we are happy to let Entity Framework detect the mismatch and re-create the database for us automatically.

**Add reference to *System.Data.Entity*** if not referenced already. *DbContext* depends upon this assembly.

**Add | New Item** (Ctrl-Shift-A) | **Class**

Call it “*ProductDbContext*”

**Inherit from *DbContext***.Leave the class *internal*; no one will call it except DevForce.

class ProductDbContext : DbContext

**Add** “*using System.Data.Entity;*”

**Add a constructor** **with a string parameter** as follows:

public ProductDbContext(string connection) : base(connection)

{

    // Do not use in production; for early development only

    Database.SetInitializer(

        new DropCreateDatabaseIfModelChanges<ProductDbContext>());

}

Notice that the constructor takes a string. When DevForce creates an instance of your *DbContext*, it supplies either a database connection string or the name of a connection string in the assembly’s configuration file.

Always add a constructor that takes a string parameter

Notice the *DropCreateDatabaseIfModelChanges<T>* object passed in the static [*Database.SetInitializer* method](http://msdn.microsoft.com/en-us/library/gg679461%28v=VS.103%29.aspx) call. That’s an [initialization strategy](http://drc.ideablade.com/xwiki/bin/view/Documentation/code-first-dbcontext#HInitializethedatabase). We’re asking Entity Framework to create the database if it doesn’t exist or (b) drop and recreate it according to the revised model if the model has changed. With a little more work we could write a strategy that seeds the database with test data as well.

Pass *null* into the *SetInitializer* to prevent EF from creating the database … ever. That’s the correct call for a production release or when accessing a database with irreplaceable data or schema.

## Configure to ignore EntityAspect

All DevForce AOP entities have an [*EntityAspect*](http://drc.ideablade.com/xwiki/bin/view/Documentation/code-first-entityaspect) property through which you gain access to your entity’s internal entity capabilities. We haven’t written such a property in any of our entity source code, but it’s there after DevForce [rewrites each entity class](http://drc.ideablade.com/xwiki/bin/view/Documentation/code-first-entity-classes) with its DevForce infrastructure.

Entity Framework assumes that the *EntityAspect* object returned by the property is an *entity* type … which it is **not**. EF model validation will fail unless someone tells EF to ignore the *EntityAspect* type and all properties that return that type.

The DevForce default *DbContext* does that for us. But when we write our own *DbContext* - as we are doing now -, ***we*** have to tell EF to ignore it, using the Entity Framework [Code First Fluent API](http://msdn.microsoft.com/en-us/library/hh295844(v=VS.103).aspx) to which we gain access by overriding *DbContext*’s *OnModelCreating* method.

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

    modelBuilder.Ignore<EntityAspect>();

}

**Add** “*using IdeaBlade.EntityModel;*”

## Add DbSet Properties for roots

We have to tell Entity Framework which entity types are in our model. We do that by specifying a DbSet<T> property where “T” is an entity type. We only need one such property for this model:

 public DbSet<Supplier> Suppliers { get; set; } // root entity for model discovery

*Supplier* is a “root entity”, meaning that one can walk an “object graph” from this entity to other entities by following navigation properties recursively. ***Supplier****.Products* returns***Product***entities*. Product.Category* returns a***Category***entity*.* Those are the three entities in our model and *Supplier* is a root to all of them.

Developers with previous Entity Framework Code First experience tend at first to write a *DbSet* property for all entity types just as we add *EntityQuery<T>* properties to the *ProductDbEntities* entity manager. They do so to make it easier to access model types from the *DbContext*. But a DevForce developer rarely (if ever) uses this *DbContext* so that kind of effort is usually a waste of time.

## Add [*DataSourceKeyName*] attribute

DevForce no longer uses the name of your *EntityManager*, “ProductEntities”, as the name for finding the database connection string or for naming the database it creates. Instead DevForce now uses the name of your *DbContext*, “ProductDbContext”.

If you could run the application now (which you can’t quite do), you’d see Entity Framework create a database named “ProductDbContext”.

“ProductDbContext” is not a good name for the database, even in a demo. “ProductEntities” wasn’t a great name either. Neither of them is a wonderful name for the connection string in a configuration file.

Rather than be at the mercy of potentially changing class names, it is best to specify the connection string name explicitly. In DevForce, this name is called the *DataSourceKeyName* and you specify it by decorating the *DbContext* class with the *DataSourceKeyNameAttribute*. For our example, we’ll name it “CodeFirstWalk”.

[DataSourceKeyName("CodeFirstWalk")]

class ProductDbContext : DbContext {...}

We could have added this attribute to the entity manager class, *ProductEntities*, and had the same effect. We saved the attribute for our *DbContext* because the name (or attribute) of the *DbContext* always trumps the name (or attribute) of the *EntityManager*.

## Re-configure *Supplier* mapping with fluent API (optional)

The author of this sample does not want to see database specifics in the entity model classes. He dislikes the [*Table*] and [*Column*] attributes in particular. If you are not similarly troubled by it, you can skip this section or read-and-ignore it.

The Entity Framework naming conventions translate a singular entity class name to a plural table name. We prefer singular table names. Recall that we applied [*Table*] to the *Supplier* class so that our model would conform to our preferred table naming style:

[Table("Supplier")]

public class Supplier

We can achieve that goal through the fluent interface.

**Delete** **that *Table* attribute** from the *Supplier* class definition in the Model file.

**Add** **imperative Table configuration** to *ProductDbContext’s* *OnModelCreating* method.

modelBuilder.Entity<Supplier>().ToTable("Supplier");

It seems that other people don't like the Entity Framework plural table name convention either. This is the one convention you can change in the current EF Code First release. Let's remove that convention so that all of our table names are singular.

**Replace** the "*ToTable()"* line we just added with

modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

The revised *ProductDbContext* follows:

[DataSourceKeyName("CodeFirstWalk")]

class ProductDbContext : DbContext

{

    public ProductDbContext(string connection)

        : base(connection)

    {

        // Do not use in production; for early development only

        Database.SetInitializer(

            new DropCreateDatabaseIfModelChanges<ProductDbContext>());

    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)

    {

        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();

modelBuilder.Ignore<EntityAspect>();

    }

public DbSet<Supplier> Suppliers { get; set; } // root entity for model discovery

}

## Clean and Rebuild

**Build the model (Ctrl-Shift-B)**. We recommend that you **always build the model after making a change** **to the model** because it is usually easier to catch and repair a modeling error immediately rather than discover and figure it out later.

You may notice in the project inventory that the old *ProductEntities.ibmmx* metadata file associated with the *ProductEntities* *EntityManager* has disappeared and been replaced by a new metadata *CodeFirstWalk.ibmmx*, named to match the *DataSourceKeyName*.

# Update the UI for Supplier

We’ll only use one Supplier in our demo system so our *MainWindowViewModel* revisions will be a little different from the way we’ve done them before. To the top of the *AddTestData*() method **add** …

private void AddTestData()

{

    GetOrAddSupplier();

// ...

}

**Add** the ***GetOrAddSupplier* method** and ***CurrentSupplier* property**

/// <summary> Get first supplier, if exists, or make one</summary>

private void GetOrAddSupplier()

{

    CurrentSupplier = Manager.Suppliers.FirstOrDefault();

    if (null != CurrentSupplier) return;

    CurrentSupplier = new Supplier { CompanyName = "All Fine Foods" };

    Manager.AddEntity(CurrentSupplier);

}

private Supplier CurrentSupplier { get; set; }

If there’s a *Supplier* in the database, that’s our current supplier; if not, we make a new one and add it to the *Manager* for subsequent save with the new *Category* and new *Products* that we’re already making in *AddTestData*.

**Assign the new *Product*’s *Supplier*** in the *AddNewProduct* method by adding another initialization. The revised method is as follows:

private Product AddNewProduct(string productName = "A new product")

{

    var newProduct = new Product

    {

        ProductName = productName,

        Category = CurrentCategory,

        Supplier = CurrentSupplier,

    };

    //Manager.AddEntity(newProduct); // harmless but unnecessary

    return newProduct;

}

We don’t have to add the new *Product* to the Manager because both the *CurrentCategory* and the *CurrentSupplier* are in the *Manager*’s cache already and will pull the new *Product* into the cache upon assignment.

## Prove that *Supplier.Products* works

The *MainWindow.xaml* is already data bound to the *Product*’s supplier (we got ahead of ourselves). That binding shows how to navigate from *Product* to *Supplier*. We should demonstrate the *Supplier.Products* navigation property works in the opposite direction by querying the count of a *Supplier* entity’s products and logging the count in the view.

**Go** to the *ShowTestData* method

**Add** to the bottom of the *ShowTestData* method:

LogCurrentSupplierProductsCount();

Implement that method:

 private void LogCurrentSupplierProductsCount()

 {

     Log(

         String.Format("Supplier '{0}' has {1} products.",

         CurrentSupplier.CompanyName, CurrentSupplier.Products.Count));

 }

The *CurrentSupplier.Products* navigation property returns a *RelatedEntityList<Product>* whose *Count* method causes DevForce to query for related products.

You may recall that we prevented further querying of the database when we set the *Manager*’s *DefaultQueryStrategy* to *CacheOnly*.

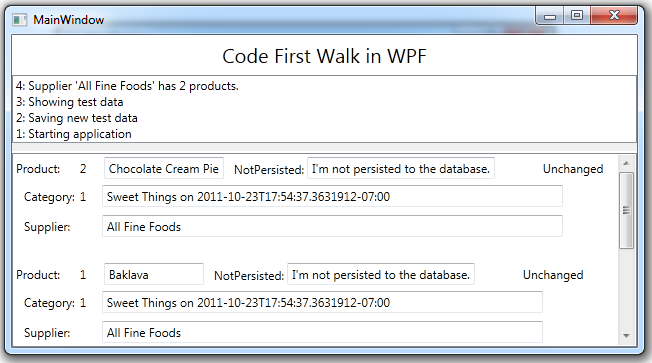
// DEMO ONLY – DO NOT DO THIS IN YOUR CODE

Manager.DefaultQueryStrategy = QueryStrategy.CacheOnly; // let's see only what's in cache

Fortunately, *ShowTestData* already fetches every product in the database; by the time we look for them, the *CurrentSupplier*’s products will be in the local cache.

# Run in Debug [F5]

It should display a window such as this one:



Notice

* The log presents the count of products from the *CurrentSupplier.Product* navigation property.
* The *NotPersisted* value appears on screen but is not a column the database.
* The *Product.Supplier* property returns the product’s supplier

# Summary

In this part of the “Code First Walk” we

* added the *Supplier* entity whose properties defied EF’s naming conventions.
* consequently, we mapped Supplier explicitly with attributes and the fluent API.
* added a collection navigation property (*Supplier.Products*) that returns a *RelatedEntityList<T>*
* added a custom *DbContext*, having relied on the DevForce default *DbContext* until now.
* applied the DataSourceKeyName attribute to determine the connection string name and the name of the generated database
* now re-create the database whenever the model changes – a technique suitable only during early development.

Validation & Property Interception

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](http://drc.ideablade.com/xwiki/bin/view/Documentation/code-first-entity-classes).

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.

# Add Validation and UI Hints

If you look at properties of entities [generated by DevForce](http://drc.ideablade.com/xwiki/bin/view/Documentation/model%2Dgenerate) 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.

You can use *System.ComponentModel.DataAnnotations* validation attributes instead if you prefer but we recommend the [DevForce validation](http://drc.ideablade.com/xwiki/bin/view/Documentation/validate) attributes for their superior capabilities.

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

using System.ComponentModel;

using IdeaBlade.Validation;

**Go to the *Supplier* class**

**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.

**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](http://stats.sfgate.com/fb/teamstats.asp?teamno=13&type=scheduleshttp://drc.ideablade.com/xwiki/bin/view/Documentation/property%2Dinterceptors) 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 propety interceptor attribute such as *AfterGet*.

**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](http://drc.ideablade.com/xwiki/bin/view/Documentation/model%2Dmember%2Dvisibility).

# Add, Delete, and Save

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

## Add AddProductAction

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

**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.

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

SelectedProduct = AddNewProduct("Chocolate Cream Pie");

Now it’s time to define *SelectedProduct*.

**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.

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

public class MainWindowViewModel : INotifyPropertyChanged

Requires “*using System.ComponentModel;*”

**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*](http://drc.ideablade.com/xwiki/bin/view/Documentation/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.

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

**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](http://drc.ideablade.com/xwiki/bin/view/Documentation/code-first-entityaspect).

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.

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

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

**Build** the project.

## Add the Buttons

**Open** *MainWindowViewModel.xaml*

**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>

**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.

**Right-Click** | **View Code**

**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.

**Add the \_viewMode**l private variable and **assign it in the constructor**.

private readonly MainWindowViewModel \_viewModel;

public MainWindow()

{

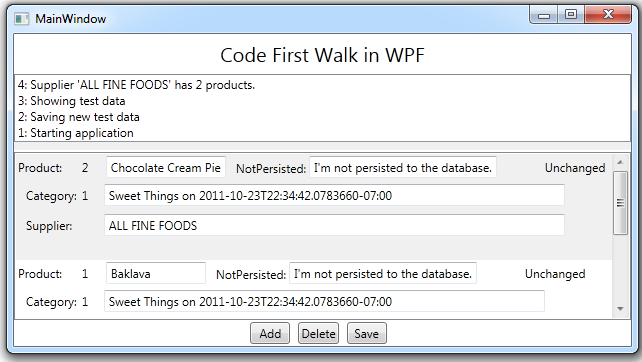
    InitializeComponent();

    DataContext = \_viewModel = new MainWindowViewModel();

}

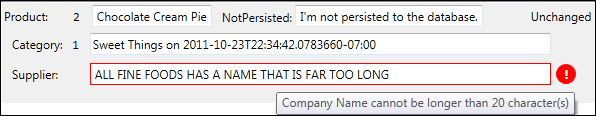
# Try the UI

**Build and run [F5]** … you should see

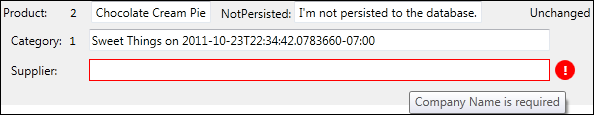


Notice that the *Supplier.CompanyName* property interceptor is capitalizing “*ALL FINE FOODS*” (which is actually “*All Fine Foods*” in the data), both in the log at the top and for each *Product* entity displayed.

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

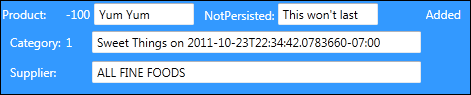


… or if you clear the name:



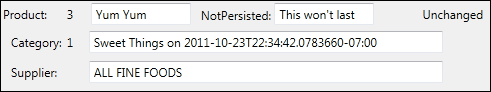
**Add a new Product** and scroll to the bottom.

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



Notice the temporary *ProductId* (-100) and the *EntityState* of “Added”

**Press the** **Save** button



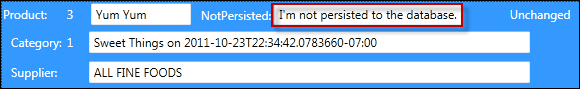
The save succeeds. The *ProductId* receives its permanent value and the *EntityState* becomes “Unchanged”. The NotPersisted value remains as we last modified it; this product entity is the same physical object in cache. That won’t change until we evict it or terminate the application.

**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

**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.

**Delete** product #3 and **Save**

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

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

Separate Model Project

Many of us prefer to keep our entity models in their own projects, separate from other applications projects … especially from the UI projects.

Of course we could have begun development of the “Code First Walk” application with an independent model project. But we didn’t because we were keeping it as simple as possible … as you might do.

Fortunately, it’s easy to move the model to its own project … as we do here.

## Create the Model Project

**Close** all Visual Studio editor windows (Alt-W, L)

**Add | New** **Project** | **Class Library**

Name it “***CodeFirstWalk.Model***”

**Delete** the template-generated “Class1”

**Add Entity Framework Code First Library reference**

* Tools | Library Package Manager | Package Manager Console Window
* Enter at the prompt:   
   Install-Package EntityFramework -ProjectName CodeFirstWalk.Model

The console should display

PM> Install-Package EntityFramework -ProjectName CodeFirstWalk.Model

'EntityFramework 4.2.0.0' already installed.

Successfully added 'EntityFramework 4.2.0.0' to CodeFirstWalk.Model.

**Add the other** **references**

CodeFirstWalk also requires references to these libraries:

* *IdeaBlade.Aop*
* *IdeaBlade.Core*
* *IdeaBlade.EntityModel*
* *IdeaBlade.Linq*
* *IdeaBlade.Validation*
* *PostSharp*
* *System.ComponentModel.DataAnnotations*
* *System.Data.Entity*

The easy way to add them is to let the “*DevForce CodeFirst File*” item template do it for you while it adds the marker file.

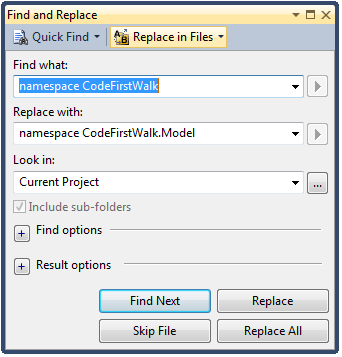
**Add | New** **Item** (or [ctrl+shift+A]) | **DevForce 2010** | **DevForce CodeFirst File** | “**DevForce.cf**”

**Copy** the **Model** and **ProductDbContext files** in the *CodeFirstWalk* application projectinto the **Model** project.

**Open the Model file**

**Change all Model file namespaces** to *CodeFirstWalk.Model* by using (Ctrl-H) “Global Replace” (for the ***model project only***)

Replace all “*namespace CodeFirstWalk*” with “*namespace CodeFirstWalk.Model*”.



**Build the Model project only**. It should compile cleanly.

## Adjust the Application Project

**Go** to the **CodeFirstWalk application project**

**Delete the following Model-oriented files**

* *the “.ibmmx” file*
* *DevForce.cf*
* *Model.cs*
* *Packages.config*
* *ProductDbContext.cs*

**Add project reference** to **CodeFirstWalk.Model**

**Build** the solution

It will fail, complaining that it can’t find the model classes. The error messages tell you where you need to add “*using CodeFirstWalk.Model;*”. In our example, you only need to …

**Add “*using CodeFirstWalk.Model;*”** to *MainWindowViewModel*.

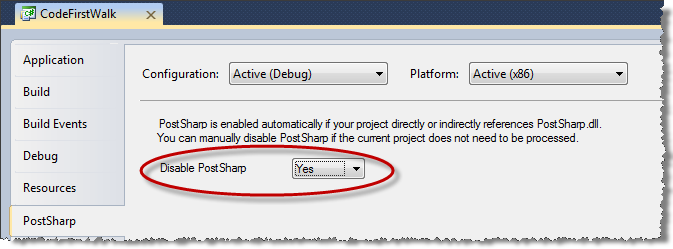
## Disable PostSharp in the application project

At build, PostSharp investigates the classes of every project in which it is referenced, directly or indirectly. That's a waste of time. We can disable PostSharp analysis by flipping a switch.

**Open** the application **project** **property editor** (Alt-Enter)

**Open the PostSharp tab** near the bottom

**Set "Disable PostSharp"** to "Yes"

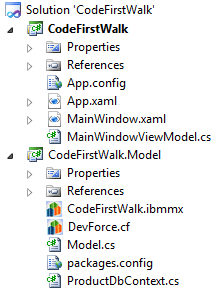


In principle, you shouldn’t need a reference to **PostSharp** in the application project because you aren’t compiling any AOP classes in there; you are merely consuming AOP objects defined in the Model assembly. In practice, **you must reference PostSharp.dll in WPF projects**. This is *only* a problem for WPF projects.

## Build and run [F5]

The application should run as it did before.

At the conclusion of Model separation, the solution structure should look like this:



## Optional Steps

**Consider setting** “**Specific Version=False**” for all project references

The "Specific Version = True” for most of the added references (no matter how you added them). DevForce releases new versions of its libraries every 6 to 10 weeks. If you upgrade frequently when "Specific Version = True", you’ll notice that your project with IdeaBlade references don’t compile until you upgrade them individually … which is painful. We like to set this flag to “False” during development.

* Select all the references
* Open the Properties Window
* Find and change the “Specific Version” setting to “False”

**Remove** **references** you won’t need from the **model project** (optional)

* *Microsoft.CSharp*
* *System.Data.DataSetExtensions*
* *System.Xml*
* *System.Xml.Linq*

**Remove references** you won’t need from the **application project** (optional)

* EntityFramework
* IdeaBlade.AOP
* System.Data.Entity
* System.ComponentModel.DataAnnotations

The *PostSharp* library is still required in the application project although it is only used by the Model project. This is a peculiarity of WPF projects as explained above.

**Add an App.config** file if necessary (optional)

If Entity Framework is (re)creating the development database for you (as we’ve asked it to do in this CodeFirstWalk sample) then you don’t need to add an *App.config*.

If you require design time access to a specific database - as you might if you hadn’t installed SQL Server Express on your development machine -, you may need to create an *App.config* file in the Model project.

* Add | New Item | Application Configuration File
* **Copy only the *<connectionStrings>* tag (and its contents)** from the *App.config* in your application project (CodeFirstWalk).

See the “[Metadata Generation](http://drc.ideablade.com/xwiki/bin/view/Documentation/code%2Dfirst%2Dgenerate%2Dmetadata)” and “[Advanced database connections](http://drc.ideablade.com/xwiki/bin/view/Documentation/advanced%2Ddatabase%2Dconnections)” topics for a discussion of the issues.

Testing the Model (introduction)

In this segment, we introduce automated testing a DevForce Code model. The lessons apply to all DevForce entity models, whether built “Code First” or “Database First” with an EDM.

We feel automated testing should be an essential part of any development regime. We recommend testing as you go rather than bolting it on at the end. In that respect, we’ve waited too long. The future evolution of the CodeFirstWalk sample will include tests and often rely on tests to explain each step.

# Establish an MS-Test project for Model testing

**Add** | **New Project** | **Test Project**

Call it “CodeFirstWalk.Model.Test”

**Add a reference to the model project**,*CodeFirstWalk.Model*.

**Add DevForce references**:

* *IdeaBlade.Core*
* *IdeaBlade.EntityModel*
* *IdeaBlade.EntityModel.Server*
* *IdeaBlade.Linq*
* *IdeaBlade.Validation*

**Set “Copy Local=True”** in the Properties window for all of them.

**Set “Specific Version=False”** for all of them

**Disable PostSharp analysis** for the test project

* Open the project properties (Alt-Enter).
* Open the “PostSharp” tab at the bottom.
* Set “Disable PostSharp” to “Yes”.

**Delete the generated constructor and *TestContext*** stuff because not used

**Delete the “Additional test attributes” region** when you understand the comments.

**Rename *TestMethod1* -> *DoNothing***

**Change its body** to *Assert.Fail(“Fail on purpose”);*

**Run all tests [Ctrl-R, A]** … and **see the FAILURE** report in the Test Results window

These Visual Studio keyboard shortcuts are timesavers:

[Ctrl-R, A] – Run all tests.

[Ctrl-R, T] – Run the test my mouse is in; run the tests in the test class my mouse is in.

[Ctrl-R, Ctrl-A] – Debug all test(s)

[Ctrl-R, Ctrl-T] – Debug test(s) selected as in [Ctrl-R, T]

**Delete** *DoNothing*

# First offline EntityManager Test

We will do all testing for a long while with offline *EntityManager*s.

Most testing ***can*** be done with offline *EntityManager*s.

Most testing ***should*** be done with offline *EntityManager*s. Most tests don’t need to involve a server or the Entity Framework or a database. These are downstream potential points of test failure that have nothing to do with the subject of your test. We will need to involve them – to use a connected *EntityManager* – when we write end-to-end integration tests. But here we are only interested in the interaction of application code with the model.

The *EntityManager* is an essential dependency when testing most entity model functionality. But we can limit the risk of downstream failures by keeping the *EntityManager* offline and working out of its cache of entities … which we control completely during test setup.

**Rename the class file** to *When\_exploring\_offline\_ProductEntityManager*

Visual Studio should offer the opportunity to rename the class as well; say “Yes”. If it doesn’t …

**Rename the class** to *When\_exploring\_offline\_ProductEntityManager*

**Add *public* *Manager* property** returning the type of the model’s *EntityManager* which is *ProductEntities* in our example. The property is *public* because we’ll probably relocate this material for reuse in other test classes down the road.

**Add *TestInitialize*** method at the top, method runs before every test in the test class

**Assign *Manager* inside *TestInitialize*** with a new instance of *ProductEntities* that is disconnected.

The current state of the code should be this:

public ProductEntities Manager { get; set; }

[TestInitialize]

public void TestInitialize()

{

    Manager = new ProductEntities(shouldConnect: false);

}

# Add “offline-optional” constructor to ProductEntities

The base EntityManager class has numerous optional constructors, many with optional parameters. We haven’t needed any of them to date. Now we need to implement one of them (or a part of it).

**Go to the Model** file and find *ProductEntities* and add a constructor with optional *shouldConnect* parameter:

public ProductEntities(bool shouldConnect = true) : base(shouldConnect) { }

# Make sure test manager never fetches or saves

We know the manager is offline when we create it. We don’t want a test to get sneaky and connect it. Let’s fail any test that tries to query from or save to the database.

**Return** to the test project which can now compile.

**Add** using IdeaBlade.EntityModel;

**Add *Fetching* and *Saving* event handlers** that fail the test when called:

 [TestInitialize]

 public void TestInitialize()

 {

     Manager = new ProductEntities(shouldConnect: false);

     Manager.Fetching += (s, e) => Assert.Fail("Manager tried to fetch entities.");

     Manager.Saving += (s, e) => Assert.Fail("Manager tried to save.");

 }

**Add test for save blocking**

[TestMethod]

[ExpectedException(typeof(EntityManagerSaveException))]

public void Fails\_if\_you\_try\_to\_save()

{

    Manager.Connect(); // no-no

    Manager.SaveChanges(); // should catch and fail

}

Notice how the *ExpectedException* MS-Test attribute asserts that the body of the test will throw an exception of a particular type.

**Run the test** by placing the mouse somewhere in the test and pressing [Ctrl-R, T].

# Write your first EntityManager test

**Write and execute the test** *Can\_query\_added\_supplier* as follows:

 [TestMethod]

 public void Can\_query\_added\_supplier()

 {

     var supplier = new Supplier {CompanyName = "Test Supplier"};

     Manager.AddEntity(supplier);

     Assert.AreSame(supplier, Manager.Suppliers.FirstOrDefault());

 }

Congratulations! Your first real DevForce test demonstrates that you can add a new entity to cache and get it back as if you were querying from the database.

Of course this is not a test of the Manager’s actual ability to query the database. We can test that elsewhere. Our immediate goal is to test code that “thinks” it is querying for entities and is doing something useful with them afterward. We’re testing (a) how the code responds to querying outcomes and (b) whatever it is that the code is doing with the entity.

# Test entity business logic

Let’s try testing the small bits of business logic that we added to the *Supplier* entity in our model:

1. The *CompanyName* property returns an uppercase value, no matter what its internal value.
2. *CompanyName* is required and has a maximum length of 20 characters

**Write and call test of the uppercasing property interceptor**

[TestMethod]

public void Then\_Supplier\_CompanyName\_is\_uppercased\_on\_get()

{

    const string companyName = "lower case name";

    var detachedSupplier = new Supplier {CompanyName = companyName};

    Assert.AreEqual(companyName.ToUpper(), detachedSupplier.CompanyName);

}

Property interceptors operate whether or not the entity is attached to an *EntityManager* so we didn’t even bother adding the supplier to the *Manager*.

However, [property validation](http://drc.ideablade.com/xwiki/bin/view/Documentation/validate) **is disabled** until the entity becomes attached, as we see in this test:

[TestMethod]

public void Then\_detached\_Supplier\_CompanyName\_is\_not\_required()

{

    var supplier = new Supplier { CompanyName = "Supplier" };

    // Manager.AddEntity(supplier); // comment out to keep supplier detached

    supplier.CompanyName = string.Empty;

    var msg = GetFirstValidationErrorMessage(supplier);

    Assert.IsNull(msg, "Did not expect the validation error: " + msg);

}

The validation test requires the following *GetFirstValidationErrorMessage* helper method:

public string GetFirstValidationErrorMessage(IEntity entity)

{

    var firstError = entity.EntityAspect.ValidationErrors.FirstOrDefault();

    return (null == firstError) ? null : firstError.Message;

}

Notice how the helper method acquires instance validation error information by way of the [*EntityAspect* property](http://drc.ideablade.com/xwiki/bin/view/Documentation/code%2Dfirst%2Dentityaspect) that we added to our model’s *BaseEntity* class.

**Run the test**. If the “Required” rule were functioning, we would detect a validation error when we set the *CompanyName* to the empty string. The *Assert* confirms that the entity is unaware of the error.

You can validate a detached entity explicitly if you wish. The property validation **on set** is disabled for detached entities but the rule is still there to be enforced … as shown in the lesson appendix

After attaching a Supplier to the Manager, the required and length validation rules are applied when the property is set as these tests show.

[TestMethod]

public void Then\_attached\_Supplier\_errs\_when\_CompanyName\_set\_StringEmpty()

{

    var supplier = new Supplier {CompanyName = "Supplier"};

    Manager.AddEntity(supplier);

    supplier.CompanyName = string.Empty;

    var msg = GetFirstValidationErrorMessage(supplier);

    Assert.IsNotNull(msg,

                  "Expected a validation error when CompanyName set to empty string.");

    Assert.IsTrue(msg.Contains("required"),

                  "Error message did not contain 'required'; was " + msg);

}

[TestMethod]

public void Then\_attached\_Supplier\_errs\_when\_CompanyName\_set\_null()

{

    var supplier = new Supplier {CompanyName = "Supplier"};

    Manager.AddEntity(supplier);

    supplier.CompanyName = null;

    var msg = GetFirstValidationErrorMessage(supplier);

    Assert.IsNotNull(msg,

                  "Expected a validation error when CompanyName set to null.");

    Assert.IsTrue(msg.Contains("required"),

                  "Error message did not contain 'required'; was " + msg);

}

[TestMethod]

public void Then\_attached\_Supplier\_errs\_when\_CompanyName\_set\_too\_long()

{

    var supplier = new Supplier { CompanyName = "Supplier" };

    Manager.AddEntity(supplier);

    supplier.CompanyName = "Supplier name with more than 20 characters";

    var msg = GetFirstValidationErrorMessage(supplier);

    Assert.IsNotNull(msg,

           "Expected a validation error when CompanyName > 20 chars.");

    Assert.IsTrue(msg.Contains("20"), // looking for "cannot be longer than 20 character(s)

           "Error message did not contain '20'; msg = " + msg);

}

# Prepare test entities with a “Data Mother”

We often write a battery of tests that work with a bunch of related test entities. We will want to pretend that these test entities either are in the database already or were freshly queried. We don’t need – or want - an actual database of test entities to fulfill our testing intentions.

We could code these test entities within the body of a test … as we’ve been doing so far. That’s often a good practice because it keeps our tests from relying on far away code that we can’t see.

But writing the same test entity setups over and over is tedious setup and can distract attention from the test purpose. It’s usually better to delegate this kind of setup to a helper method … or such a method in a helper class which is sometimes known as a “Data Mother”.

**Write this public *PopulateTestManager* method**

// Add test entities to the Manager

public void PopulateTestManager(EntityManager manager)

{

    testSupplier = new Supplier { CompanyName = "Test Supplier" };

    manager.AttachEntity(testSupplier); // attached unchanged as if from query

    testCategory = new Category {CategoryId = 123, CategoryName = "Test Category"};

    manager.AttachEntity(testCategory); // attached unchanged as if from query

}

public Supplier testSupplier;

public Category testCategory;

The treatment of *testSupplier* and *testCategory* is almost the same as in our earlier tests. The important difference is the call to *AttachEntity()* instead of *AddEntity()*. ***AttachEntity()*** puts the entity in cache in an “*Unchanged*” state, as if it had been queried from the database.

**Write a test** to confirm that the *testCategory* is configured as we expect. Put it above *PopulateTestManager* then run it.

[TestMethod]

public void Then\_testCategory\_is\_valid()

{

    PopulateTestManager(Manager);

    var cat = Manager.Categories.First();

    Assert.AreSame(testCategory, cat,

        "query didn't return the testCategory");

    var state = cat.EntityAspect.EntityState;

    Assert.AreEqual(state, EntityState.Unchanged,

        "unexpected category EntityState, " + state);

    Assert.IsTrue(cat.CategoryId > 0,

        "CategoryId appears to be a temporary id, " + testCategory.CategoryId);

}

**Add some test products** to *PopulateTestManager*. After revision, it looks like this:

// Add test entities to the Manager

public void PopulateTestManager(EntityManager manager)

{

    testSupplier = new Supplier { CompanyName = "Test Supplier" };

    manager.AttachEntity(testSupplier); // attached unchanged as if from query

    testCategory = new Category { CategoryId = 123, CategoryName = "Test Category" };

    manager.AttachEntity(testCategory); // attached unchanged as if from query

    testProduct1 = new Product

                       {

                           ProductName = "Product 1",

                           Category = testCategory,

                           Supplier = testSupplier,

                           ProductId = 1, // must set after navigation property!

                       };

    testProduct2 = new Product

                       {

                           ProductName = "Product 2",

                           Category = testCategory,

                           Supplier = testSupplier,

                           ProductId = 2, // must set after navigation property!

                       };

    // Setting a Product's navigation property pulls it into cache as an ADDED entity

    Manager.AcceptChanges(); // makes everything in cache appear as if queried.

}

public Supplier testSupplier;

public Category testCategory;

public Product testProduct1;

public Product testProduct2;

The additional definitions of the test products are straightforward. There are three notable DevForce effects addressed in this implementation:

1. DevForce pulls the products into the Manager’s cache as entities in “added” state *immediately* after the code first sets a navigation property with an entity in cache (*Category*).
2. DevForce sets the product keys with temporary key values *immediately* upon entering the cache; therefore, if we want our own test key values, we have to re-set the product keys ***after*** the product enters the cache.
3. Because the products are in the “added” state, we must flip them to “*Unchanged*” state when we’re done adding them … which we do by accepting all changes pending in the Manager.

**Write a test** to confirm that the *testProduct1* is configured as we expect. Put it above *PopulateTestManager* then run it.

[TestMethod]

public void Then\_testProduct\_is\_valid()

{

    PopulateTestManager(Manager);

    var state = testProduct1.EntityAspect.EntityState;

    Assert.AreEqual(state, EntityState.Unchanged,

        "unexpected testPoduct EntityState, " + state);

    Assert.IsTrue(testProduct1.ProductId > 0,

        "ProductId appears to be a temporary id, " + testProduct1.ProductId);

    Assert.AreSame(testCategory, testProduct1.Category, "unexpected testProduct Category");

    Assert.AreSame(testSupplier, testProduct1.Supplier, "unexpected testProduct Supplier");

}

In this “Prepare Test Entities” segment, the only testing we did is of our ability to write test entities. Yes, it’s a good idea to validate the assumptions you have about your test data. And in the process, we’ve provided clues for testing facts about individual entities and their relationships to each other.

But until we use these entities to test our application, we’ve wasted time. Treat this work as a promise of future utility, a promise we’ll redeem in an upcoming segment.

# Lesson Appendix: validate on-demand

We showed earlier that property validation – automatic validation when a property is set – is enabled only for entities attached to an *EntityManager*. But you can always validate any object, attached or not, by acquiring a DevForce *VerifierEngine* and passing the object into its *Execute* method.

In the following test, we acquire a *VerifierEngine* from the Manager and use it to validate a detached *Supplier* instance with a bad *CompanyName*. A comment shows that we could have created a new *VerifierEngine* if we preferred.

[TestMethod]

public void Can\_validate\_detached\_Supplier()

{

    var supplier = new Supplier { CompanyName = "Supplier" };

    // Manager.AddEntity(supplier); // commented out to keep supplier detached

    supplier.CompanyName = string.Empty;

    // var engine = new IdeaBlade.Validation.VerifierEngine();

    var engine = Manager.VerifierEngine;

    var validationResults = engine.Execute(supplier);

    var firstErr = validationResults.FirstOrDefault(r => r.IsError);

    Assert.IsNotNull(firstErr, "Expected a validation error");

    var msg = firstErr.Message;

    Assert.IsTrue(msg.Contains("required"),

                  "Error message did not contain 'required'; was " + msg);

}