Using Transfer Decorators to Deal with Invalid Data

Posted At : April 16, 2008 7:36 PM | Posted By : Bob Silverberg
Related Categories: ColdFusion, Transfer

Update: I have written another blog post that has an updated version of the populate() method discussed in this article. If this technique is of interest to you, I suggest checking out the populate() method here.

I came across what I feel is a nice solution to a sticky problem, so I thought I'd post about it. This issue has to do with form processing, so I'll start by describing a use case:

I have a Product Edit screen, which displays the details of a product and allows a user to edit them. When a user submits the form, the information they provide is validated. If any of the validations fail, the Product Edit screen is redisplayed, with the appropriate error messages. The data in each of the fields on this redisplayed screen should reflect what the user entered, even if that data was invalid (in fact, especially if that data was invalid).

Here's how that is implemented:

As I'm using Transfer, the form is coded such that the data displayed in each field is retrieved from the Transfer Object. Before the screen is first displayed, I do a:

```
<cfset ProductTO = getTransfer().get("product.product",1) />
```

and in the code of the screen each field is populated by calling a getter on the Transfer object, for example:

```
<input type="Text" name="Price" id="Price" value="#ProductTO.getPrice()#" />
```

When the form is submitted, I get the Transfer object again, and then call the populate() method on the Transfer object. populate() is a method that I've added to my AbstractTransferDecorator. It essentially takes the contents of the Form scope and "pushes" it into the Transfer object by calling its getters. Included in this process is a check to ensure that the data coming from the Form is valid for the datatype of the Transfer object's property. This is all done generically and automatically using TransferMetaData. After the automatic datatype validations are performed any business validations that are coded into the object's decorator are performed. If any of these validations fail, the Product Edit form is redisplayed, using the same instance of the Transfer object.

This allows the screen to be redisplayed, showing all of the values entered by the user, using the exact same view code (i.e., calling the getters on the Transfer object). Well, not in fact all of the values. If a user submits data of an invalid datatype, it cannot be pushed into the Transfer object. For example, if a user enters the value "abc" into the Price field, I cannot call setPrice("abc") as Transfer will throw an error because Price is a numeric field.

I need a way of tracking this data so that it can be redisplayed to the user along with all of the valid data, when a validation fails. I was using one method, which I didn't particularly like, and after speaking at length with Mark Mandel about it realized that I really needed to come up with something different. I then figured out that I could use onMissingMethod to create mock properties for any data that could not be put directly into a Transfer object's properties. Then I could override the getter in any objects that require this functionality (as many don't) to check for this property first. Let's look at some code to make this clearer.

First, here's my onMissingMethod handler, that allows for the use of these mock properties:

```
<cffunction name="onMissingMethod" access="public" output="false" returntype="Any" hint="Used to create mock properties to hold invalid data">

 <cfargument name="missingMethodName" type="any" required="true" />

 <cfargument name="missingMethodArguments" type="any" required="true" />



 <cfset var varName = 0 />

 <cfset var ReturnValue = "" />



 <!--- If we're trying to set an invalid value property, set it --->

 <cfif Left(arguments.missingMethodName,11) EQ "setInvalid_" AND StructKeyExists(arguments.missingMethodArguments,"1")>

  <cfset varName = ReplaceNoCase(arguments.missingMethodName,"setInvalid_","") &  "_Invalid" />

  <cfset variables.myInstance[varName] = arguments.missingMethodArguments.1 />

 <!--- If we're trying to get an invalid value property, get it --->

 <cfelseif Left(arguments.missingMethodName,11) EQ "getInvalid_">

  <cfset varName = ReplaceNoCase(arguments.missingMethodName,"getInvalid_","") &  "_Invalid" />

  <cfif StructKeyExists(variables.myInstance,varName)>

  <cfset ReturnValue = variables.myInstance[varName] />

  </cfif>

 </cfif>

 <cfreturn ReturnValue />



</cffunction>
```

This allows me to call setInvalid_Price("abc"), which will put that value into the myInstance struct in my decorator. Later, when I want to get that value, I can call getInvalid_Price and if one was set I get it back.

Here's a snippet from my populate() method where I use this method. It's quite long, so I'll not post the whole thing here - just enough so you can get an idea of how I'm using this. Note that the full version addresses any ManyToOne and ParentOneToMany that exist as well as the Properties.

```
<cffunction name="populate" access="public" output="false" returntype="void" hint="Populates the TO with values from a formstruct">
```

```
 <cfargument name="args" type="any" required="yes" />


 <!--- Get the MetaData and Properties --->
 <cfset var TransferMetadata = getTransfer().getTransferMetaData(getClassName()) />
 <cfset var Properties = TransferMetadata.getPropertyIterator() />
 <cfset var theProperty = 0 />
 <cfset var varName = 0 />
 <cfset var varType = 0 />
 <cfset var varValue = 0 />


 <!--- Loop through the properties --->
 <cfloop condition="#Properties.hasnext()#">
  <cfset theProperty = Properties.next() />
  <cfset varName = theProperty.getName() />
  <cfset varType = theProperty.getType() />
  <cfset varValue = arguments.args[varName] />
  <!--- validate the datatype --->
  <cfif IsValid(varType,varValue)>
   <!--- If valid, set the Property in the object --->
   <cfinvoke component="#this#" method="set#varName#">
    <cfinvokeargument name="#varName#" value="#varValue#" />
   </cfinvoke>
  <!--- If not, check for an empty value that needs to be passed as a null --->
  <cfelseif theProperty.getIsNullable() AND NOT Len(varValue)>
   <!--- Set the property to NULL --->
   <cfinvoke component="#this#" method="set#varName#Null" />
  <cfelse>
  <!--- Put the invalid value into an invalid value holder (uses onMissingMethod) --->
   <cfinvoke component="#this#" method="setInvalid_#varName#">
    <cfinvokeargument name="1" value="#varValue#" />
   </cfinvoke>
   <cfset ArrayAppend(arguments.args.Errors,"The contents of the " & varName & " field must be a valid " & varType & " value.") />
  </cfif>
 </cfloop>


</cffunction>
```

So, now I have my invalid value inside my decorator, so I should be able to get it back out in my view.

To do that, I have to write a decorator method for each property that needs this treatment. Luckily there aren't that many, so this is not a lot of work. Here's an example of how I'd do that with the Price field. This code is from my Product decorator:

```
<cffunction name="getPrice" access="public" output="false" returntype="any" hint="Checks for an invalid value for Price before returning">
 <cfset var Price_Invalid = this.getInvalid_Price() />
 <cfif Len(Price_Invalid)>
  <cfreturn Price_Invalid />
 <cfelse>
  <cfreturn getTransferObject().getPrice() />
 </cfif>
</cffunction>
```

Now, I've made a couple of decisions here that I'm not totally convinced of. One is the use of getInvalid_Price(), rather than just looking directly at variables.myInstance.Price_Invalid. Although this is a private variable inside the component, the code is actually separated out into two places. This getter code is in Product.cfc, whereas the setter code is in AbstractTransferDecorator.cfc. I was really torn between the two approaches, but I thought it best to hide the implementation of the mock properties from the Product decorator.

Also, I could probably do away with having to code each of these getters in the decorator by just creating a generic get() in my AbstractTransferDecorator, which would always check for an invalid value first, but I thought that would be overkill as there are so few properties that actually need this functionality.

So, finally, I can just call ProductTO.getPrice() in my view and I will always get the data submitted by the user, even if it is invalid.

I know that there are lots of ways of approaching this issue, and I'd be keen to hear what other people have done.