How I Use Transfer - Part X - My Abstract Transfer Decorator Object - Validations

Posted At : July 27, 2008 4:49 AM | Posted By : Bob Silverberg
Related Categories: OO Design, How I Use Transfer, ColdFusion, Transfer

In the previous post in this series about Transfer (an ORM for ColdFusion) I discussed the populate() method of my AbstractTransferDecorator. Having discussed some of the simple methods of this object in the post prior to that, what remains is a discussion of how I'm currently using this object to do validations.

There are three methods in this object that are used to support validations in my apps:

- **getValidations()** - This returns an array of business rules that apply to the Business Object.
- **addValidation()** - This is a helper method to make defining validations simpler.
- **validate()** - This contains generic validation routines that are shared by all Business Objects.

Let's start with getValidations(), which returns an array of business rules. This array does not include any information about implementing those rules, it simply describes the rule, and any requirements for the rule. For example, one rule might be *Required*, which simply means that the user must enter a value for the property. Another rule might be *Range*, in which case the array item would also include the upper and lower limits of the range. One of the nice things about this approach is that this array can be used to generate both client side and server side validations. So I only have to describe the validations for a given object once in my app, and then I have additional pieces of code which take these rules and automatically generate the JavaScript required for client validations, and the CF code required for server side validations. We'll see an example of the latter when we discuss the validate() method later in this post.

Because getValidations() simply returns an array, all that actually happens in a getValidations() method is that a number of items are added to an array, which is then returned. For this reason the getValidations() method in the Abstract Decorator isn't very interesting, as it doesn't define any actual validation rules. Here's what it looks like:

```
<cffunction name="getValidations" access="Public" returntype="any" output="false" hint="I get the validations for the object.">

 <cfargument name="Context" type="any" required="false" default="" />



 <cfset var Validations = ArrayNew(1) />

 <cfreturn Validations>



</cffunction>
```

This method accepts one argument, Context, which allows it to return different sets of validations depending on the context of the request. For example, for a user object, a password may or may not be required depending on whether they are registering for the first time or simply updating their profile.

To get a better idea of how this method is actually used, we'll take a peek inside a Concrete Decorator to see how it's actually implemented. This is the getValidations() method from promo.cfc, which is the Concrete Decorator for the Promo object:

```
<cffunction name="getValidations" access="Public" returntype="any" output="false" hint="I get the validations for the promo.">

 <cfargument name="Context" type="any" required="false" default="" />



 <cfset var Validations = ArrayNew(1) />

 <cfset Validations = addValidation(Validations,"Required","PromoName","Discount Name") />

 <cfset Validations = addValidation(Validations,"PositiveNumber","RequirementValue","Requirement Value") />

 <cfset Validations = addValidation(Validations,"Required","CriterionType","Product Criterion") />

 <cfif arguments.Context EQ "Coupon">

  <cfset Validations = addValidation(Validations,"Required","CouponCode","Coupon Code") />

 </cfif>

 <cfreturn Validations>



</cffunction>
```

I'm configuring four validations for the Promo object, the first three of which are always to be used, and the last one is only used if I'm working in the context of a Coupon. To set up these array items, I'm using the addValidation() helper method:

```
<cffunction name="addValidation" access="Public" returntype="any" output="false" hint="I add an item to the validations array.">

 <cfargument name="ValArray" type="array" required="yes" hint="The array to append to">

 <cfargument name="ValType" type="string" required="yes" hint="Type of validation">

 <cfargument name="PropertyName" type="string" required="yes" hint="Name of the property/form field">

 <cfargument name="PropertyDesc" type="string" required="no" default="#arguments.PropertyName#" hint="Descriptive name of the property">

 <cfargument name="FormName" type="string" required="no" default="frmMain" hint="The name of the form for which the validations are being added.">

 <cfargument name="Value1" type="numeric" required="no" default="0" hint="Optional numeric argument, used for max, min, ranges, etc.">

 <cfargument name="Value2" type="numeric" required="no" default="0" hint="Optional numeric argument, used for max, min, ranges, etc.">



 <cfset var theArray = arguments.ValArray>

 <cfset StructDelete(arguments,"ValArray")>

 <cfset ArrayAppend(theArray,arguments)>

 <cfreturn theArray>



</cffunction>
```

So, when configuring a validation, I must provide the following information:

- A Validation Type (e.g., Required, Range, PositiveNumber, etc.).
- The Property/Formfield Name that I want to validate against.

As well, the following information is optional:

- The descriptive name of the property/field. I supply this if it is different than the actual property name, which allows me to generate friendly validation failure messages.
- The name of the form to which this validation applies. Most of my forms are simply called frmMain, which is why this is an optional argument. This allows me to have a screen with multiple forms on it, and lets the system know that this validation only needs to be performed on one specific form. This is required for the generation of JavaScript validations. This needs to be refactored, as I can see how it couples the validations in this object to the view, sometimes.
- Value1 and Value2 allow me to pass in required numeric data, to be used by the validation rule, such as a Min value, a Max value, a Range, etc.

That just leaves the validate() method:

```
<cffunction name="validate" access="public" output="false" returntype="void" hint="Validates data for a Transfer Object.">
 <cfargument name="args" type="any" required="true" />
 <cfargument name="Context" type="any" required="false" default="" />

 <cfset var Validations = getValidations(arguments.Context) />
 <cfset var ValIndex = 0 />
 <cfset var Validation = 0 />
 <cfset var FieldValue = 0 />
 <cfset var CompValue = 0 />

 <cfif IsArray(Validations) and ArrayLen(Validations)>
  <cfloop from="1" to="#ArrayLen(Validations)#" index="ValIndex">
   <cfset Validation = Validations[ValIndex] />
   <cfif StructKeyExists(this,"get" & Validation.PropertyName)>
    <cfinvoke component="#this#" method="get#Validation.PropertyName#" returnvariable="FieldValue" />
    <cfswitch expression="#Validation.ValType#">
     <cfcase value="Required">
      <cfif NOT Len(FieldValue)>
       <cfset ArrayAppend(arguments.args.Errors,"You must provide the #Validation.PropertyDesc#.")>
      </cfif>
     </cfcase>
     <cfcase value="PositiveNumber">
      <cfif (FieldValue NEQ "" AND NOT IsNumeric(FieldValue)) OR Val(FieldValue) LT 0>
       <cfset ArrayAppend(arguments.args.Errors,"The #Validation.PropertyDesc# must be a positive number.")>
      </cfif>
     </cfcase>
     <cfcase value="Range">
      <cfif NOT IsNumeric(FieldValue) OR Val(FieldValue) LT Validation.Value1 OR Val(FieldValue) GT Validation.Value2>
       <cfset ArrayAppend(arguments.args.Errors,"The #Validation.PropertyDesc# must be a number between #Validation.Value1# and #Validation.Value2#.")>
      </cfif>
     </cfcase>
     <cfcase value="FutureDate">
      <cfif DateCompare(FieldValue,Now()) NEQ 1>
       <cfset ArrayAppend(arguments.args.Errors,"The #Validation.PropertyDesc# must be a date in the future.")>
      </cfif>
     </cfcase>
     <cfcase value="Email">
      <cfif Len(FieldValue) AND NOT isValid("email",FieldValue)>
       <cfset ArrayAppend(arguments.args.Errors,"The #Validation.PropertyDesc# must be a valid Email Address.")>
      </cfif>
     </cfcase>
    </cfswitch>
   </cfif>
  </cfloop>
 </cfif>
</cffunction>
```

This method contains the implementation of all of the server side code for standard validations that are shared by multiple business objects. Things like *Required*, *Positive Number*, *Future Date*, etc. The listing above is only partial, and I add validations to this method as I come up with new ones. Ideally I'd like to refactor this method into its own Abstract Validation Object, which could then act as a base object for Concrete Validation Objects that would be created for each Business Object. But for now it works.

The first thing that happens is that this method calls the getValidations() method, passing in the Context, to determine the set of validations to be performed. Then I loop through the validations checking to see if the property to be validated exists. If it does I retrieve the value of the property and then use a switch to perform the specified validation type.

The last piece of the validation puzzle is the set of rules that are specific to an individual Business Object. The validate() method in the AbstractTransferDecorator, that I just described above, only contains validations for common business rules. If I need to add Business Object specific validations I do that by extending this method in the Concrete Decorator. Here's an example from the Promo Object:

```
<cffunction name="validate" access="public" output="false" returntype="void" hint="Validates data for a Promo Transfer Object.">

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

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



 <cfset super.validate(arguments.args,arguments.Context) />

 <cfset arguments.args.Errors = checkCouponCode(arguments.args.Errors) />



</cffunction>
```

So first I call super.validate() to perform all of the "common" validation rules, and then I manually add any Business Object specific validations. In this case I need to check for the validity of the Coupon Code provided by the user, so I call another method in the decorator, checkCouponCode() to perform this validation.

I mentioned earlier that I can reuse the getValidations() method to generate client side JavaScript validations as well. I'm not going to go into the code for that, as it's somewhat lengthy and this post is already long enough, but in a nutshell, it works like this:

- In the controller, I call getValidations() for any Business Objects that are being displayed on a form. All of the validations are stored in an array called Validations.
- Before rendering the layout, I check to see whether the Validations array has any items.
- If it does I call a method on a Utility Object, passing in the Validations array, which generates all of the JavaScript code required to implement the client side validations.
- I then insert that code into my layout.

All of this is done automatically, so in order to add a new client side and server side validation, all I have to do is add a new addValidation() call to my getValidations() method in the Business Object's decorator.

And that's pretty much it! The code in these methods was actually written a long time ago, and was originally in a validations.cfc object that I was using outside of the model framework that I'm now discussing. I refactored it to move the methods into my Abstract Decorator, but have left the implementation pretty much the same, which explains why it's currently kind of "hacky", and not as OO as I'd like. Because it works, I haven't taken the time to refactor it further, but that is something that I will be doing in the future. I like the overall idea of what I'm doing, but I don't recommend taking it as an example of OO "best practices".

Whew, another long post. I hope at least one person has made it this far. That pretty much completes what I was hoping to cover in this series. I've now discussed the three main object types that comprise my model: Services, Gateways and Business Objects. I guess one thing that's missing is a "bringing it all together piece", where I describe how I implement everything using a controller and views. That in itself could become a whole new series, so I'll have to evaluate if and how to approach that. I'm leaving for two weeks vacation in an hour or two (had to squeeze this post out in a hurry), but I'll put some thought into where to go from here when I return. Anyone following who has suggestions for other areas that they'd like to see me cover, or anything that I have covered that leaves them with questions, please feel free to add a question or comment to this post and I'll do my best to address them upon my return.

Thanks for listening!