How I Use Transfer - Part IX - My Abstract Transfer Decorator Object - The Populate Method

Posted At : July 22, 2008 10:44 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 introduced my AbstractTransferDecorator and discussed some of its simpler methods. In this post I want to go through the populate() method in detail. I have posted bits and pieces of this method in the past, but I don't think I've ever documented the whole thing, as it stands today. I'll break it into pieces to make it a bit more manageable.

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

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

 <cfargument name="FieldList" type="any" required="no" default="" />



 <cfset var theFieldList = "" />

 <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 />

 <cfset var CompType = 0 />

 <cfset var hasIterator = false />

 <cfset var theIterator = 0 />

 <cfset var theComposition = 0 />

 <cfset var ChildClass = 0 />

 <cfset var ChildPKName = 0 />

 <cfset var theChild = 0 />
```

This method accepts just two arguments:

- **args**, which is a structure containing data with which to populate the object. In my apps, args is generally passed the attributes scope, which is where Fusebox puts all user input.
- **FieldList**, which is a list of fieldnames which can be used to limit the properties that are populated by this method.

To start, a bunch of local variables are declared. This includes requesting the TransferMetaData for the object's class from Transfer, as well as getting the PropertyIterator from that metadata.

Next, I loop through all of the object's properties:

```
<cfloop condition="#Properties.hasnext()#">

 <cfset theProperty = Properties.next() />

 <cfset varName = theProperty.getName() />

 <cfset varType = theProperty.getType() />

 <cfif NOT ListLen(arguments.FieldList) OR ListFindNoCase(arguments.FieldList,varName)>

  <cfif varName EQ "LastUpdateTimestamp" AND varType EQ "Date">

   <cfset setLastUpdateTimestamp(Now()) />

  <cfelseif Right(varName,4) EQ "Flag" AND varType EQ "Numeric">

   <cfif StructKeyExists(arguments.args,varName)>

    <cfset varValue = Val(arguments.args[varName]) />

   <cfelse>

    <cfset varValue = 0 />

   </cfif>

   <cfinvoke component="#this#" method="set#varName#">

    <cfinvokeargument name="#varName#" value="#varValue#" />

   </cfinvoke>

  <cfelseif StructKeyExists(arguments.args,varName)>

   <cfset varValue = arguments.args[varName] />

   <cfif variables.myInstance.CleanseInput>

    <cfset varValue = HTMLEditFormat(varValue) />

   </cfif>

   <cfif IsValid(varType,varValue)>

    <cfinvoke component="#this#" method="set#varName#">

     <cfinvokeargument name="#varName#" value="#varValue#" />

    </cfinvoke>

   <cfelseif theProperty.getIsNullable() AND NOT Len(varValue)>
```

```
    <cfinvoke component="#this#" method="set#varName#Null" />
   <cfelse>
    <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>
  </cfif>
 </cfif>
</cfloop>
```

I start by extracting the name and datatype of the property into local variables. I then use the FieldList to filter which properties get populated. This allows me to use this routine to populate only part of an object. An empty FieldList means that all properties get populated.

Most of my Business Objects have a LastUpdateTimestamp property, which should be populated with the current timestamp, so I do that next.

To deal with the age old checkbox problem, most of my boolean fields in my database are named xxxFlag. I also define these to Transfer as numeric, rather than boolean. So, if I'm at one of those properties in my loop, I check to see whether I've been passed a corresponding argument, and if so I use that value, converting blanks to zeros via Val(). If I haven't been passed anything I assume that a checkbox has gone unchecked, so I default the value to 0. Then I load that value into the object via its setter using cfinvoke.

Now that I've dealt with the special cases, I check to see whether a corresponding argument for the property has been passed in, and if so I save the value to a local variable. If my object has been configured to CleanseInput, which was discussed in the**previous post**, I use HTMLEditFormat() as part of a scheme to protect the site against Cross Site Scripting (XSS). I found that it is very rare that an object actually needs to allow true HTML to be loaded into it, so I find that this is a convenient way to approach this. If a particular object should allow HTML then I just override the value of CleanseInput in that object's configure() method.

I then validate the value passed in against the property's datatype. If it's valid, I set the property, if not I check to see if the property is nullable and whether the invalid value is in fact an empty string, in which case I set the property to null. If neither of those cases are true, then the value is in fact invalid, so I want to store it somewhere so I can display it back to the user. I use a "mock method" to store the invalid data in a private variable. This is made possible through the use of onMissingMethod(), which I'll describe at the end of this post. I also add an error message to the array of errors to be displayed to the user.

Now that all of the object's properties have been addressed, I turn my attention to the object's compositions. The hurdle I had to overcome here is that the values being submitted by the user for these compositions are usually just the primary key (or id, in Transfer terminology) of the associated object, but in order to load these into the object we need to first retrieve the actual object that corresponds to that key. What follows is my attempt to overcome that hurdle for ManyToOne and ParentOneToMany compositions:

```
<cfloop list="ManyToOne,ParentOneToMany" index="CompType">
 <cfinvoke component="#TransferMetadata#" method="has#CompType#" returnvariable="hasIterator" />
 <cfif hasIterator>
  <cfinvoke component="#TransferMetadata#" method="get#CompType#Iterator" returnvariable="theIterator" />
  <cfloop condition="#theIterator.hasnext()#">
   <cfset theComposition = theIterator.next() />
   <cfset varName = theComposition.getName() />
   <cfset ChildClass = theComposition.getLink().getTo() />
   <cfset ChildPKName = theComposition.getLink().getToObject().getPrimaryKey().getName() />
   <cfif StructKeyExists(arguments.args,ChildPKName)>
    <cfset varValue = arguments.args[ChildPKName] />
    <cfset theChild = getTransfer().get(ChildClass,varValue) />
    <cfif theChild.getIsPersisted()>
     <cfif CompType CONTAINS "Parent">
      <cfset varName = "Parent" & theComposition.getLink().getToObject().getObjectName() />
     </cfif>
     <cfinvoke component="#this#" method="set#varName#">
      <cfinvokeargument name="transfer" value="#theChild#" />
     </cfinvoke>
    </cfif>
   </cfif>
  </cfloop>
 </cfif>
</cfloop>


</cffunction>
```

The code for each composition type (ManyToOne and ParentOneToMany) is pretty much identical, so I loop over my list of composition types, effectively executing the code block once for ManyToOne and once for ParentOneToMany.

I first check to see if an Iterator for the composition type exists, and if so I get a copy of it from the TransferMetaData. I then use the Iterator to loop through the compositions defined for the object, extracting the name of the composition, the Transfer class of the child, and the name of the primary key of that child.

If an argument has been passed in that corresponds with the primary key that I just found, e.g., The primary key is *ProductId* and an argument of *ProductId* was passed in, I do the following:

- extract the value of the argument
- get the child object from Transfer
- check to see whether I've been handed back an object from the database, using getIsPersisted()

- if I do find a valid Transfer Object, I want to load that object into the current object

The wrinkle is that if I'm working with a ParentOneToMany composition the name of the set() method needs to be determined differently than if I'm working with a ManyToOne, so I address that with an if statement.

When that loop is done my object is fully loaded, so I end the method.

I mentioned my use of onMissingMethod() above, so here's the code for that:

```
<cffunction name="onMissingMethod" access="public" output="false" returntype="Any" hint="Very useful!">

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

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


 <cfset var varName = 0 />

 <cfset var ReturnValue = "" />


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

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

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

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

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

  <cfif StructKeyExists(variables.myInstance,varName)>

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

  </cfif>

 </cfif>

 <cfreturn ReturnValue />

</cffunction>
```

Basically what I'm doing here is allowing for an unlimited number of Invalid private properties, which can be accessed via standard getters and setters. I actually go into a lot more detail about this approach and why I'm doing this in **this blog post**, so I won't repeat it all here. I know it's not perfect, but I have yet to come up with a better method that would allow me to use my Business Object to keep track of the invalid data that a user has entered. Feedback in this area would be greatly appreciated.

I do recall someone indicating that there was a problem with the logic of my populate() method, that would make it unreliable, but it seems to work consistently for me. If anyone gives it a try and encounters any strange results please let me know.

The only area that I see changing dramatically at this point is the validations. My original thought was that I was already documenting the datatypes of the Business Objects in my transfer.xml file, so I should just use that data to perform these validations. I still see the logic in that statement, but I'd rather move all of my validation logic into external validation objects, so I'll probably do away with that part of the code. I'll still have to address the issue of storing invalid values, and dealing with nullable properties, but I'm sure that there are other solutions to those problems.

Speaking of validations, that is the last part of the AbstractTransferDecorator left to describe, so I'll address that in my next post.