How I Use Transfer - Part IV - My Abstract Service Object

Posted At : July 3, 2008 11:11 AM | Posted By : Bob Silverberg
Related Categories: OO Design, How I Use Transfer, ColdFusion, Transfer

In the previous post in the series I discussed how I base most of my objects on Abstract Objects, which allows me to eliminate a lot of duplicate code. I then took a look at one method in my AbstractService object to demonstrate this. In this post I'm going to look at the rest of the methods in the AbstractService, so be prepared for a lot of code.

Let's start with the Init() method and the Configure() method:

```
<cffunction name="Init" access="Public" returntype="any" output="false" hint="I build a new Service">
 <cfargument name="AppConfig" type="any" required="true" />
 <cfset variables.Instance = StructNew() />
 <cfset variables.Instance.AppConfig = arguments.AppConfig />
 <cfset Configure() />
 <cfreturn this />
</cffunction>

<cffunction name="Configure" access="Public" returntype="void" output="false" hint="I am run by the Init() method">
</cffunction>
```

I am using Coldspring to manage all of my singletons, which in my case are all of the Services, Gateways and some Utility Objects, so I provide an Init() method for all of my services, which is called automatically by Coldspring when it creates them.

The AppConfig object that is loaded into the Service during the Init() method contains a bunch of information about how the app is configured, some of which is required by some services. I wanted to avoid having to extend this Init() method in my concrete services, because I was worried that I might accidentally mess up the API, so I created a Configure() method as well. In the AbstractService the Configure() method is empty, so by default it does nothing. In order to have a service perform certain actions when it is created I need to create a Configure() method in the concrete Service Object. This does introduce a tiny bit of overhead, but I felt it was cleaner. Now I can have the only implementation of Init() in the AbstractService, and work with the Configure() method as needed.

We looked at the Get() method in the previous article, so that just leaves GetList, Update, Delete and some accessors that are used by Coldspring to compose the service. Let's walk through those, as they'll crop up when we look at the other methods:

```
<cffunction name="getTransferClassName" access="public" output="false" returntype="any">
 <cfreturn variables.Instance.TransferClassName />
</cffunction>
<cffunction name="setTransferClassName" returntype="void" access="public" output="false">
 <cfargument name="TransferClassName" type="any" required="true" />
 <cfset variables.Instance.TransferClassName = arguments.TransferClassName />
</cffunction>

<cffunction name="getEntityDesc" access="public" output="false" returntype="any">
 <cfreturn variables.Instance.EntityDesc />
</cffunction>
<cffunction name="setEntityDesc" returntype="void" access="public" output="false">
 <cfargument name="EntityDesc" type="any" required="true" />
 <cfset variables.Instance.EntityDesc = arguments.EntityDesc />
</cffunction>

<cffunction name="getTheGateway" access="public" output="false" returntype="any">
 <cfreturn variables.Instance.TheGateway />
</cffunction>
<cffunction name="setTheGateway" returntype="void" access="public" output="false">
 <cfargument name="TheGateway" type="any" required="true" />
 <cfset variables.Instance.TheGateway = arguments.TheGateway />
</cffunction>

<cffunction name="getTransfer" access="public" output="false" returntype="any">
 <cfreturn variables.Instance.Transfer />
</cffunction>
<cffunction name="setTransfer" returntype="void" access="public" output="false">
 <cfargument name="transfer" type="any" required="true" />
 <cfset variables.Instance.Transfer = arguments.Transfer />
</cffunction>

<cffunction name="getTransientFactory" access="public" output="false" returntype="any">
```

```
 <cfreturn variables.Instance.TransientFactory />

</cffunction>

<cffunction name="setTransientFactory" returntype="void" access="public" output="false">

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

 <cfset variables.Instance.TransientFactory = arguments.TransientFactory />

</cffunction>



<cffunction name="getEmailService" access="public" output="false" returntype="any">

 <cfreturn variables.Instance.EmailService />

</cffunction>

<cffunction name="setEmailService" returntype="void" access="public" output="false">

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

 <cfset variables.Instance.EmailService = arguments.EmailService />

</cffunction>



<cffunction name="getTranslationService" access="public" output="false" returntype="any">

 <cfreturn variables.Instance.TranslationService />

</cffunction>

<cffunction name="setTranslationService" returntype="void" access="public" output="false">

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

 <cfset variables.Instance.TranslationService = arguments.TranslationService />

</cffunction>



<cffunction name="getFileSystem" access="public" output="false" returntype="any">

 <cfreturn variables.Instance.FileSystem />

</cffunction>

<cffunction name="setFileSystem" returntype="void" access="public" output="false">

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

 <cfset variables.Instance.FileSystem = arguments.FileSystem />

</cffunction>



<cffunction name="getAppConfig" access="public" output="false" returntype="any">

 <cfreturn variables.Instance.AppConfig />

</cffunction>
```

Let's briefly go over these values and objects that are composed into every service. Values for the first three are specified in the Coldspring config file, while the rest are autowired:

- TransferClassName - This is the classname (Package.Name) that must be used to identify the main entity to Transfer. For example, for the UserService this would be user.user. We saw an example of the use of this property in the Get() method described in the previous article.
- EntityDesc - This is a user-friendly description of the main entity (e.g, User). This is used in messages generated by the service.
- TheGateway - This is a Gateway Object that provides queries for the main entity (e.g., UserGateway).
- Transfer - This is the high level Transfer object from the TransferFactory.
- TransientFactory - This is a factory that I wrote that creates Transient objects for me. As I only have a few transient objects that need to be created, I just have one TransientFactory which creates them all. If a service needs to create a transient it uses this factory.
- EmailService - If a service needs to send an email, it can do so via the composed EmailService.
- TranslationService - If a service needs to translate from one language to another (e.g., for user messages), it can do so via the composed TranslationService.
- FileSystem - Interaction with the file system (e.g., file uploads, copying, etc.), is achieved via the composed FileSystem object.
- AppConfig - I have chosen to access all of my private variables via getters. So rather than referring to variables.Instance.AppConfig, I just use getAppConfig(). I know there has been a lot of debate about this approach, and I believe that it's really a matter of personal preference, and this is what I prefer.

OK, so now we've seen what makes up the AbstractService, let's take a look at the remaining methods:

```
<cffunction name="GetList" access="public" output="false" returntype="query" hint="Gets a default listing from the default gateway">

 <cfargument name="args" type="struct" required="yes" hint="Pass in the attributes structure.">

 <cfreturn getTheGateway().getList(argumentCollection=arguments.args) />

</cffunction>
```

I am using this model with **Fusebox**, so all of the data supplied by the user (e.g., Form and URL variables) will be sitting in the attributes scope, so that's what I pass into this method. The args argument will contain any criteria specified for the listing. Then all I do is pass that criteria to the getList() method of the default Gateway. I don't override this method in any of my Service Objects, rather, the specific implementation is dealt with by different getList() methods in the Gateway Objects themselves.

The Delete() method is pretty simple, so let's look at that next:

```
<cffunction name="Delete" access="public" output="false" returntype="struct" hint="Used to Delete a record">

 <cfargument name="theId" type="any" required="yes" hint="The Id of the record to delete.">
```

```
 <cfset var ReturnStruct = StructNew() />
 <cfset var theTO = get(arguments.theId) />
 <cfif theTO.getIsPersisted()>
  <cfset theTO.delete() />
  <cfset ReturnStruct.sScreenMessage = "OK, the #getEntityDesc()# has been deleted.">
 </cfif>
 <cfreturn ReturnStruct>
</cffunction>
```

I start by getting the Business Object requested, which I do by using the Get() method of the service which will return a Transfer Object to me. If a corresponding record exists in the database I then ask the object to delete itself and I set a message to display to the user. Note that this does not deal with any sort of cascading deletes. If I need that functionality I override this method in my concrete Service Object.

Note that I have recently refactored this method to push logic down into the Business Object. It used to look like this:

```
<cffunction name="Delete" access="public" output="false" returntype="struct" hint="Used to Delete a record">
 <cfargument name="theId" type="any" required="yes" hint="The Id of the record to delete.">

 <cfset var ReturnStruct = StructNew() />
 <cfset var theTO = get(arguments.theId) />
 <cfif theTO.getIsPersisted()>
  <cfset getTransfer().delete(theTO) />
  <cfset ReturnStruct.sScreenMessage = "OK, the #getEntityDesc()# has been deleted.">
 </cfif>
 <cfreturn ReturnStruct>
</cffunction>
```

I know that there has also been a lot of debate about which of the above two methods is preferable, and I personally prefer the first example. One of the advantages I see in this is that it allows me to minimize direct references to Transfer in the service layer.

That leaves just the Update() method. Honestly, I'm not that pleased with the design of this method. It works very well, doing exactly what I need it to do, but it definitely is much more procedural than I ultimately want it to be. I have a number of ideas about how I can improve it, but have not yet taken the time to refactor it. I guess due to time constraints this is one of those "if it ain't broke" situations.

```
<cffunction name="Update" access="public" output="false" returntype="struct" hint="Used to add or update a record.">
 <cfargument name="theId" type="any" required="yes" hint="The Id of the record.">
 <cfargument name="args" type="struct" required="yes" hint="Pass in the attributes structure.">
 <cfargument name="Context" type="any" required="no" default="" />

 <cfset var ReturnStruct = StructNew() />
 <cfset var sAction = "updated" />
 <cfset var theTO = Get(arguments.theId,true,arguments.args) />
 <cfset var Upload = 0 />
 <cfset var fld = 0 />
 <cfset var varName = 0 />
 <cfset ReturnStruct.sScreenMessage = "" />
 <cfif NOT ArrayLen(arguments.args.Errors)>
  <!--- datatype validations passed --->
  <cfset theTO.validate(arguments.args,arguments.Context)>
  <cfif NOT ArrayLen(arguments.args.Errors)>
   <!--- Business rule validations passed --->
   <cfif NOT Val(arguments.theId)>
    <cfset theTO.onNewProcessing(arguments.args) />
    <cfset sAction = "added">
   </cfif>
   <cfif StructKeyExists(arguments.args,"FieldNames") AND arguments.args.FieldNames CONTAINS "UploadFile">
    <cfloop collection="#form#" item="fld">
     <cfif ListFirst(fld,"_") EQ "UploadFile" AND Len(form[fld])>
      <cfset Upload = getFileSystem().upload(getDestination(),fld) />
      <cfif Upload.getSuccess()>
       <cfset varName = ListLast(fld,"_") />
       <cfif StructKeyExists(theTO,"set" & varName)>
        <cfinvoke component="#theTO#" method="set#varName#">
```

```
        <cfinvokeargument name="#varName#" value="#Upload.getServerFile()#" />
       </cfinvoke>
      </cfif>
     <cfelse>
      <cfset ArrayAppend(arguments.args.Warnings,"A file upload was unsuccessful.") />
     </cfif>
    </cfif>
   </cfloop>
  </cfif>
  <cfset theTO.save() />
  <cfset ReturnStruct.sScreenMessage = "OK, the #getEntityDesc()# has been #sAction#." />
 </cfif>
</cfif>
<cfset ReturnStruct.theTO = theTO />
<cfreturn ReturnStruct>
</cffunction>
```

I pass in the Id of the entity, as well as a structure that contains all of the values that should be populated into the object. I'm also passing in a context, which is used for validations inside the decorator. This is an idea that I picked up from **Paul Marcotte**.

Once again, I'm getting my Business Object using the service's Get() method, but this time I'm passing in a struct of values to use to populate the object. I'm also telling it (via the second argument) that I need a**cloned** object back. I'm doing that because I'm going to be putting these values into the Transfer Object before I've had a chance to validate them against business rules. I don't want Transfer's cache to reflect these new values until the object is saved, so I use a cloned object.

I have an array of errors, which is stored in attributes.Errors outside of the model. This therefore is passed in as arguments.args.Errors, and that array gets updated any time an error is encountered. During the population of the Transfer Object, which occurs inside of the Get() method, I check for any datatype errors, adding them to the array. Hence the check for ArrayLen(arguments.args.Errors) before proceeding.

If there were no datatype errors, I ask the Business Object to validate itself, passing in the context. This allows me to define different sets of validations in the Business Object for different scenarios. Once again, if arguments.args.Errors is empty I can continue.

Next I'm checking to see whether I'm adding a new record or updating an existing record. If the Id passed in is blank or zero, then, according to my business rules, it's a new record, so I ask the Business Object to do any processing that is required for new records. An example of this would be to set the status of a new Review to "Submitted". In my AbstractTransferDecorator this onNewProcessing() method is empty, so by default nothing happens. If I need a Business Object to perform some special processing on new records I add that method into the concrete decorator.

I originally tried to use the **BeforeCreate event** of Transfer's Event Model for this, but the processing that I wanted to do required data submitted by the user, and the Event Model does not provide a way to pass that in. That is why this method gets passed the args argument, which contains all of the data submitted by the user.

I often need to upload files when adding or updating a record, so I've included logic to do that as well. This is an area that definitely needs work from a design perspective. It works very much by convention. Any File input fields will be called "UploadFile_xxx", where xxx is the name of the property in which I want to store the file name. For example, if I have a Product and I need to upload two image files, the names of which should be stored in properties called ImageSmall and ImageLarge, my two File inputs would be called UploadFile_ImageSmall and UploadFile_ImageLarge.

For any FileUpload fields that I find, I call the upload() method of my composed FileSystem object, which will attempt to upload the file, and report back success or failure. If it was successful, I attempt to set properties in my Business Object as described in the paragraph above. If it is not successful I simply add a generic warning message to be displayed to the user. This error reporting is not ideal, but isn't really an issue for any of the apps with which I'm using this.

Finally I ask the Business Object to save itself and then create a message to display to the user.

And that's pretty much it for my AbstractService. I'd say that the majority of my concrete services inherit these methods from the AbstractService, so that saves me a lot of coding, and also encapsulates the logic nicely. For any situations where this generic logic does not meet my business requirements I simply extend (via super) or override the method in my concrete service.

This was a good exercise for me as going through all of this has raised some questions in my mind and given me a few ideas. I'll present them here, and would be happy to hear any feedback on them:

1. How much of this logic could I actually push into the Business Object itself via the decorator? All of these examples deal with a single business object, so it seems possible, but often the cases where I override these methods require interaction with multiple business objects, which makes me think the service is the best place for them.
2. I'm wondering about my adoption of the Configure() method. Does my reasoning make sense, or should I just be extending Init() where needed?
3. I think I should move the logic for processing the file uploads into a separate object - perhaps the FileSystem object. I could just pass the full structure in and allow it to do the looping and extraction.
4. I have some ideas for improving the way I do validations as well, many of which are thanks to **Brian Kotek** and **Paul Marcotte**. The internals of the validations are actually in the decorator, so we haven't seen them yet, but those changes will also impact the way the Update() method looks down the road - there will be less conditional logic.

Whew, that was a long post. If you're still reading, congratulations and thanks! In the next installment I'll take it easy and look at my AbstractGateway.