How I Use Transfer Today - Encapsulating Database Access

Posted At : November 12, 2008 10:59 AM | Posted By : Bob Silverberg
Related Categories: OO Design, How I Use Transfer, ColdFusion, Transfer

I've been meaning to follow up on my **How I Use Transfer** series, as I've made a few changes to the way I write my ColdFusion code since that series was written. One of the biggest changes was to encapsulate all database access in my Gateway components.

Now I say biggest not because it took a lot of time and effort to make the change, in fact the opposite is true. I say biggest simply because it represented a significant shift in the way I'm designing my model. Let me start with the rationale for making this change.

A lot of my design decisions are based on encapsulation, and I find the object oriented principle "encapsulate what varies" to be a good guiding principle. I tend to ask myself, "Self, if something were to change, how many components would have to change as a result?" Ideally the answer to that question would be "one" (notice I said*ideally*). So, regarding database access, what can change?

- Our underlying database could change. For example, moving from SQL Server to MySQL.
- Our database structure could change. For example, adding/modifying tables, columns, etc.
- *If* we're using an ORM, our ORM could change. For example, moving from Transfer to a different ORM (heresy!).

Many of us already use gateways to address the first two issues. We put all of our queries, written in either SQL or TQL into gateways because, if something were to change about our database we would like to be in a position where we only have to change the Gateway. In a large part because of comments made by**Brian Kotek**, I came to the realization that, as long as I'm attempting to encapsulate database access, I should really encapsulate access to the ORM as well.

So, what does that mean in terms of Transfer? It means that the only place I should be putting any references to Transfer is in a Gateway. That means no references to Transfer in my Controllers (not that there ever was, but I just thought I'd point it out as a rule), no references to Transfer in my Services, and no references to Transfer in my Decorators. The only component that ever talks to Transfer is the Gateway.

Now, what makes this fairly simple to implement is the fact that I can simply inject the gateway into my Service and into my Decorator, so I can pretty much do exactly what I was doing before, but instead of talking to Transfer I talk to a Gateway. Another thing that makes it simple for me is that I'm using abstract objects, so for the most part my code only has to change in the abstract objects (i.e., only one object has to change), rather than requiring me to make changes to dozens of concrete objects.

OK, time for some examples. I'm going to take a before and after approach, looking at code from my previous Transfer series and code as I'm writing it today. Let's start with the get() method in my Abstract Service object. Here's the old version:

```
<cffunction name="get" access="public" returntype="any">
 <cfargument name="theId" type="any" required="yes" />
 <cfargument name="needsClone" type="any" required="false" default="false" />
 <cfargument name="args" type="any" required="no" default="" />
 <cfargument name="TransferClassName" type="any" required="no" default="#getTransferClassName()#" />
 <cfargument name="FieldList" type="any" required="no" default="" />
 <cfset var theTO = 0 />
 <cfif arguments.theId EQ 0>
  <cfset theTO = getTransfer().new(arguments.TransferClassName) />
 <cfelse>
  <cfset theTO = getTransfer().get(arguments.TransferClassName,arguments.theId) />
  <cfif arguments.needsClone>
   <cfset theTO = theTO.clone() />
  </cfif>
 </cfif>
 <!--- If an args struct was passed in, use it to populate the TransferObject --->
 <cfif IsStruct(arguments.args)>
  <cfset theTO.populate(arguments.args,arguments.FieldList) />
 </cfif>
 <cfreturn theTO />
</cffunction>
```

And here's the new version:

```
<cffunction name="get" access="Public" returntype="any"
 hint="I return a Business Object, ready to be used.">
 <cfargument name="theId" type="any" required="yes" />
 <cfargument name="readOnly" type="any" required="false" default="true" />
 <cfargument name="args" type="any" required="no" default="" />
 <cfargument name="Result" type="any" required="no" default="" />
 <cfargument name="MainObjectName" type="any" required="no" default="#getMainObjectName()#" />
 <cfargument name="FieldList" type="any" required="no" default="" />
 <cfset var theTO = getTheGateway().get(arguments.theId,arguments.readOnly,arguments.MainObjectName) />
 <!--- If an args struct was passed in, use it to populate the TransferObject --->
 <cfif IsStruct(arguments.args) AND IsObject(arguments.Result)>
  <cfset theTO.populate(arguments.args,arguments.Result,arguments.FieldList) />
 </cfif>
 <cfreturn theTO />
</cffunction>
```

So I've moved all of the logic that has anything to do with Transfer into the Gateway, thereby encapsulating access to the ORM. Just ignore the *Result* stuff, that's part of another aspect to my new approach, which also makes use of my **validation framework**. You can probably guess what's inside the Gateway's get() method, but let's take a look anyway:

```
<cffunction name="get" access="Public" returntype="any">

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

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

 <cfargument name="MainObjectName" type="any" required="no" default="#getMainObjectName()#" />

 <cfset var theTO = 0 />

 <cfif (IsNumeric(arguments.theId) AND NOT Val(arguments.theId)) OR NOT Len(arguments.theId)>

  <cfset theTO = getTransfer().new(arguments.MainObjectName) />

 <cfelse>

  <cfset theTO = getTransfer().get(arguments.MainObjectName,arguments.theId) />

  <cfif NOT arguments.readOnly>

   <cfset theTO = theTO.clone() />

  </cfif>

 </cfif>

 <cfreturn theTO />

</cffunction>
```

So that was a pretty quick and easy refactor. Let's take a look at a couple of the methods in my Abstract Transfer Decorator. Here's the old version:

```
<cffunction name="save" access="public" returntype="void">

 <cfset getTransfer().save(this) />

</cffunction>


<cffunction name="delete" access="public" returntype="void">

 <cfset getTransfer().delete(this) />

</cffunction>
```

And here's the new version:

```
<cffunction name="save" access="public" returntype="void">

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

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

 <cfset validate(arguments.Result,arguments.Context) />

 <cfif arguments.Result.getIsSuccess()>

  <cfset getTheGateway().save(this) />

  <cfset Result.setSuccessMessage("OK, the #variables.myInstance.ObjectDesc# record has been saved.") />

 </cfif>

</cffunction>


<cffunction name="delete" access="public" returntype="void">

 <cfset getTheGateway().delete(this) />

</cffunction>
```

Again, it's pretty obvious that I've simply moved the Transfer "stuff" into the Gateway. All of the extra stuff in my save() method (i.e., Result, validate and Context) have to do with the new way that I've implemented validations using my framework. Let's take a look at the save() and delete() methods in the Abstract Gateway:

```
<cffunction name="save" access="public" returntype="void">

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

 <cfset getTransfer().save(arguments.theTO) />

</cffunction>


<cffunction name="delete" access="public" returntype="void">

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

 <cfset getTransfer().delete(arguments.theTO) />

</cffunction>
```

Another quick and easy refactor, allowing me to achieve my objective without having to write or change much code at all. There are other specific cases where I was referencing Transfer directly in my Services and Decorators, and I've moved most of those into the Gateways as well.

Let me close by saying that I am not putting this forward as a recommendation or best practice. This just happens to be the way that *I'm* doing it, and I hope that I've made my reasons clear. I do accept that I'm adding an extra layer of abstraction to my model, which is in fact my intention, and I can totally see the argument that some may not wish to add this, and that it's much simpler to just talk to Transfer via Services and Decorators. That's one of the great things about software development, there are always many *good* solutions to any given problem.