How I Use Transfer - Part V - A Concrete Service Object

Posted At : July 8, 2008 7:15 AM | Posted By : Bob Silverberg
Related Categories: OO Design, How I Use Transfer, ColdFusion, Transfer

I was going to describe my Abstract Gateway Object in this post, but during a conversation with a fellow developer it was suggested that I should take a moment to describe a Concrete Service Object, as there was still a bit of confusion in his mind about how I use the Abstract Service Object.

To recap a bit, I have an Abstract Service Object and it is used as an extension point for most of my Concrete Service Objects. Perhaps a bit more of a definition is in order.

- **The Abstract Service Object**
- Is never instantiated as an object.
- Cannot be used as is.
- Is only ever used as a base object for Concrete Service Objects.
- There is only one Abstract Service Object, called AbstractService.cfc.
- Does not have any Transfer classes associated with it.

- **Concrete Service Objects**
- Are instantiated as objects.
- Methods on them are called by Controllers, other Concrete Service Objects and Business Objects.
- Most extend AbstractService.cfc.
- There are many Concrete Service Objects, e.g., UserService.cfc, ProductService.cfc, ReviewService.cfc, etc.
- Have one "main" Transfer class associated with them, but can interact with others via code specific to the Concrete Service.

I'll digress for a moment to discuss the comment that "Most extend AbstractService.cfc." Really, the Abstract Service Object is a starting point for all Service Objects that persist their data in a database, like UserService, ProductService, etc. If a Service Object does not persist data in a database, it really gains nothing by extending AbstractService. For example, I have a CartService Object, which only persists data in the session scope. Therefore my CartService Object does not extend AbstractService.

Let's take a look at an example of a Concrete Service Object, ReviewService.cfc. To start, here's the Bean definition of this service from my Coldspring config file:

```
<bean id="ReviewService" class="model.service.ReviewService">

 <property name="TransferClassName"><value>product.review</value></property>

 <property name="EntityDesc"><value>Review</value></property>

 <property name="TheGateway">

  <ref bean="ReviewGateway" />

 </property>

</bean>
```

In here I indicate that the main Transfer class with which this service interacts is *product.service*. That means that calls to Get(), Update() and Delete() will be directed at the table defined to Transfer as product.review. I can write additional methods in my service that will interact with other Transfer classes, but the default methods, inherited from the AbstractService, will be directed at product.review.

I also indicate that the description of the main class is *Review*. That will be used for UI messages (e.g., "The Review has been updated"). And I specify that the *ReviewGateway*, which is defined in a separate Coldspring bean, is the default Gateway Object for this service. That means that calls to GetList() will be directed at that Gateway Object.

And here's the code for ReviewService.cfc:

```
<cfcomponent displayname="ReviewService" output="false" hint="I am the service layer component for the Review model." extends="AbstractService">



<cffunction name="Get" access="Public" returntype="any" output="false" hint="I override the abstract get in order to determine the proper ReviewId for a member.">

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

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

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



 <cfset var TQL = "" />

 <cfset var TQuery = "" />

 <cfset var qryReview = "" />

 <cfset var ReviewId = 0 />



 <cfif StructKeyExists(arguments.args,"CurrentUser")>

  <cfif arguments.args.CurrentUser.IsAdmin()>

   <cfset ReviewId = arguments.theId />

  <cfelse>

   <cfsavecontent variable="TQL">

    <cfoutput>

     SELECT Review.ReviewId

     FROM product.review AS Review

     JOIN product.product AS Product

     JOIN user.user AS TUser

     WHERE TUser.UserId = :UserId

     AND  Product.ProductId = :ProductId

    </cfoutput>

   </cfsavecontent>
```

```
   <cfset TQuery = getTransfer().createQuery(TQL) />

   <cfset TQuery.setParam("UserId",arguments.args.CurrentUser.getUserId()) />

   <cfset TQuery.setParam("ProductId",arguments.args.ProductId) />

   <cfset TQuery.setCacheEvaluation(true) />

   <cfset qryReview = getTransfer().listByQuery(TQuery) />

   <cfif qryReview.RecordCount>

    <cfset ReviewId = qryReview.ReviewId />

   </cfif>

  </cfif>

 </cfif>


 <cfreturn super.Get(ReviewId,arguments.needsClone,arguments.args) />

</cffunction>


</cfcomponent>
```

No big surprise, there's almost nothing in there! My ReviewService is inheriting GetList(), Update() and Delete() from the AbstractService, as it doesn't have to do anything special in those methods. The only method that I need to override (in fact I'm extending it, not overriding it) is Get().

The issue with Get() is that I have two different algorithms for determining which Review I should return, depending on whether the current user is logged in as an administrator or not. If the user *is* an administrator then I just return whichever review they requested, as an administrator is able to read and edit all reviews. However, if the user *is not* an administrator then I must return only *their own* review.

Because Review is a child of Product, and User is a child of Review:

```
<package name="product">

 <InvalidTag name="product" table="tblProduct" decorator="model.product">

  <id name="ProductId" type="numeric" />

  <property name="ProductName" type="string" column="ProductName" />

  ...

  <onetomany name="Review" lazy="true">

    <link to="product.review" column="ProductId"/>

   <collection type="array">

    <order property="LastUpdateTimestamp" order="desc" />

   </collection>

  </onetomany>

 </object>

 <InvalidTag name="review" table="tblReview" decorator="model.review">

  <id name="ReviewId" type="numeric" />

  <property name="Rating" type="numeric" column="Rating" />

  <property name="LastUpdateTimestamp" type="date" column="LastUpdateTimestamp" />

  ...

  <manytoone name="User">

   <link to="user.user" column="UserId"/>

  </manytoone>

 </object>

</package>
```

I need to use TQL to join the objects together to find the Review that corresponds to the current user, and the ProductId (which is passed in via args). That's what the bulk of the code above is doing. Once I have the proper ReviewId, I then call super.Get() to actually return the Transfer Object. This allows me to use all of the logic that is built in to the Get() method in the AbstractService, so I don't need to duplicate any of that in the ReviewService.

So that's a simple example of a Concrete Service Object that extends my Abstract Service Object. I actually have one "special" Concrete Service Object, called ValueListService, which I use to manage all of my "code" or "lookup" objects. This is used for objects like UserGroup, OrderStatus, ProductCategory, Colour, etc. It too extends AbstractService, but is built itself in an abstract way so that I can use it to manage all of those "code" objects, without having to write a Concrete Service Object for each one. I plan on discussing that in a future post.

In the next installment I'll start looking at the Abstract Gateway Object.