ValidateThis! - Server Side Validation Architecture

Posted At : October 14, 2008 2:27 PM | Posted By : Bob Silverberg
Related Categories: OO Design, ColdFusion, Transfer, ValidateThis

In this installment of my series about object oriented validations with ColdFusion I'm going to describe the overall architecture of the framework. I have tried to apply the following object oriented principles as I've been designing this, which should help me achieve the goals described in the first article in the series:

- Encapsulate what varies.
- Don't repeat yourself (DRY).
- The Open-Closed Principle: Classes should be open for extension, yet closed for modification.

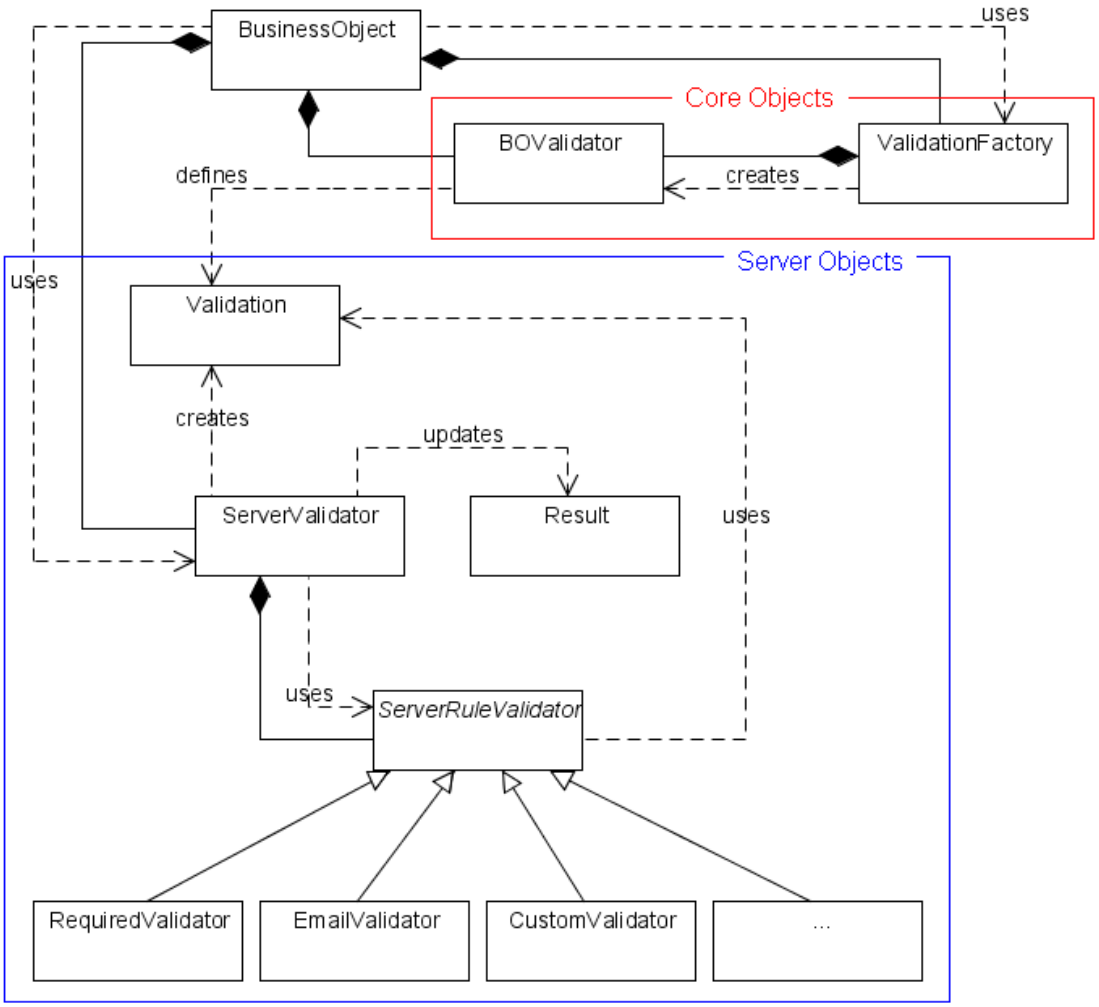
Following those principles has yielded a design that results in a lot of objects and may seem unnecessarily complex. I believe that the complexity is required to achieve the goals, but some may disagree. One of my reasons for writing about how I've designed this is to solicit feedback from any interested parties which may solidify and/or change that belief.

Because the framework is used to generate both client-side and server-side validations, there are three categories of objects:

- Core Objects, which are used for both client-side and server-side validations.
- Server Objects, which are used only when performing server-side validations.
- Client Objects, which are used only when generating client-side validation code.

In an attempt to limit the complexity of this discussion I'm only going to describe the Core Objects and the Server Objects in this post. I will describe the Client Objects in a future post.

Because a picture is worth 1024 words, I created a diagram of the objects involved. I found that a UML class diagram was a useful container for what I'm trying to describe, but I haven't followed the rules for a true class diagram. The picture below has been shrunk to fit on the blog page, to see a larger version just click on the picture itself.



Note that these are not actually all of the objects used by the framework. There are a few "helper" objects which I will describe when I get to the implementation, but for now they're unimportant. I want to walk through the objects in the picture, but first a thought:

I didn't intend to show any code as part of this post. I really wanted to focus on the design rather than the implementation, but as I read and edited this post it became apparent that the whole thing might be very confusing without any point of reference. So I went back and added very simple code samples to some of the object descriptions. For the most part these are not examples of how the object was implemented, but rather examples of how the object would be used.

So, without further ado, here are the objects:

Business Object

This is *your* Business Object, so it's not actually part of the framework. You will have to add a few methods to your Business Object in order to plug it in to the framework, but, in theory at least, you shouldn't have to change anything else about your existing object.

In my case these Business Objects would be generated by Transfer based on a decorator. As all of my decorators are based on my AbstractTransferDecorator, I only have to add the methods required for integration with the framework into my abstract decorator.

In terms of sample code, let's look at the save() and validate() methods in my AbstractTransferDecorator to demonstrate how it is integrated with the rest of the framework:

```
<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 getTransfer().save(this) />
 </cfif>
</cffunction>

<cffunction name="validate" access="public" returntype="void">
 <cfargument name="Result" type="any" required="true" />
 <cfargument name="Context" type="any" required="true" />
 <cfset getServerValidator().validate(this,arguments.Context,arguments.Result) />
</cffunction>
```

I would be getting ahead of myself if I were to start describing Result and ServerValidator, so let's just leave those for now, but you may want to refer back to this code a bit later.

Core Objects

Validation Factory

This object is a singleton that is composed into your Business Object when it is created. I am using **Brian Kotek**'s excellent **Bean Injector** to do this automatically. The Validation Factory's sole purpose is to create Business Object Validators (see below).

BO Validator

This stands for Business Object Validator, and this object is a singleton that is composed into your Business Object by the Validation Factory after your Business Object has been created. Its purpose is to provide your Business Object with information about the validation rules that have been defined for the object.

It can obtain those validation rules by reading an XML file and also can be told the rules by calling methods on it (for anyone who prefers to write ColdFusion code over XML files).

So, to get the validation rules for a Business Object you simply ask the composed BOValidator for those rules.

In terms of sample code, let's look at the setUp() method in my AbstractTransferDecorator to see how a configured BOValidator is composed into the Business Object. The setUp() method is run automatically by the Bean Injector after all dependencies have been injected into the Business Object.

```
<cffunction name="setUp" access="public" returntype="void" >
 <cfset setValidator(getValidationFactory().getValidator(getClassName())) />
</cffunction>
```

All that's happening here is that the composed ValidationFactory is being asked to create a BOValidator that contains rules for the current Business Object type (getClassName()), and that BOValidator is then being composed into the Business Object.

We can also look at the getValidations() method of the AbstractTransferDecorator to see how we get the validation rules when we need them:

```
<cffunction name="getValidations" access="Public" returntype="any">
 <cfargument name="Context" type="any" required="false" default="" />
 <cfreturn getValidator().getValidations(arguments.Context) />
</cffunction>
```

We're simply asking the composed BOValidator to return the validation rules to us. Context is used to allow for different validation rules in different contexts. For example, the validation rules for a User object might be different if the User is being registered vs. if the User is being updated.

Server Objects

Server Validator

This object is a singleton that is composed into your Business Object when it is created. It is responsible for performing server-side validations. It uses the rest of the Server Objects in the diagram to accomplish that feat.

To perform the server-side validations on a Business Object you simply ask the composed ServerValidator to validate the object, passing the Business Object in as a parameter, which we saw in a code sample above. The ServerValidator then asks the Business Object for the validation rules, and proceeds to perform validations for each of those validation rules. That takes us to our next object.

Validation

The Validation object is one of two transient objects in the framework. As the ServerValidator is going through the list of validation rules, it creates a new Validation object for each rule. In fact, it doesn't literally do that, as that might not be very performant. Instead it creates one Validation object at the beginning and then just reuses that Validation object for each validation rule.

The Validation object, when created, is passed the Business Object, so the Business Object ends up being composed into the validation Object. The Validation Object also contains all of the metadata about the specific validation rule being processed (e.g., ValidationType, PropertyName, CustomFailureMessage, etc.). It also has the ability to record whether the particular validation was successful, and if it is unsuccessful, can record a failure message.

The Validation object is then passed to the appropriate Server Rule Validator object.

Let's look at a snippet from the ServerValidator's validate() method to illustrate this flow:

```
<cfset var Validations = arguments.theObject.getValidations(arguments.Context) />

<cfset var theVal = getTransientFactory().newValidation(arguments.theObject) />

... snip ...

<cfloop Array="#Validations#" index="v">

 <cfset theVal.load(v) />

 <cfset variables.RuleValidators[v.ValType].validate(theVal) />

 <cfif NOT theVal.getIsSuccess()>

  <cfset arguments.Result.setIsSuccess(false) />

  ... snip ...

 </cfif>

... snip ...

</cfloop>
```

There's a whole lot more going on in that method than just that, but hopefully it illustrates the flow:

1. The validation rules are obtained from the Business Object.
2. A Validation object is created, passing in the Business Object.
3. The validation rules are looped through and for each rule:
    1. The Validation object is loaded with the metadata of the validation rule.
    2. The validate() method is called on the appropriate composed ServerRuleValidator object, passing in the Validation object.
    3. The Validation object is asked whether the validation was successful or not.
4. etc.

Server Rule Validator

ServerRuleValidator is an abstract object, which forms the base of any number of concrete ServerRuleValidator objects. One such object exists for each validation type (e.g., Required, Email, MinLength, Custom, etc.). Each of these concrete ServerRuleValidators are composed into the ServerValidator, so when the ServerValidator wants to perform the validation for a specific validation rule, it simply asks the appropriate composed ServerRuleValidator to do it. The implementation of the actual validation rules, therefore, exists only in the concrete ServerRuleValidators.

The Validation object itself is the only argument passed to the ServerRuleValidator. The ServerRuleValidator then performs the validation and asks the Validation object to record whether the validation passed or not. If the validation failed, it also asks the Validation object to record a failure message specific to the context of the validation.

The Validation object is then passed back (not literally) to the ServerValidator, which asks it whether the validation passed or not. If the validation failed, the ServerValidator then asks the Result object to record that information.

The code snippet above illustrates the flow of this process.

Result

The Result object is the other transient object in the framework. It is actually created in the Service Layer and is passed into the Business Object when validations need to be performed (e.g., on a Save operation). The Business Object in turn passes it to the ServerValidator. If a server-side validation fails, the Result Object is asked to record information about the failure, which then becomes available to other objects up the chain, and can eventually be used by a view to display validation failures.

Again, this flow should be evident from the code samples above.

Whew!

This is probably the most difficult article I've written thus far. I am trying to explain a fairly complex design, and I'm so close to it that I may not have done a very good job. If you've made it this far, and are totally, or even somewhat, confused, please leave me a comment letting me know if you have any questions or any suggestions about areas that could use greater clarification, and I will endeavour to make this easier to understand.

I imagine that for some this won't become clear until they see more of the actual code, so I will try to post some of that soon. Again, I ask that if you are interested and there is a particular object that you'd like to see or that you find confusing, just let me know and I'll make it a priority to post about those objects first.