ValidateThis! - Client Side Validation Architecture

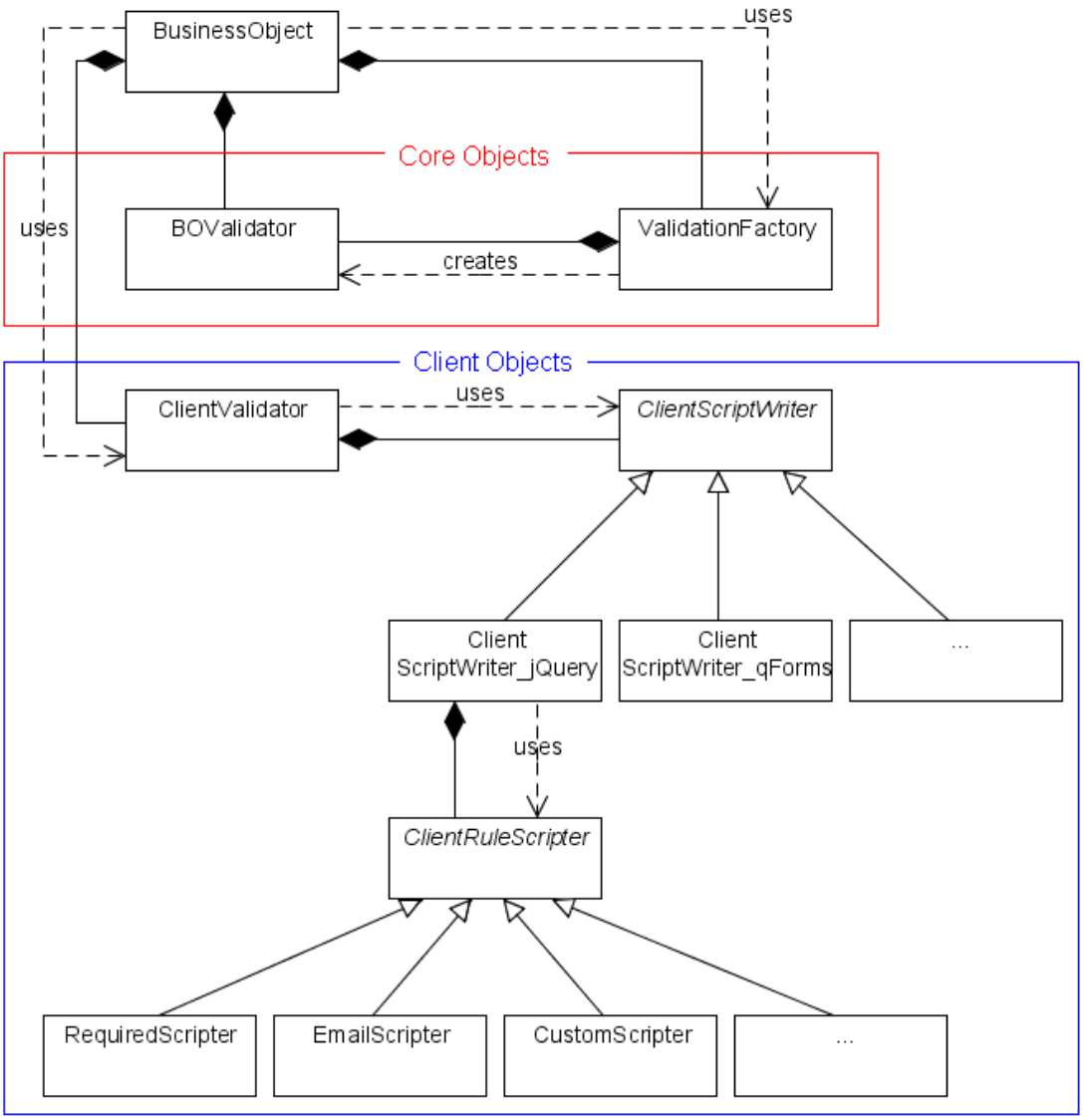
Posted At : October 21, 2008 8:56 AM | Posted By : Bob Silverberg
Related Categories: OO Design, ColdFusion, ValidateThis

In this installment of my series about object oriented validations with ColdFusion I'm going to finish the discussion of the architecture of the framework. In a previous article I discussed that 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.

That article described the Core Objects and the Server Objects, so that just leaves the Client Objects.

Again, let's start with a picture. The picture below has been shrunk to fit on the blog page, to see a larger version just click on the picture itself.



You can refer to this post for a description of the Business Object, Validation Factory and BO Validator. Let's go through the rest of the objects.

Client Validator

This object is a singleton that is composed into your Business Object when it is created. It is responsible for generating client-side JavaScript code which a developer can then include in an html page however they choose. To request the generated JavaScript from a Business Object you simply ask the composed ClientValidator for the script, for example, you might have the following method in your Business Object:

```
<cffunction name="getValidationScript" returntype="any" access="public">
 <cfargument name="Context" type="any" required="false" default="" />
 <cfargument name="JSLib" type="any" required="false"
  default="jQuery" />

 <cfreturn getClientValidator().getValidationScript(getValidations(arguments.Context),arguments.JSLib) />
</cffunction>
```

We optionally pass two arguments into the method in the Business Object:

- *Context*, which describes the context for the validations, e.g., a User is being registered, a User is being updated, etc.
- *JSLib*, which will tell the ClientValidator what type of JS validations to script. The framework currently has a translation layer for the jQuery Validation plugin, but a developer can create a new layer for any JS implementation.

We then simply call the getValidationScript() method on the composed ClientValidator, passing in the actual validation rules (acquired by calling getValidations(arguments.Context)) and the JSLib.

In terms of the code a developer has to write to make use of the built-in JS validation layer, you've just seen it. Add that one method to a Business Object, then call that method from a Service or Controller (or wherever you choose) and voila, you have your JS code, ready to insert into your document. The ClientValidator uses the rest of the Client Objects in the diagram to actually generate the JS, so let's move on to those.

Client Script Writer

ClientScriptWriter is an abstract object, which forms the base of any number of concrete ClientScriptWriter objects. One such object exists for each JS translation layer (e.g., jQuery, qForms, myCustomJSValidationLibrary, etc.). A JS translation layer is able to take the validation rules, as defined in the xml document, and translate them into JavaScript code which can then be inserted into a document. Each of these concrete ClientScriptWriters are composed into the ClientValidator, so when the ClientValidator needs to generate some JS code, it simply asks the appropriate composed ClientScriptWriter to do it.

Client Rule Scripter

ClientRuleScripter is an abstract object, which forms the base of any number of concrete ClientRuleScripter objects. One such object exists for each validation type (e.g., Required, Email, MinLength, Custom, etc.). Each of these concrete ClientRuleScripters are composed into their corresponding concrete ClientScriptWriter object, so when the ClientScriptWriter needs to generate the JS code for a specific rule, it simply asks the appropriate composed ClientRuleScripter to do it.

To see how these three object types (ClientValidator, concrete ClientScriptWriter and concrete ClientRuleScripter) interact, let's first take a look at the only method in the ClientValidator object:

```
<cffunction name="getValidationScript" returntype="any" access="public">
 <cfargument name="Validations" type="any" required="true" />
 <cfargument name="JSLib" type="any" required="false"
  default="jQuery" />

 <cfset var validation = "" />
 <cfset var theScript = "" />
 <cfset var theScriptWriter =
  variables.ScriptWriters[arguments.JSLib] />
 <cfsetting enableCFoutputOnly = "true">

 <cfif IsArray(arguments.Validations) and ArrayLen(arguments.Validations)>
  <cfsavecontent variable="theScript">
   <cfoutput>
    #Trim(theScriptWriter.generateScriptHeader())#
   </cfoutput>
   <cfloop Array="#arguments.Validations#" index="validation">
    <cfoutput>
     #Trim(theScriptWriter.generateValidationScript(validation))#
    </cfoutput>
   </cfloop>
   <cfoutput>
    #Trim(theScriptWriter.generateScriptFooter())#
   </cfoutput>
  </cfsavecontent>
 </cfif>
 <cfsetting enableCFoutputOnly = "false">
 <cfreturn "<script type=""text/javascript"">" &
  theScript & "</script>" />

</cffunction>
```

As mentioned earlier, we pass in the Validations array as well as the name of the JSLib that should be used to generate the JS statements. We then get the appropriate composed ClientScriptWriter (*theScriptWriter*) and ask it to generate our JS for us. You'll notice that the ClientScriptWriter has three methods to aid in this generation:

- *generateScriptHeader()*, which generates any JS statements that need to appear at the top of the JS block.
- *generateScriptFooter()*, which generates any JS statements that need to appear at the bottom of the JS block.
- *generateValidationScript()*, which generates the JS necessary to implement a single validation rule.

That last method, *generateValidationScript()*, is called for each validation rule found in the Validations array that was passed into the method.

Finally, we wrap the entire JS block in a script tag, so it's ready to be inserted into our document.

The last pieces of the puzzle are the concrete ClientRuleScripter objects. Let's take a look at the *generateValidationScript()* method in a concrete ClientScriptWriter to see how it interacts with the concrete ClientRuleScripter objects:

```
<cffunction name="generateValidationScript" returntype="any"
```

```
 access="public">

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


 <cfset var theScript = "" />


 <cfif arguments.validation.ValType EQ "required"

  OR NOT

  (StructKeyExists(arguments.validation.Parameters,"ClientCondition")

  OR

  StructKeyExists(arguments.validation.Parameters,"DependentPropertyName"))

  >

  <cfset theScript =

   variables.RuleScripters[arguments.validation.ValType].generateValidationScript(arguments.validation) />

 </cfif>


 <cfreturn theScript />


</cffunction>
```

For the most part all we're doing here is asking the appropriate composed ClientRuleScripter object to generate the JS statements for the validation rule struct that was passed in as an argument. The reason for the extra code in the cfif is that, with this particular JS library implementation, we can only generate conditional validations for *required* rule types. On the server side it's possible to have any validation type be conditional. Realistically I cannot think of many use cases for conditions on validation types other than *required*, but it is possible on the server-side.

The *generateValidationScript()* method inside a concrete ClientRuleScripter object simply returns the JS statements necessary to implement that rule. I am using yet another abstract object, an abstract ClientRuleScripter, which forms the base of each concrete ClientRuleScripter object, but the only reason I'm doing that is to eliminate duplication of code when generating the JS validations. I left it off the diagram because I figured it was already crowded enough, and it doesn't really have anything to do with the framework itself.

A Fork in the Road

Whew, that finishes my explanation of the design of this object oriented validation framework. I think that the code base is just about ready to be shared with interested parties. I realize at this point that I need to make a decision about what to write about in future blog posts. Obviously I still have a lot of material that I can cover, all of which I do intend to cover over time. I think that this material can be divided into two categories:

1. How I've implemented the design. That is, the actual code that I wrote, along with an explanation of that code.
2. How to implement the framework in an existing model. That is, what would a developer do to integrate the framework with an existing model.

I'd like to throw this question out there for anyone that is interested. Which would *you* prefer to see? If you care either way, please either leave a comment or feel free to email me directly.