Performing Server Side Validations Using Objects

Posted At : October 29, 2008 11:53 AM | Posted By : Bob Silverberg
Related Categories: OO Design, ColdFusion, ValidateThis

I'm going to continue my series about object oriented validations with ColdFusion by looking at the approach that I've taken to performing server side validations. I discussed the architecture for server side validations in a **previous article**, so now I want to get down to the nitty gritty of how the actual validations are performed, looking at the code involved.

I'm going to do this in the context of discussing how I added generic regex support to the framework. I want to thank **Matt Quackenbush**, my regex mentor, for helping me with the required syntax, and with an example for the **demo application**.

Because one of the **design goals** I had for the framework was the ability to add new validation types without having to touch any of the existing code, adding regex support was a piece of cake. Here's how I wanted this new *regex* validation type to work:

- A developer can create a validation rule for an object property of type *regex*.
- The developer can then either:
  - Specify a single *regex* parameter, which will be used on both the client and the server.
  - Specify both a *clientregex* and a *serverregex* parameter, which will be used accordingly. This will allow a developer to take advantage of ColdFusion regex syntax that would not be valid in JavaScript.
- When processing validations, either on the client or the server, the contents of the specified property will be tested against the specified regex, and if no match is found the validation will fail.

In order to implement this new validation type I had to create two new files, a Server Rule Validator and a **Client Rule Scripter**. Because the focus of this post is server side validations I'm only going to look at the Server Rule Validator for now.

As discussed in a **previous article**, a Server Rule Validator implements the actual logic for testing a property against a validation rule type. I had already written a number of these (e.g., Required, Email, RangeLength, etc.), so I just took an existing one and made a few edits. Here's what ServerRuleValidator_Regex.cfc looks like:

```
<cfcomponent output="false" name="ServerRuleValidator_Regex"

 extends="AbstractServerRuleValidator">



 <cffunction name="validate" returntype="any" access="public">

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



  <cfset var Parameters = arguments.valObject.getParameters() />

  <cfset var theRegex = "" />

  <cfset var theValue = arguments.valObject.getObjectValue() />

  <cfif StructKeyExists(Parameters,"serverRegex")>

   <cfset theRegex = Parameters.serverRegex />

  <cfelseif StructKeyExists(Parameters,"regex")>

   <cfset theRegex = Parameters.regex />

  <cfelse>

   <cfthrow errorcode="validatethis.ServerRuleValidator_Regex.missingParameter"

   message="Either a regex or a serverRegex parameter must be defined for a regex rule type." />

  </cfif>

  <cfif Len(theValue) AND REFind(theRegex,theValue) EQ 0>

   <cfset fail(arguments.valObject,"The #arguments.valObject.getPropertyDesc()# must match the specified pattern.") />

  </cfif>

 </cffunction>



</cfcomponent>
```

Let's walk through this code. First off, you'll notice that this cfc extends an AbstractServerRuleValidator, so we'll take a look at that in a moment.

Next, you'll see that the validate method expects a Validation object. I discussed that in an **earlier blog post**, but basically each validation rule that you define becomes an object when the server side validations are done. That object will contain all of the metadata about the validation rule, and will also have the actual Business Object composed into it, which allows for access to that Business Object's properties.

Continuing with the code, we get the *Parameters* struct from the Validation object, which should include either a *regex* key or a *serverRegex* key. If neither of those are found an error is thrown. We also need to get the value of the property for which this rule was defined, which is done by calling the *getObjectValue* method of the Validation object. We'll take a look at that in a moment as well.

Now that we have all of the information we need, the actual test is performed. We check that there is something to validate, which allows for an empty field to pass a regex validation, and if there is a value an REFind is performed to determine whether the contents of the property match the specified regex. **Matt** suggested that this approach be taken, rather than using isValid, based on his experience with isValid which he documented in a **blog post**.

If the validation fails, then the *fail* method is called to register that. We are passing a generic failure message to the *fail* method, but in the context of this validation type (regex) one would usually override that generic failure message with a custom one in the rule definition. As you might guess, the *fail* method is contained in the AbstractServerRuleValidator, so let's look at that next:

```
<cfcomponent output="false" name="AbstractServerRuleValidator">



<cffunction name="init" returnType="any" access="public">

 <cfreturn this />

</cffunction>



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

```
 <cfargument name="valObject" type="any" required="yes" />
 <cfthrow errorcode="validatethis.AbstractServerRuleValidator.methodnotdefined"
  message="I am an abstract object, hence the validate method must be overriden in a concrete object." />
</cffunction>

<cffunction name="fail" returntype="void" access="private">
 <cfargument name="valObject" type="any" required="yes" />
 <cfargument name="FailureMessage" type="any" required="yes" />

 <cfset arguments.valObject.setIsSuccess(false) />
 <cfset arguments.valObject.setFailureMessage(arguments.FailureMessage) />
</cffunction>

</cfcomponent>
```

The *init* and *validate* methods are self-explanatory, so let's just look at the *fail* method. Because the behaviour that occurs when a validation fails is always the same, I have encapsulated this logic into the abstract object which forms the base for all of the concrete Server Rule Validators. When a validation fails the *IsSuccess* property of the Validation object is set to *false* and the failure message generated by the *validate* method is placed in the *FailureMessage* property. Now the Validation object knows everything that it needs to know about the result of the validation attempt.

The next piece of the puzzle is the Validation object itself. It contains three methods that "do something", and a whole load of getters and setters. We'll just look at the former:

```
<cffunction name="init" access="Public" returntype="any">
 <cfargument name="theObject" type="any" required="yes" />
 <cfset variables.theObject = arguments.theObject />
 <cfreturn this />
</cffunction>

<cffunction name="load" access="Public" returntype="any">

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

 <cfset variables.instance = Duplicate(arguments.ValStruct) />
 <cfset variables.instance.IsSuccess = true />
 <cfset variables.instance.FailureMessage = "" />
 <cfreturn this />

</cffunction>

<cffunction name="getObjectValue" access="public" returntype="any">
 <cfset var theValue = "" />
 <cfset var propertyName = getPropertyName() />
 <cfif StructKeyExists(variables.theObject,"get#propertyName#")>
  <cftry>
   <cfinvoke component="#variables.theObject#"
    method="get#propertyName#" returnvariable="theValue" />
   <cfcatch type="any"></cfcatch>
  </cftry>
  <cfreturn theValue />
 <cfelse>
  <cfthrow errorcode=errorcode="validatethis.propertyNotFound"
   message="The property #propertyName# was not found in the object." />
 </cfif>
</cffunction>
```

For each validation rule that must be processed, the Server Validator (which I'll discuss in a moment) creates a Validation Object. To limit the number of actual objects that needs to be created, the Server Validator actually only creates one Validation object, and passes the Business Object itself into the *init* method. This gives the Validation Object access to the Business Object's properties. As the Server Validator processes each subsequent validation rule, it loads each one into the existing Validation Object, which is what the *load* method is for.

The *getObjectValue* method is a shortcut that allows the ValidationObject to get the contents of the property that corresponds to the current validation rule. First it gets the propertyName from the metadata that was loaded into the Validation object. It then checks to make sure that a method exists in the Business Object that will return the contents of that property. The attempt to retrieve that value using that method is wrapped in a try/catch because it is possible that the method could throw an error, for example if one is asking for a composed object from a Transfer decorator but the composed object hasn't been loaded yet. If the cfinvoke fails, an empty string is returned, otherwise the value of the property is returned. If no method can be found that corresponds to the property then an error is thrown. You

can see how the *getObjectValue* method is used in the code for the ServerRuleValidator_Regex object above.

The final piece of the server side validation puzzle is the Server Validator itself. Its *validate* method does a whole bunch of things, including dealing with conditional validations, custom failure messages, and packaging up the final result. I think this post is already long enough, so I'm going to leave a discussion of the internals of that method to a future post. If it isn't already clear from what I've written above, the Server Validator basically does this:

1. Asks the Business Object for its validation rules.
2. Loops through those rules, conditionally creating a Validation object for each rule.
3. Asks an appropriate Server Rule Validator (such as the one documented above) to perform a validation using the Validation object.
4. Asks the Validation object whether the validation passed, and if not, what default message to use.
5. Packages the results (pass or fail, plus any failure messages) and makes them available to the object that called it.

Getting back to the point of this post, I hope that I have demonstrated two things:

1. To add a new validation type I only had to create one file (for the server side), which involved writing around half a dozen lines of code, and most of that code dealt with parameters specific to this validation type.
2. The framework code itself is actually quite simple and there's not that much of it.

I will continue with the discussion of the Server validator, and also discuss the other file (Client Rule Scripter) in future posts.