Getting Started with VT and Transfer - Part IV - Using VT for Server-Side Validations

Posted At : March 21, 2009 11:02 AM | Posted By : Bob Silverberg
Related Categories: ColdFusion, ValidateThis

Update: The information in this post is no longer correct due to changes to the framework. A new series is available via the Getting Started with VT category of my blog.

Original content of the article follows:

In the previous post in this series about getting started with Transfer and ValidateThis!, my validation framework for ColdFusion objects, we looked at the few changes that we had to make to the initial sample application in order to integrate VT into our Business Object, so now we're ready to start using VT to perform our server-side validations. To do that we're going make the following changes to the sample application:

- Add an xml file that defines our validation rules for our User object.
- Modify the updateUser() method in the UserService to tell our User object to validate itself.
- Modify our form to display any validation failure messages.

For now we're just going to add the validations for the UserName property, which appears on the form as the Email Address field. We're also only going to utilize server-side validations for now. A version of the sample application with those validations in place can be viewed here.

Let's get started by creating the xml file that will define our validation rules. The file will have the same name as our Transfer class, so in this example the file will be called user.user.xml, and will contain the following:

```
<?xml version="1.0" encoding="UTF-8"?>

<validateThis xsi:noNamespaceSchemaLocation="validateThis.xsd"

 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

 <objectProperties>

  <property name="UserName" desc="Email Address">

   <rule type="required" />

   <rule type="email"

    failureMessage="Hey, buddy, you call that an Email Address?" />

  </property>

 </objectProperties>

</validateThis>
```

I don't want to spend a lot of time in this post describing the format of the ValidateThis! xml schema. I described it at length in a couple of previous posts. In a nutshell, what we've done above is to declare that our Business Object will have a *UserName* property, and that that property has two validation rules associated with it:

1. It is required.
2. It must be a valid email address.

In addition to that we've declared that the "friendly name" of the property (to be used in validation failure messages), is *Email Address* and that the custom message that we'd like displayed when the email validation fails is "Hey, buddy, you call that an Email Address?".

Now that we've defined our validation rules, we can move on to making the required change to the UserService Object. Here's what the original version of the updateUser() method looks like:

```
<cffunction name="updateUser" access="public" returntype="any">

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

 <cfargument name="args" type="struct" required="yes" />



 <cfset var objUser = getUser(arguments.theId) />

 <cfset objUser.setUserName(arguments.args.UserName) />

 <cfset objUser.setUserPass(arguments.args.UserPass) />

 <cfset objUser.setNickname(arguments.args.Nickname) />

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

 <cfreturn objUser />



</cffunction>
```

And here's the new version:

```
<cffunction name="updateUser" access="public" returntype="any">

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

 <cfargument name="args" type="struct" required="yes" />



 <cfset var objResult = "" />

 <cfset var objUser = getUser(arguments.theId) />

 <cfset objUser.setUserName(arguments.args.UserName) />

 <cfset objUser.setUserPass(arguments.args.UserPass) />

 <cfset objUser.setNickname(arguments.args.Nickname) />

 <cfset objResult = objUser.validate() />
```

```
 <cfif objResult.getIsSuccess()>

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

 </cfif>

 <cfreturn objResult />

</cffunction>
```

So what's changed? After we populate the User object with data submitted via the form, we call the User object's validate() method. This method is actually part of the VT framework, but the integration that we did in the last post magically makes this method (and several others) available to our User object. The validate() method returns a Result object (also part of the framework) with which we can interact.

The Result object can tell us whether or not the validations passed, which we do by checking its getIsSuccess() method. If that returns "true" then we go ahead and save our object.

Finally, instead of returning the User object to the calling routine, we return the Result object, which will contain information about any validation failures that occurred and will also contain a copy of the User object.

To sum it up:

1. We call validate() on our Business Object, which returns a Result object.
2. We check to see if the validations passed, and act accordingly.
3. We return the Result object to the calling routine.

And that's it - our server-side validations will be performed and the results of those tests will be returned to us in the Result object.

The only change left to make to our sample application is to make use of that Result object in our view. Here's the new code for the form, with all changes preceded by comments:

```
<!--- Default the validation failures to an empty struct --->

<cfset validationFailures = {} />



<cfset UserService = application.BeanFactory.getBean("UserService") />



<cfif NOT StructKeyExists(Form,"Processing")>

 <cfset UserTO = UserService.getUser(theId=0) />

<cfelse>

 <!---  The updateUser() method returns a Result object--->

 <cfset Result = UserService.updateUser(theId=0,args=Form) />

 <!--- Get the validation failures from the Result object --->

 <cfset validationFailures = Result.getFailuresAsStruct() />

 <!--- Get the User Business Object from the Result object --->

 <cfset UserTO = Result.getTheObject() />

</cfif>



<cfoutput>

<h1>

 ValidateThis! and Transfer Sample App - Part II

  - Some Server-Side Validations

</h1>

<cfif UserTO.getUserId() NEQ 0>

 <h3>The user has been added!</h3>

</cfif>

<div class="formContainer">

<form action="index.cfm" id="frmMain" method="post" name="frmMain" class="uniForm">

 <input type="hidden" name="Processing" id="Processing" value="true" />

 <fieldset class="inlineLabels">

  <legend>User Information</legend>

  <div class="ctrlHolder">

   <!--- Output any validation failure messages --->

   <cfif StructKeyExists(validationFailures,"UserName")>

    <p id="error-UserName" class="errorField bold">

     #validationFailures["UserName"]#

    </p>

   </cfif>

   <label for="UserName">Email Address</label>

   <input name="UserName" id="UserName"

    value="#UserTO.getUserName()#"

    size="35" maxlength="50" type="text"

    class="textInput" />
```

```
    <p class="formHint">
     Validations: Required, Must be a valid Email Address.
    </p>
   </div>
   <div class="ctrlHolder">
    <label for="UserPass">Password</label>
    <input name="UserPass" id="UserPass" value=""
     size="35" maxlength="50" type="password"
     class="textInput" />
    <p class="formHint">
     Validations: Required, Must be between 5 and 10 characters.
    </p>
   </div>
   <div class="ctrlHolder">
    <label for="Nickname">Nickname</label>
    <input name="Nickname" id="Nickname"
     value="#UserTO.getNickname()#"
     size="35" maxlength="50" type="text"
     class="textInput" />
    <p class="formHint">
     Validations: Custom - must be unique. Try 'BobRules'.
    </p>
   </div>
  </fieldset>

  <div class="buttonHolder">
   <button type="submit" class="submitButton">Submit</button>
  </div>
 </form>
</div>
</cfoutput>
```

To summarize the changes made to the view to make use of the Result object:

1. We create a default empty struct to hold our validation failure messages, because we are going to be using that struct later.
2. Instead of simply expecting a User object back from our updateUser() method we now expect a Result object.
3. We ask the Result object for our validation failure messages in a format that we can use in our form.
4. We ask the Result object for our User object, so we can use it in the form.
5. When rendering the form field for UserName, we output any validation failure messages that were returned by the Result object.

And that's it. Our application is now using the validation rules that we specified in our xml file. Obviously the code in the form to display the validation failures is less than ideal, but this is meant to be a simple example. The thing to keep in mind is that the framework will return metadata about validation failures to you, via the Result object, and you can use that metadata any way you like.

In the next post I'll discuss what needs to be done to enable client-side validations for the rules that we defined. A hint: all we have to do is add a few lines of code to our form!

You can see a demo of this sample application in action at **www.validatethis.org/VTAndTransfer_PartII/**. As always all of the code for this version of the sample application can be found attached to this post.