Getting Started with VT and Transfer - Part II - The Initial Sample App

Posted At : March 12, 2009 1:53 PM | 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 frameworks required and the configuration of those frameworks. This post will cover the remainder of the initial sample app, so we have a starting point for integration.

First off, you can view an online version of the sample application at www.validatethis.org/VTAndTransfer_Start/.

The code that we're going to look at now resides in the following three files:

- **index.cfm**, which drives the app.
- **theForm.cfm**, which is the form that is used to add a new User.
- **UserService.cfc**, which is the User Service Object.

Let's start by looking at index.cfm:

```
<cfsilent>
 <cfif StructKeyExists(url,"init") OR
  NOT StructKeyExists(application,"BeanFactory")>
  <cfset application.BeanFactory = CreateObject("component","coldspring.beans.DefaultXmlBeanFactory").init() />
  <cfset application.BeanFactory.loadBeans(beanDefinitionFileName=expandPath("/model/config/Coldspring.xml.cfm"),constructNonLazyBeans=true) />
 </cfif>
</cfsilent>
<html>
 <head>
  <title>ValidateThis! and Transfer Sample App</title>
  <link href="css/style.css" type="text/css" rel="stylesheet" />
  <link href="css/uni-form-styles.css" type="text/css" rel="stylesheet" />
 </head>
 <body>
 <div id="container">
  <div id="sidebar">
   <cfinclude template="theSidebar.cfm" />
  </div>
  <div id="content">
   <cfinclude template="theForm.cfm" />
  </div>
 </div>
 </body>
</html>
```

Nothing earth shattering here. Basically it just checks to see if the Coldspring bean factory needs to be initialized, and if so does it. Then it includes the cfm files for the Sidebar (which is just text) and the Form. Looking at the code for theForm.cfm:

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

<cfif NOT StructKeyExists(Form,"Processing")>
 <cfset UserTO = UserService.getUser(theId=0) />
<cfelse>
 <cfset UserTO = UserService.updateUser(theId=0,args=Form) />
</cfif>

<cfoutput>
<h1>ValidateThis! and Transfer Sample App - Part I - No 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">
   <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>
```

We start off by asking the Coldspring bean factory for our UserService object, which we'll need in order to get a User object to display on the screen, and will also be used to persist a new User object to the database.

This is essentially a self-posting form, so next we check to see if we're displaying the form for the first time, or processing a form post. If the former, we simply ask the UserService for a brand new empty User object, which is done by passing in an Id of 0. If the latter, we ask the UserService to persist a new User object to the database for us, by passing in an Id of 0, and the contents of the Form scope.

The remainder of the code renders our form:

- If a User object was just saved (which means that the UserId property of the object will not be 0), we display a message to that effect.
- We then render each form field, asking the User object for the value of its property for each form field. For example, the value of the UserName field comes from UserTO.getUserName().

Finally, we look at the code for UserService.cfc:

```
<cfcomponent displayname="UserService" output="false">


 <cffunction name="init" access="Public" returntype="any">
  <cfreturn this />
 </cffunction>


 <cffunction name="getUser" access="Public" returntype="any">
  <cfargument name="theId" type="any" required="yes" />
  <cfreturn getTransfer().get("user.user",arguments.theId) />
 </cffunction>
```

```
 <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>

 <cffunction name="getTransfer" access="public" returntype="any">
  <cfreturn variables.Transfer />
 </cffunction>
 <cffunction name="setTransfer" access="public" returntype="void">
  <cfargument name="transfer" type="any" required="true" />
  <cfset variables.Transfer = arguments.Transfer />
 </cffunction>

</cfcomponent>
```

Looking at the bottom of the file first, the setTransfer() method is there so that Coldspring can automatically inject Transfer into the UserService, and the getTransfer() method is there so that we can access Transfer from within the UserService.

The getUser() method simply asks Transfer for a User object that corresponds to the UserId that is passed in. The database is configured so that there will never be a User with a UserId of 0, which is why we can pass 0 into that method and know that we'll get back a new, empty User object.

The updateUser() method calls the getUser() method to get a User object, after which it populates the User object with data from the Form and then asks Transfer to save the User object, which persists the User object to the database.

And that's it. As I mentioned in the previous post, this is **not even close** to the way I actually do things, but it works and it's a very simple example.

As I also mentioned in the previous post, there are no validations in place. We can see from the form that some validation rules exist for the User object, but there is no code in place to enforce those rules. So our next step will be to integrate VT into this simple application and implement the validation rules for the UserName property. That will be covered in my next post.

Again, an online version of this sample application is available at **www.validatethis.org/VTAndTransfer_Start/**, and all of the code for it can be found attached to this post.