Validating Business Objects with Metro

Posted At : February 27, 2009 5:22PM | Posted By : Matt Quackenbush
Related Categories: ColdFusion, Transfer

Over the years it seems that threads on two topics appear over and over again on various mailing lists and forums: code generation and object validation. Paul Marcotte recently released his Metro project which provides an answer to both of these topics. Metro takes advantage of features in Transfer and ColdSpring that are leveraged to help the developer get an application up and running in short order.

In this post, I want to focus on the second topic I mentioned earlier, that being object validation, so let's get to it.

Rich Business Objects

Metro encourages you to write rich business objects by writing decorators [hereinafter referred to as 'class(es)'] that extend the metro.core.Decorator class, which in turn extends Transfer's decorator class.

(For the sake of brevity, I'm not going to go into much detail on the methods of the Metro Decorator, focusing primarily on how to handle validation of your rich business object.)

Example: An 'Address' Class

Our UberShippingApp needs an Address class to keep track of where items should be shipped. First we'll define it in our transfer.xml.

```
<object name= "Address" table= "tbl_Address" decorator= "model.shipping.Address" >
<id name= "ID" column= "addressID" type= "numeric" />
<property name= "LineOne" column= "line_one" type= "string" />
<property name= "Linetwo" column= "line_two" type= "string" />
<property name= "Suite" column= "suite" type= "string" />
<property name= "City" column= "city" type= "string" />
<property name= "PostalCode" column= "postal_code" type= "string" />
<!-- // Many-to-One \ -->
<manytoone name= "Province" lazy= " false " proxied= " false " >
<link to= "Province" column= "province_id" />
</manytoone>
</object>
```

Since this post is not intended to be a tutorial on Transfer, I'm going to skip any explanation of what we just did. If you don't understand it, I would recommend reading through the transfer config file docs .

loadRules()

When Metro grabs a new object from Transfer, it will automatically call the loadRules() method on the new object, which, as the name implies, will load the validation rules that have been declared in the method. So now that we've told Transfer about our Address class, let's add our validation rules to it.

```
<cffunction name= "loadRules"
hint= "Loads the object's validation rules"
returntype= "void"
output= "no"
access= "public" >
```

```
<cfscript>
addRule(
property: "LineOne" ,
label: "Address" ,
testtype: "string" ,
low: 5,
high: 80
);
addRule(
property: "LineTwo" ,
label: "Address (Line 2)" ,
testtype: "string" ,
ignoreOnEmpty: true ,
low: 5,
high: 80
);
addRule(
property: "Suite" ,
testtype: "string" ,
ignoreOnEmpty: true ,
low: 1,
high: 15
);
addRule(
property: "City" ,
testtype: "string" ,
low: 2,
high: 80
);
addRule(
property: "ProvinceId" ,
label: "State/Province" ,
testtype: "method" ,
method: "hasProvince" ,
message: "Please select your State/Province."
);
addRule(
property: "PostalCode" ,
label: "Postal Code" ,
testtype: "string" ,
low: 3,
high: 20
);
</cfscript>
</cffunction>
```

As you can see, we have made six (6) calls to the addRule() method (more on this method in a minute). Let's go over each rule in detail.

The first rule we've declared applies to the 'LineOne' property of our Address class. This rule tells the Validator that this property must meet the following criteria:

1. It must be a string
2. It must be at least 5 characters in length
3. It must be a maximum of 80 characters in length

The 'label' argument provided tells the Validator to use 'Address' in any error messages instead of using the property name ('LineOne'), which is the default.

Our next rule applies to the 'LineTwo' property. The LineTwo rule is identical to the LineOne rule, except for one very important item: ignoreOnEmpty. By setting this argument to true, we're telling the Validator that if the property has no value, ignore the rule. Otherwise, enforce it normally.

> "But why would I want to declare a rule just to have the Validator to ignore it?"

Some properties are optional. If such properties have not been provided, there's nothing to enforce. However, if the user supplied a response to the property, it still needs to be validated.

The rules for the 'Suite', 'City' and 'PostalCode' properties are pretty much identical to the ones we've just covered, so I'm going to skip over them.

That brings us to the 'ProvinceId' rule, which introduces another testtype that is available in Metro: method. Here's the plain English translation of what this rule tells the Validator:

1. Run the 'hasProvince()' method on the object
2. If it returns false, use the message "Please select your State/Province" instead of the default message

> "What the hell is the 'ProvinceId' property? I don't see it in the transfer.xml? And while we're at it, what the hell is the 'hasProvince()' method?"

When building rich business objects, you should build objects that actually *do something* . In other words, you should be modeling behavior, not data. If all your object does is hold data, then you've built nothing more than a glorified struct. This is commonly referred to as the Anemic Domain Model .

The 'ProvinceId' is an over-simplified example of a business object that *does something* . Let's take a look at a snippet from our address form...

```
<uform:field label= "State/Province"
name= "provinceId"
type= "select"
isRequired= " true " >
<uform:option display= "Select State/Province" value= "0" />
<cfloop query= "qProvinces" >
<uform:option display= "#qProvinces.provinceName[currentRow]#"
value= "#qProvinces.id[currentRow]#"
isSelected= "#address.getProvinceId() EQ qProvinces.id[currentRow]#" />
</cfloop>
</uform:field>
```

This little snippet simply creates a select box named 'ProvinceId' that uses a query to populate the options. (If you have questions about what exactly those funky <uform:foo /> tags are doing, they can be answered by checking out cfUniForm .)

So, when the form is submitted, it will include a field named 'ProvinceId', which our Address class will use to compose the correct Provice class. We'll add get/setProvinceId() methods to our class that will handle this behavior for us.

In our transfer.xml, we set a Many-To-One relationship from our Address class to the Province class. Doing this means that Transfer automagically adds a getProvince(), setProvince(), and hasProvince() method to the generated TransferObject. Now we can build our ProvinceId-related behavior around those generated methods .

getProvinceId()

```
<cffunction name= "getProvinceId"
hint= "Retrieves the provinceId property"
returntype= "numeric"
output= " false " access= "public" >
<cfscript>
if ( hasProvince() ) {
return getProvince().getId();
} else {
return 0;
}
</cfscript>
</cffunction>
```

All we're doing in the getProvinceId() method is checking for the existence of a composed Province object. If it's there, we grab it, and return its ID. If it is not there, we return 0.

setProvinceId()

```
<cffunction name= "setProvinceId"
hint= "Sets the provinceId property"
returntype= "void"
output= " false " access= "public" >
<cfargument name= "provinceId"
hint= "The provinceId property"
required= "yes" type= "numeric" />
<cfscript>
if ( (provinceId > 0) &&
(getService().isProvince(id: provinceId)) ) {
setProvince(getService().getProvince(id: provinceId));
}
</cfscript>
</cffunction>
```

In setProvinceId() we check to make sure that the provided ID is greater than 0, and then we pass it off to our service object* to make sure that it matches a Province. If it does, we compose the matching Province object into the Address object.

With these two methods in place, when the form is submitted with a selected ProvinceId, the Address class intelligently creates the correct composition, and the 'ProvinceId' rule that we discussed earlier can take advantage of the generated hasProvince() method to assist us in making sure that our Address object is in a valid state.

*It should be noted that Metro does not automatically inject the service object into the object. To do so, you will need to add get/set{FooService}() methods on your decorator. In the example we used getService(), but that could be getContactService() or getGeoService() or getAddressService() or getShippingService(), depending upon how you have modeled your application. Whichever service object returns a Province object, that is the one we want to compose into our user object.

The addRule() Method

The addRule() method has a number of arguments that can be used in a variety of combinations to provide a very rich set of validation rules. Here are the details on the method and its various arguments.

property:
    "string"
    required
    the name of the property the rule applies to
testtype:
    "string"
    required
    the type of test this rule should perform
    any value supported by CF's isValid() function's type parameter, plus:
    'alpha', 'alphanum', 'inList', 'inListNoCase', 'notInList', 'notInListNoCase', 'isNot', 'isNotNoCase', 'isMatch', 'isMatchNoCase', 'method', 'daterange', 'required'
label:
    "string"
    optional
    the property idenitifier text to display in error messages; defaults to the value of the property parameter
ignoreOnEmpty:
    true|false
    optional
    defaults to false
    if set to true and the property has no value, the rule will not be enforced
    if set to false, the rule will be enforced regardless of whether the property has a value set or not
contexts:
    "string"
    optional
    comma-delimited list of contexts the rule will be applied to
    defaults to all
message:
    "string"
    optional
    the message to display if the rule test fails validation
    if not provided, a very generic message will be generated (e.g. '{Property Name} failed validation.')
low:
    numeric|date
    ignored, unless testtype is 'range', 'date', 'string', or 'regex'; required for 'range', optional for 'string' or 'regex'
    if testtype='range', this represents the lowest valid value
    if testtype='string' or testtype='regex', this represents the lowest valid character count
    if testtype='date', this represents the earliest valid date
high:
    numeric|date
    ignored, unless testtype is 'range', 'date', 'string', or 'regex'; required for 'range', optional for 'string' or 'regex'
    if testtype='range', this represents the highest valid value
    if testtype='string' or testtype='regex', this represents the highest valid character count
    if testtype='date', this represents the latest valid date
compareProperty:

"string"
ignored, unless testtype is 'isMatch', 'isMatchNoCase', 'isNot', or 'isNotNoCase', then optional
name of the property whose value this rule's property value should be compared to
if testtype is 'isMatch', 'isMatchNoCase', 'isNot', or 'isNotNoCase', either compareProperty OR compareValue MUST be supplied

compareValue:
"string"
ignored, unless testtype is 'isMatch', 'isMatchNoCase', 'isNot', or 'isNotNoCase', then optional
value this rule's property value should be compared to
if testtype is 'isMatch', 'isMatchNoCase', 'isNot', or 'isNotNoCase', either compareProperty OR compareValue MUST be supplied

list:
"string"
ignored, unless testtype is 'inList', 'inListNoCase', 'notInList' or 'notInListNoCase', then required
if testtype='inList', provides a comma-delimited list of valid values that the property value must be one of
if testtype='notInList', provides a comma-delimited list of invalid values that the property value may not match
note that testtype='inList' can easily be testtype='regex' instead, e.g.
testtype='regex', pattern='A(pipe|delimited|list|of|valid|values)'
one should favor the regex method over the list method, since regex is more efficient

allowSpaces:
true|false
ignored, unless testtype is 'alpha' or 'alphanum', then optional

pattern:
"string"
ignored, unless testtype='regex', then required
a regular expression that the value must match
NOTE: this regex will be tested with reFind(), giving the developer full flexibility in defining the pattern

method:
"string"
ignored, unless testtype='method', then required
the name of the method to execute to perform the test
the method must return a boolean, otherwise an exception will be thrown

isDependent:
true|false
optional
defaults to false

dependency:
"string"
ignored, unless isDependent=true, then required
name of property that this rule is dependent upon; if the property named in this parameter has a value set, then this rule will be invoked
if no value is set on the property named, then this rule will be ignored

dependencyValue:
"string"
ignored, unless isDependent=true, then optional
value that the dependency property must match for this rule to be invoked