THOR – a different kind of CFC Validator

Author: John Mason, mason@fusionlink.com
Blog: codfusion.com

Thor is a ColdFusion Component (CFC) based data validator. If you are looking into ColdFusion based validator solutions, you have probably found quite a few. Thor offers several unique features that differentiate it from other validators.

- Explicit / Implicit Validations
- Global Validation
- Handle Server and client side validation with one set of validator libraries
- Specific and general error codes and flexibility to do internationalization
- Field name mapping
- Validator/mapping/error code importing by CFC or XML
- Validator list reporting
- Fields not required option
- On-the-fly validation
- Logging

Validation is a critical function of any application, but it also can be difficult to manage and maintain. Thor was developed to make the process of data validation easier. It is very easy to use and incorporate in your existing projects. Thor is framework agnostic and is a free and open source project under a MIT license. There is no underlining database system; to use Thor, you just need the core CFC and some library files. You can download and find updates to Thor from codfusion.com or Thor.portcullis.com.

Server versions tested and supported:
ColdFusion 9
ColdFusion 8
ColdFusion 7

jQuery is optional if you need Thor to also handle client-side validation.

I have not tested on BlueDragon or Railo but it should work. Feel free to contact me if you have problems on those platforms.

Getting Started

You can download the latest version of THOR from either of these sites:
- labs.fusionlink.com
- www.codfusion.com
- thor.riaforge.org

The zip file will contain:
- this document
- Thor.cfc – the core CFC for Thor
- Library CFC and XML files – validator libraries that can be imported into Thor
- Several example cfm pages on how to use Thor

How to install Thor in your projects

Simply copy the Thor.cfc to a place with your ColdFusion application and then call it up via createObject():

```
<cfif isdefined("application.Thor")>
<cfset application.Thor =
```

```
createObject("component","validation.com.fusionlink.Thor").init()/>
</cfif>
```

This creates an instance of the Thor class as a shared, application-level, singleton object. We are now ready to start loading in our validators to test against.

Note for ColdSpring Users:
Refer to the Coldspring section under the “Loading Validator Libraries”. Example8.cfm also shows how to use Thor with ColdSpring.

Loading Validator Libraries

There are several ways to load validators into Thor. The import() method handles the loading of libraries. Here are your options:

Loading a CFC based library

```
<cfset lib = createObject(
            "component",
            "com.fusionlink.cfValidators").init()/>
<cfset application.Thor.import(lib)/>
```

Loading a XML based library

```
<cffile action="read"
            variable="lib1"
            file="#expandpath("com/fusionlink/lib1.xml")#"/>
<cfset application.Thor.import(lib1)/>
```

or an XML document by directory path

```
<cfset application.Thor.import(expandpath("com/fusionlink/lib1.xml"))/>
```

or an XML document directly via <cfxml../>

```
<cfxml variable="lib1">
<validators>

      <validator name="isColor">
      <regex><![CDATA[^#?([A-F]|[0-9]){3}(([A-F]|[0-9]){3})?$]]></regex>
      <errorCode>Not a valid hexadecimal color code</errorCode>
      </validator>

</validators>
</cfxml>
<cfset application.Thor.import(lib1)/>
```

or with Coldspring

```
<beans>
<bean id="Postalcodes" class="libraries.Postalcodes" />
<bean id="BasicValidators" class="libraries.BasicValidators" />

<bean id="Thor" class="com.fusionlink.Thor">
      <property name="import"><ref bean="Postalcodes" /></property>
      <property name="import"><ref bean="BasicValidators" /></property>
```

```
</bean>
</beans>
```

Thor will check to see if a validator with the same name has previously been loaded. If it has, Thor will not overwrite it. If there are any problems with the import() call, the method will return a boolean “false” response. Typically, problems with import are due to badly formed Xml documents. If the import worked, the method will return a “true” response.

Once you have your validators loaded in, you can have Thor report back to you the list of imported validators.

For example:

```
<cfdump var="#application.Thor.getValidators()#">
```

Running Thor

We now have Thor installed and have loaded in some validator libraries. On an application, there is a form which allows the user to send data via that form into our application. Before we accept this data, we want to run it across a validator to make sure the data is proper and well-formed.

Note on SQL Injection and Cross Site Scripting (XSS) Attacks:
Thor is not by design blocking SQL injection or XSS. It may help just by the way the validators for Thor are used. I do have another project that does help defend against these attack vectors called Portcullis – portcullis.riaforge.org. If you do use Thor and Portcullis together, make the call to Portcullis first. There's no reason to validate data that should simply be blocked.

A simple example to run Thor against form submitted data follows:

```
<cfset request.errors = application.Thor.validate(form)/>
```

Thor will check the form data submitted against the validators. Note: there is an order-of-operation to how Thor validates information.

Explicit Validation

First, Thor will try to validate data against a specific validator. For example, if the user submitted form.email, Thor will look for an “isEmail” validator. If an explicit validator exists, then it will run it and report back if the value is valid or not. The result is always a true/false situation.

Implicit Validation

If Thor can't find an explicit validator, it will then run an implicit validation. As an example, the user submits “form.phone”. There isn't a specific isPhone validator because there are so many different types of phone numbers. It does however, have validators for each different type of phone number. For example, isUSPhone, isUKPhone, isIndiaPhone, etc. An implicit validation will test the data submitted against any validator that contains the object name is the validator name. So “form.phone” will get tested against isUSPhone, isUKPhone, isIndiaPhone, etc. If it passes any one of these validators, it will be considered valid. If it fails all of them, then it will naturally fail.

You can turn off the implicit validation option with the following setting found

in the Thor.cfc:

```
<cfset variables.instance.implicitValidation = true/>
```

Global Validation

There is a special validator method found within Thor called GlobalValidator, which will get tested against all submitted data. By default, it simply checks to be certain the data is within a certain number of characters. You can alter the validator however you like, or turn it off with the following Thor.cfc setting:

```
<cfset variables.instance.useGlobal = true/>
```

Field Name Mapping (FNM)

In some cases, the field name does not properly run a validation test. For example, “form.ccnumber” will not by default run the credit card validator, “isCreditCard”. In these cases, you have to establish a field name map (FNM). The FNMs can be incorporated into Thor in several ways:

Within the Thor CFC

```
<cfset variables.instance.mappings["ccnumber"] = "CreditCard"/>
```

By directory path to an XML document

```
<cfset application.Thor.import(expandpath("com/fusionlink/mappings1.xml"))/>
```

By XML via the <cfxml../>

```
<cfxml variable="map1">
<mappings name="Optional Example Mapping List">

      <mapping name="ccnum">CreditCard</mapping>
      <mapping name="ccnumber">CreditCard</mapping>
      <mapping name="dayphone">Phone</mapping>

</mappings>
</cfxml>
<cfset application.Thor.import(map1)/>
```

Another option is to simply contain the FNMs with the CFC or XML validator libraries. This third option is a good way to organize your mappings with their respective validators. You can look at some of the libraries included with Thor to see how this works.

Required / Non-required Fields

If Thor can run an explicit or implicit validation test against a value, it will require that field by default. You can alter that logic in a number of ways. The first and most obvious alternative is to simply alter the specific validator to allow an empty string to be submitted.

The other option is to tell Thor, when it is called, to NOT require certain fields. For example,

```
<cfset fieldsNotRequired = "email,phone">
<cfset request.errors = application.Thor.validate(form,fieldsNotRequired)/>
```

So even though form.email and form.phone would normally be required by Thor, we can now submit an empty value for those two items. If, however, a value is submitted, the validation test will then run the normal validation tests for those fields.

This is a bit of reverse logic from the way most other validators run. Normally, you would tell the validator which items are required. In Thor, we are telling the validator which fields are NOT required. I feel that if a item is required, you should have either a specific validator for it or, at the very least, a field name map so it can run an implicit validation. This logic requires you to add in validation rules over time and to not simply have Thor remain static in the background. Since most fields will have some type of validation running, you can then opt-out the ones not required as needed with ease.

On-the-fly Validation

In some cases, you need to run specific validation tests against specific fields for one time or a special case. This “on-the-fly” type of operation can easily been done with Thor.

For example,

```
<cfxml variable="lib1">
<validators>

    <validator name="isColor">
    <regex><![CDATA[^#?([A-F]|[0-9]){3}(([A-F]|[0-9]){3})?$]]></regex>
    <errorCode>Not a valid hexadecimal color code</errorCode>
    </validator>

</validators>
</cfxml>

<cfset fieldsNotRequired = "">
<cfset errors = application.Thor.validate(form,fieldsNotRequired,lib1)/>
```

So in this case, “form.color” will be validated against isColor for this case only. None of the imported validators, the global validator, error codes or field name mappings will be used, and the isColor validator will not be directly imported into Thor.

Client-side Validation with jQuery

Thor does a good job of handling server-side validation, but you may still need a solution for client-side validation. This can be difficult for many developers because it may require using two different validation systems. This means doubling the code to get any basic form up and running. And again, this can be difficult to manage and maintain.

Thankfully with some basic jQuery scripting, you can make Thor handle client-side validations. One of the examples (should be Example 9) included with Thor is one which handles client-side validation with the use of the popular JavaScript library, jQuery. The jQuery code will dynamically determine the form input elements on the page. As a user enters data in that input element and then tabs

over to the next item, jQuery will do a simply POST call to ColdFusion which will then route that input data through Thor to see if it's valid. If it isn't, it will respond with the proper error code which jQuery will then display for the user. This continuous client-side validation can be very handy if you have large html forms where it takes several minutes for the user to enter all the data. As they go through the process of entering data, they discover in 'real-time' if there are any problems. When the users gets to a submit button, they can be fairly certain that most if not all the data they have enter is correct. This can make the experience more enjoyable for the end user.

This process can be very server chatty. So feel free to let me know how to it scales for your applications. The benefit of this is it's very easy to add on client-side validation and also you can now have just one set of validation libraries that can handle both client and server side validation. There's no other ColdFusion based solution that I'm aware of that can handle this.

Validate Ranges

In some cases, you have to validate a value to see if it is contain within a range of possible values. With CFC based validators you can use the full power of ColdFusion to validate the data. For example, look at the isCreditCard validator. With XML based validators, you can only use a regex operation which can make this a bit harder but thankfully there are plenty of regex tests out there to help.

Regex to test a value is between 1 and 100

```
^[1-9][0-9]?$
```

Regex to require password is at least 6 characters long, alphanumeric,etc..

```
(?=[-_a-zA-Z0-9]*?[A-Z])(?=[-_a-zA-Z0-9]*?[a-z])(?=[-_a-zA-Z0-9]*?[0-9])[-_a-zA-Z0-9]{6,}
```

Regex from http://regexlib.com/REDetails.aspx?regexp_id=1896

Logging

Thor can log data submitted and the validations that were run against it including the on-the-fly attempts. This helps in debugging and development.

To call the Thor log simply call getLog():

```
<cfdump var="#application.Thor.getLog()#">
```

This will dump a query of the items submitted and the type of validation it went through. You can turn the logging off by the setting with the Thor.cfc.

```
<cfset variables.instance.logging = true/>
```

In some cases, due to security concerns such as PCI, you may want to prevent Thor from recording certain values. That can also be done from a specific setting found within Thor.

```
<cfset variables.instance.itemValuesNotToLog = "CreditCard,SSN"/>
```

With this setting, social security numbers and credit card will not be recorded by the logger.

You can also limit the size of the log with a date/time cutoff. This setting is found within Thor.cfc

```
<cfset variables.instance.maxLogTime = 86400/>
```

This setting will allow a log to stay recorded for up to 86400 seconds, or 1 day.

Error codes and Internationalization

Each validator has an attribute called “ErrorCode”. This can be anything you like to pass back as the explanation of a failed validation test. If you work exclusively with English, then this can be the specific error in English. If however, you deal with multiple languages or are likely to, then you should consider making the error code some type of identifier which gets translated inside your view logic to the proper text for the user.

In cases where you have implicit validation, you are encouraged to create a specific error code. For example, “form.phone” does not have an explicit validator. It runs through implicit validation and could then use a error code called “phone” which will provide a speific error code or string if the implicit validation fails.

Error codes can be imported in with the validator libraries just like the field name mappings (FNMs) or as a separate CFC and XML. There are two Error codes set within Thor. One is the default error code if there is a validation test without a stated error code. The seocnd is the global error code if the value fails the global validator.

Enable Options for Libraries and Validators

A library has a “enabled” attribute to it. This is also true for specific validators. This can make it easier to toggle specific items off or on depending on your needs.

Validator CFC and XML Library Formats

The validators can be organized in either CFCs or XML documents or some combination of both. You can organize your validators by topic in this way. The validators have to start with “is” in the name to be consider by Thor. So “isEmail” will be used as an email validator. In that way you have other methods that might be used by the validators but is itself not a validation function. You have an optional “enabled” toggle for any validator or the entire file. To see examples of the validator formats and how they are used look at the examples under the “formatexamples” directory in the Thor respository you downloaded. There are also examples of the field name mapping documents and error codes.