

STRUTS VALIDATOR FRAMEWORK

TABLE OF CONTENTS

INTRODUCING THE JAKARTA COMMONS VALIDATOR	3
THE BENEFITS OF USING THE VALIDATOR FRAMEWORK	3
THE RELATIONSHIP BETWEEN STRUTS AND THE VALIDATOR	4
OVERVIEW OF THE VALIDATOR COMPONENTS	4
WHAT ARE VALIDATORS?	4
<i>The byte, short, integer, long, float, and double validators</i>	5
<i>The creditCard validator</i>	6
<i>The date validator</i>	6
<i>The email validator</i>	7
<i>The mask validator</i>	8
<i>The maxLength validator</i>	9
<i>The minLength validator</i>	9
<i>The range validator</i>	10
<i>The required validator</i>	11
<i>The requiredif validator</i>	11
CONFIGURING THE VALIDATOR FOR A STRUTS APPLICATION	13
HOOKING UP THE VALIDATOR TO STRUTS	13
THE VALIDATOR-RULES.XML FILE	14
THE VALIDATION.XML FILE	15
THE RESOURCE BUNDLE	17
CREATING YOUR OWN VALIDATORS	18
CLIENT-SIDE VERSUS SERVER-SIDE VALIDATION	20
CONCLUSION	21
RESOURCES	21

Every application has a responsibility to ensure that only valid data is inserted into its repository. After all, what value would an application offer if the data that it relied upon were corrupted? For applications that use a formal database, like a RDBMS, for example, there are rules or constraints that can be placed upon the fields, which help to guarantee that the data stored within it meets a certain level of quality. Any and all applications that utilize the data within the repository have a responsibility to protect the integrity of the data that they submit.

Attempts to insert or update data that do not meet the criteria should be detected as soon as possible and rejected. This detection usually occurs in several places throughout an application; the presentation tier (if one is present) might perform some level of validation, the business objects typically have business-level validation rules, and as mentioned, the data repository usually does, as well.

Unfortunately, because data validation can occur in several areas within an application, there's usually a certain amount of redundancy that exists in applications for validating application data. This redundancy is generally an unwanted characteristic of an application, since it typically means longer development and maintenance times because the work is being repeated in multiple places. A more resilient application will attempt to reuse the validation rules throughout the application. This most likely translates into quicker development, easier customization cycles, and a more flexible application.

Introducing the Jakarta Commons Validator

The Validator framework is an open source project that was created by David Winterfeldt and is part of the Jakarta Commons subproject. The Commons project was created for the purpose of providing reusable components like the Validator. Version 1.0 of the Validator was released at the beginning of November 2002.

The Benefits of Using the Validator Framework

The Validator framework offers several benefits over the more conventional method of defining validation rules within the source code of an application. A few of the benefits include:

- A single definition of validation rules for an application.
- Validation rules are loosely coupled to the application.
- Server-side and client-side validation rules can be defined in one location.
- Configurations of new rules and/or changes to existing rules are made simpler.
- Supports Internationalization.
- Supports regular expressions.
- Can be used for both Web-based and standard Java applications.
- Promotes a declarative approach rather than a programmatic one.

These benefits aside, the most important characteristic of the Validator is its inherent support for pluggability. The Validator comes with several built-in validation rules that

you can leverage right "out of the box." More importantly, however, the Validator allows you to define your own validation routines and easily plug them into the existing framework.

Note: A *regular expression* is a formula for matching strings that follow some pattern. Many Unix command-line and programming uses regular expressions utilities. For more about regular expressions, see the "Using Regular Expressions" web page by Stephen Ramsay. <http://etext.lib.virginia.edu/helpsheets/regex.html>.

The Relationship Between Struts and the Validator

It should be pointed out that the Validator was originally created for use with the Jakarta Struts framework. The creator of the Validator, David Winterfeldt, was using Struts and realized that there was a great deal of redundancy in programming the same validation rules over and over again inside of the Struts ActionForm classes. He decided to create the Validation framework to eliminate that redundancy and subsequently, the Validator was born.

Although the Validator was originally created for use with Struts, it is designed and built in such a way that makes it possible to use on its own, without Struts. This characteristic allows you to use the framework in your applications, Struts-based or not. Just because you are not using Struts doesn't mean that you can't leverage the fine work that's been done. In fact, this is why the Validator is part of the Jakarta Commons project rather than being tied directly to the Struts project.

Overview of the Validator Components

There are several components that make up the Validator framework.

- Validators
- Configuration Files
- Resource Bundle
- JSP Custom Tags
- Validator Form Classes

What are Validators?

A Validator is a Java class that, when called by the Validator framework, executes a validation rule. The framework knows how to invoke a Validator class based on its method signature, as defined in a configuration file. Typically, each Validator provides a single validation rule, and these rules can be chained together to form a more complex set of rules.

Note: It's possible to define multiple rules within a single Java class. You might do this for convenience. Each validation rule is a static method and contains no client-specific state.

The framework provides 15 default validation rules, which are sometimes referred to as "basic validators." The basic Validators are listed in Table 1.

Table 1. Basic Validators provided by the Validator framework in Struts Ver 1.1.

Name	Description
<code>byte, short, integer, long, float, double</code>	Checks if the value can safely be converted to the corresponding primitive.
<code>CreditCard</code>	Checks if the field is a valid credit card number.
<code>date</code>	Checks if the field is a valid date.
<code>email</code>	Checks if the field is a valid email address.
<code>mask</code>	Succeeds if the field matches the corresponding regular expression mask.
<code>maxLength</code>	Checks if the value's length is less than or equal to the given maximum length.
<code>minLength</code>	Checks if the value's length is greater than or equal to the given minimum length.
<code>range</code>	Checks if the value is within a minimum and maximum range.
<code>required</code>	Checks if the field isn't null and length of the field is greater than zero, not including white space.
<code>requiredif</code>	Checks if the field isn't null based on the values of other fields.

The byte, short, integer, long, float, and double validators

These validators all apply the standard `type.parseType` methods to the value. If an Exception is caught, the validator returns false. Otherwise, it succeeds:

```
<field property="amount"
depends="required, double">
<arg0 key="1002"/>
</field>
```

The key 1002's value is being taken from the resource bundle (application resources properties file).

The creditCard validator

The creditCard validator analyzes the value to see if it could be a credit card number:

```
<field property="creditCard"  
depends="required, creditCard">  
<arg0 key="1025"/>  
</field>
```

The key 1025's value is being taken from the resource bundle (application resources properties file).

Credit card numbers include a parity-check digit. The validation checks for this digit and other business rules, such as whether the card's prefix matches one of the credit card vendors (American Express, MasterCard, VISA) and whether the length of the number is correct for the indicated vendor.

The date validator

The date validator checks to see if the value represents a valid date:

```
<field property="date"  
depends="required, date">  
<arg0 key="1026"/>  
</field>
```

The key 1026's value is being taken from the resource bundle (application resources properties file).

The validator passes the standard Locale object (java.util.Locale) maintained, So the framework to the date utilities automatically localizes the result. The datePattern attribute will pass a standard date pattern to a java.text.SimpleDateFormat object:

```
<var>  
<var-name>datePattern</var-name>  
<var-value>MM/dd/yyyy</var-value>  
</var>
```

As in the above case datePattern attribute accepts dates like 12/23/2003 or 1/1/2003 or 1/1/03.

This pattern **did not validate** dates like 12/2/20/03,1/1/03/, 1/1/2003/,1/1/200/3.

Internally, the datePattern attribute is used in the SimpleDateFormat constructor and then used to parse the value:

```
SimpleDateFormat formatter = new SimpleDateFormat (datePattern);  
Date date = formatter.parse(value);
```

If the parse succeeds, the validator succeeds.

If the datePatternStrict attribute is set instead, the length is also checked to

ensure a leading zero is included when appropriate:

```
<var>
<var-name>datePatternStrict</var-name>
<var-value>MM/dd/yyyy</var-value>
</var>
```

As in the above case datePatternStrict attribute accepts date like 12/23/2003,12/1/2003 only.

This pattern **did not validate** dates like 12/1/20/03,12/1/200/3,12/2/2003/.

So we used date validator as well as the mask validator in Reman Web Tool. We wrote regular expression for accepting dates in the required formats.

The regular expression would be like

```
^(?(\d{1,2}))?[/ ]?(?(\d{1,2})[/ ])?(\d{2,4})$
```

The above regular expression accepts dates in the following formats only
12/12/2003,1/1/2003,1/1/03,1/12/2003,12/1/2003,1/30/03,12/1/03.

When no pattern is specified, the DateFormat.SHORT format for the user's Locale is used. If the Struts Locale object is not available, the server's default Locale is used.

The setLenient method is set to false for all date transformations.

The email validator

The email validator employs an *extensive* check of the format of a prospective email address to be sure it is in accordance with the published specification:

```
<field property="emailAddress"
depends="required,email">
<arg0 key="2042"/>
</field>
```

The key 2042's value is being taken from the resource bundle (application resources properties file).

The email address like

_em@hty.com

xyz@x.com

xyz@x.com_

+*we@yahoo.com

are **not** being **validated** by the **validator framework's email validator**.

The mask validator

The mask validator checks the value against a regular expression and succeeds if the pattern matches:

```
<field property="zip" depends="mask">
  <arg0 key="1080"/>
  <var>
    <var-name>mask</var-name>
    <var-value>^\d{5}\d*$</var-value>
  </var>
</field>
```

The key 1080's value is being taken from the resource bundle (application resources properties file).

The above regular expression accepts zip in the following format only 45678 (ie. it accepts only 5 digits)

The Jakarta RegExp package [ASF, Regexp] is used to parse the expression. If an expression needs to be used by more than one field, it can also be defined as a constant in the validation.xml file—for example:

```
<constant>
  <constant-name>zip</constant-name>
  <constant-value>^\d{5}\d*$</constant-value>
</constant>

<constant>
  <constant-name>phone</constant-name>
  <constant-value>^\((?\d{3})\)?[- ]?(?\d{3})[- ]?(?\d{4})$</constant-value>
</constant>
```

The constants are defined in the global tag of the validation.xml.

The constants are being used like

```
<field property="phoneNumber" depends="mask">
  <arg0 key="2039"/>
  <var>
    <var-name>mask </var-name>
    <var-value>${phone}</var-value>
  </var>
</field>
```

As in the above case the phoneNumber field accepts the data like 123-234-3454 only.

Like most of the other standard validators, the mask validator is declared to be dependent on the required validator. Therefore, if a field depends on both required and mask, then the required validator must complete successfully before the mask validator is applied.

The `maxLength` validator

The `maxLength` validator checks the high end of the range; it succeeds if the field's length is less than or equal to the `max` attribute:

```
<field property="comments"
depends="required,maxlength">
<arg0 key="1075"/>
<arg1 name="maxlength" key="{var:maxlength}" resource="false"/>
<var>
<var-name>maxlength</var-name>
<var-value>1000</var-value>
</var>
</field>
```

The key `1075`'s value is being taken from the resource bundle (application resources properties file).

This field element makes sure that the length of the comments (a text area field) does not exceed 1000 characters. Note that we pass the length as an argument to the validation message.

The `minLength` validator

The `minLength` validator checks the low end of the range; it succeeds if the field's length is greater than or equal to the `min` attribute:

```
<field property="password"
depends="required,minlength">
<arg0 key="1085"/>
<arg1 name="minlength" key="{var:minlength}" resource="false"/>
<var>
<var-name>minlength</var-name>
<var-value>5</var-value>
</var>
</field>
```

The key `1085`'s value is being taken from the resource bundle (application resources properties file).

The field element stipulates that the password must be entered and must have a length of at least five characters.

The range validator

The range validator checks that the value falls within a specified minimum and maximum:

```
<field property="priority"
depends="required,integer,range">
<arg0 key="1090"/>
<var>
<var-name>min</var-name>
<var-value>1</var-value>
</var>
<var>
<var-name>max</var-name>
<var-value>4</var-value>
</var>
</field>
```

The key 1090's value is being taken from the resource bundle (application resources properties file).

This validator would succeed if the digit 1, 2, 3, or 4 were entered into the field. In practice, the error message should display the minimum and maximum of the range, to help the user get it right. You can use the arg elements to include the min and max variables in the message by reference:

```
<field property="priority"
depends="required,integer,range">
<arg0 key="1090"/>
<arg1 name="range" key="{var:min}" resource="false"/>
<arg2 name="range" key="{var:max}" resource="false"/>
<var>
<var-name>min</var-name>
<var-value>1</var-value>
</var>
<var>
<var-name>max</var-name>
<var-value>4</var-value>
</var>
</field>
```

This implies that the template for the range messages looks something like this: errors.range=Please enter a value between **{1}** and **{2}**.

If the range validator fails for the priority field, the validation message would read: Please enter a value between **1** and **4**.

By default, the validator assumes that the key of an arg element matches a key in the resource bundle and will substitute the value of the resource entry for the value of the key attribute. The resource=false switch tells the validator to use the value as is.

The required validator

The required validator is both the simplest and the most commonly used of the validators:

```
<field property="customerNo"
depends="required">
<arg0 key="2000"/>
</field>
```

The key 2000's value is being taken from the resource bundle (application resources properties file).

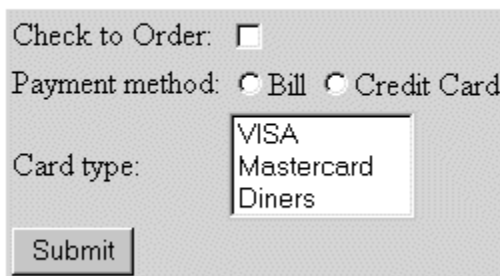
The **required** validator **did not work** for **select boxes** or **drop down** boxes in Reman Web Tool. So we wrote our **own validations** (custom validations) and added them in the validation-rules.xml, and used them like any other basic validators in the validation.xml.

If nothing or only white space is entered into a field, then the validation fails, and an error is passed back to the framework. Otherwise, the validation succeeds. To determine whether the field contains only white space, the standard String.trim() method is called (value.trim().length() == 0). Since browsers do not submit empty fields, any field that isn't required will skip all the validations, if the field is null or has a length of zero.

The requiredif validator

An example of how to use requiredif validator:

Example Order Page



Check to Order:

Payment method: Bill Credit Card

Card type:
Mastercard
Diners

Submit

If the user checks the "Check to Order" box, then "Payment method" must be chosen. If "Credit Card" is selected then the "Card type" must also be selected.

```
<form name="orderForm">
  <field property="method"
depends="requiredif">
  <arg0 key="Payment method" resource="false"/>
  <var>
    <var-name>field[0]</var-name>
    <var-value>order</var-value>
```

```

    </var>
    <var>
      <var-name>field-test[0]</var-name>
      <var-value>NOTNULL</var-value>
    </var>
  </field>
</form>

```

Read it like this: The payment method ("method") is required if the "order" control is not null (i.e. selected or checked).

In the same way we may add the next rule: Card type must be selected if payment method is "Credit Card":

```

<field property="cardtype"
  depends="requiredif">
  <arg0 key="Card type" resource="false"/>
  <var>
    <var-name>field[0]</var-name>
    <var-value>method</var-value>
  </var>
  <var>
    <var-name>field-test[0]</var-name>
    <var-value>EQUAL</var-value>
  </var>
  <var>
    <var-name>field-value[0]</var-name>
    <var-value>card</var-value>
  </var>
</field>

```

The variable names must be written exactly like this: *field[0]*, *field-test[0]*, *field-value[0]*.

The **requiredif** validator **did not work** for us in Reman Web Tool. So we wrote our **own validations** (custom validations) and added them in the validation-rules.xml, and used them like any other basic validators in the validation.xml.

In the next release (or maybe the first update of this) **requiredif** will be replaced by a much more flexible validator called **validwhen**. Since this feature is not yet available, an example of how to use this validator can be seen below:

```

<field property="emailAddress" depends="validwhen">
  <arg0 key="userinfo.emailAddress.label"/>
  <var>
    <var-name>test</var-name>
    <var-value>((sendNewsletter == null) or (*this* != null))</var-value>
  </var>
</field>

```

For more details about this validator use can see [Validator User's Guide](#)

As you can see the validator takes a single *var* named *test*, which must be *true* in order for the validator to return success. **This** refers to the field being tested and other fields may be referenced by using their names.

Configuring the Validator for a Struts Application

One of the things that give the Validator framework its flexibility is that all of the rules and details are declaratively configured in external files. The application doesn't have to know anything about the specific validation rules. This characteristic allows the set of rules to be extended or modified without having to touch the code. This is very important when you have to customize each install of the application or when business requirements change, which they inescapably do.

When using the Validator framework with Struts 1.1, there are two configuration files used. One is called *validator-rules.xml* and the other is *validation.xml*. You can name these files anything that you want, or even combine the contents into a single XML file. However, it's better to leave them separated, because each one fulfills a slightly different purpose.

Note: If you just downloaded the Validator package from the Jakarta Commons site, these two files are not included. They are only present with a download of Struts, which includes the Validator.

Hooking Up the Validator to Struts

The first thing you need to do is to make your Struts application aware of the Validator. You do this by using a new feature of Struts 1.1 called "Plug-in." Just add the following to the Struts configuration file:

```
<plug-in classname="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validation.xml"/>
</plug-in>
```

And Struts automatically becomes aware of the Validator.

The other necessary steps are to create the ActionForms (standard ones or Dynamic) and ensure the Validator configuration files are available, and also extend the ValidatorActionForms or ValidatorDynaActionForms. There's no need to call the validation rules or do anything special. Struts will automatically do this, based on the declarative configurations. You will, however, need to have the error messages tag in the JSPs in order to see the validation failures.

The validator-rules.xml File

The *validator-rules.xml* file defines the Validator definitions available for a given application. The *validator-rules.xml* file acts as a template, defining all of the possible Validators that are available to an application.

Note: this XML file and the one we'll discuss next must be placed in a location where they can be found by the class loader. When using the Validator with a Web application, the correct location is under the *WEB-INF* directory.

The *validator-rules.xml* file is governed by *validator-rules_1_1.dtd*, which can be found and downloaded at jakarta.apache.org/struts/dtds/validator-rules_1_1.dtd.

The most important element within the *validator-rules.xml* file is contained within the `<validator>` element, as shown in Example.

Example. A simple validator-rules.xml File

```
<form-validation>
  <global>
    <validator
      name="required"
      classname="org.apache.struts.util.StrutsValidator"
      method="validateRequired"
      methodparams=" java.lang.Object,
                    org.apache.commons.validator.ValidatorAction,
                    org.apache.commons.validator.Field,
                    org.apache.struts.action.ActionErrors,
                    javax.servlet.http.HttpServletRequest "
      msg="errors.required" />

    <validator name="minlength"
      classname="org.apache.struts.util.StrutsValidator"
      method="validateMinLength"
      methodparams=" java.lang.Object,
                    org.apache.commons.validator.ValidatorAction,
                    org.apache.commons.validator.Field,
                    org.apache.struts.action.ActionErrors,
                    javax.servlet.http.HttpServletRequest "
      depends="required"
      msg="errors.minlength" />
  </global>
</form-validation>
```

There is one `<validator>` element for each validator that an application uses. In Example, there are two validators shown; one is the required validator and the other is the `minlength` validator. There are many attributes supported by the `<validator>` element. These attributes are necessary so that the framework knows the correct class and method to invoke on the Validator. For example, in the `required` validator element in Example, the method `validateRequired()` will be called in the `org.apache.struts.util.StrutsValidator` class. Validators can also depend on one another.

This is shown in Example with the `minlength` Validator. It includes a `depends` attribute, which indicates that it depends on the required validator. The `msg` attribute allows you to specify a key from a resource bundle that the framework will use to generate the correct error message. A resource bundle is used to help localize the error messages.

The `<validator>` element also supports a `<javascript>` child element that allows you to specify a JavaScript function that can be executed on the client side. In this way, both server-side and client-side validation can be specified in a single location, making maintenance easier.

The validation.xml File

The second configuration file for the Validator is normally named *validation.xml*, although again you are free to name it whatever you like or just put the contents into the *validator-rules.xml* file.

The *validation.xml* file is where you couple the individual Validators defined in the *validator-rules.xml* to components within your application. Since we are talking about using the Validator with Struts, the coupling occurs between the Validators and Struts ActionForm classes. ActionForm classes are simple JavaBean-like components within the Struts framework that capture user input and help to transfer the input to components deeper within the application. ActionForms also provide a convenient spot to validate the user input before passing it to the business layer. Example shows a simple *validation.xml* file.

Example. A simple validation.xml File

```
<form-validation>
  <formset>
    <form name="orderForm">
      <field
        property="firstName"
        depends="required">
        <arg0 key="1002"/>
      </field>

      <field
        property="lastName"
        depends="required">
        <arg0 key="1003"/>
      </field>
    </form>
  </formset>
</form-validation>
```

Example shows a `<form>` element with the attribute name equal to `orderForm`. The `orderForm` is an ActionForm that has been defined in the Struts configuration file.

Therefore, the XML in Example is coupling that ActionForm and its `firstName` and `lastName` properties (remember that ActionForms are simple JavaBeans) to the required Validator. This can be seen in the `<field>` elements, respectively.

The Validation.xml general format:

```
<!-- 1 -->
<formset>
<!-- 2 -->
<form name="registerForm">
<!-- 3,4 -->
<field
property="firstname"
depends="required, mask">
<!-- 5 -->
<msg
name="mask"
key="1005"/>
<!-- 6 -->
<arg0
key="1006"/>
<!-- 7,8 -->
<var>
<var-name>mask</var-name>
<var-value>^[a-zA-Z0-9]*$</var-value>
</var>
</field>
<!-- 9 -->
<field
property="password"
depends="required,minlength">
<arg0
key="1006"/>
<var>
<var-name>minlength</var-name>
<var-value>5</var-value>
</var>
</field>
</form>
</formset>
```

1. A `formset` is a wrapper for one or more forms.
2. Each form element is given its own name. This should correspond to either the Form bean name or the action path from your Struts configuration.
3. Each `form` element is composed of a number of `field` elements.
4. The `field` elements designate which validator(s) to use with the `depends` attribute.

5. The optional `msg` element lets you specify a custom message key for a validator and the message key to use for any replacement parameters.
6. The `arg0` element specifies the first replacement parameter to use with any messages that need them.
7. The `var` element is used to pass variable properties to the validator.
8. Here we pass a regular expression to the `mask` validator. The expression says usernames can contain only the alphabet characters and numerals.
9. Here we say that a password is required and must be at least five characters long. The password length is a business requirement to help make the accounts more secure. The password validation message uses the default `minlength` or `required` messages, defined in `validation-rules.xml` (`errors.minLength` and `errors.required`).

There are actually many other features that allow you to define constants and global values that can be used throughout the `validation.xml` file. This comes in handy when you need to reuse certain constants or values over and over. For a more detailed discussion of the elements and attributes of the `validation.xml` file, you can download the DTD: jakarta.apache.org/struts/dtds/validation_1_1.dtd.

The Resource Bundle

The resource bundle is used to help localize messages and other textual information when dealing with users from various locales. It's also beneficial because it reduces the amount of redundant text hard-coded within an application. So instead of using the text label "Name:" directly within one or more JSPs, for example, you can put this text string within a resource bundle and pull the value from the bundle using a logical key. This way, if you needed to change the text label to "First Name:" you would only need to change it in one place.

For the Validator, the error messages that are created when validation rules fail come from the resource bundle. The Validator provides several default messages that can be placed in the Struts application resource bundle along with the normal application message resources. They look like:

```
#Normal resource bundle messages
1002=First Name
1003=Last Name
.....

#Error messages used by the Validator
errors.required={0} is required.
errors.minLength={0} cannot be less than {1} characters.
errors.maxLength={0} cannot be greater than {1} characters.
errors.invalid={0} is invalid.
errors.email={0}.
.....
```

When a validation rule fails, an error message is created for that specific validation rule. The framework will also automatically insert parameters for the message. For example, if we were using the validation rules from the Example's on the `orderForm` and the `firstName` property was null, we would see an error message like:

```
First Name is required.
```

You can also modify the messages within the bundle and/or configuration files to display any message you want

Creating Your Own Validators

Even though the Validators provided by the framework cover many of the validation rules that Web applications require, there are special needs that will require you to create your own rules. Fortunately, the Validator framework is easily extensible and the effort to do so is minimal.

To create your own validator, just create a Java class that implements your special validation rule. For example, suppose you needed a rule to validate that a user was the legal drinking age; this might be necessary to purchase alcoholic beverages online. You could possibly use one of the existing validation rules, but it would be more obvious to create a rule to validate that the user meets the required drinking age. The Java class might look something like the one in Example.

The validator's method must have a signature that contains a set of these parameters:

- `java.lang.Object` - The bean on which validation is being performed.
- `org.apache.commons.validator.ValidatorAction` - The current `ValidatorAction` being performed.
- `org.apache.commons.validator.Field` - The `Field` object that's being validated.
- `org.apache.struts.action.ActionErrors` - The errors objects to add an `ActionError` to if the validation fails.
- `javax.servlet.http.HttpServletRequest` - Current request object.

Example. A custom validation rule

```
package com.business.util;

import java.io.Serializable;
import javax.servlet.http.HttpServletRequest;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.GenericValidator;
import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.ValidatorUtil;
import org.apache.struts.action.ActionErrors;
```

```

import org.apache.struts.util.StrutsValidatorUtil;

public class NewValidator implements Serializable{

    public static boolean validateDrinkingAge( Object bean,
        ValidatorAction va,
        Field field,
        ActionErrors errors, HttpServletRequest request) {

        String value = null;
        if (isString(bean)) {
            value = (String) bean;
        } else {
            value =
                ValidatorUtil.getValueAsString(bean, field.getProperty());
        }
        String sMin = field.getVarValue("drinkingAge");

        if (!GenericValidator.isBlankOrNull(value)) {
            try {
                int iValue = Integer.parseInt(value);
                int drinkingAge = Integer.parseInt(sMin);

                if ( iValue < drinkingAge ){
                    errors.add(field.getKey(),
                        StrutsValidatorUtil.getActionError(request, va, field));
                    return false;
                }
            } catch (Exception e) {
                errors.add(field.getKey(),
                    StrutsValidatorUtil.getActionError(request, va, field));
                return false;
            }
        }
        return true;
    }

    private static boolean isString(Object o) {
        if (o == null) {
            return (true);
        }
        return (String.class.isInstance(o));
    }
}

```

After you create the new Validator, you simply add it to the list of existing Validators in the *validation-rules.xml* file. Once that's done, you can use the new Validator just like the "basic validators."

Add the following entry to Validation-rules.xml:

```
<validator
  name="drinking"
  classname="com.business.util.NewValidator"
  method=" validateDrinkingAge "
  methodparams=" java.lang.Object,
                 org.apache.commons.validator.ValidatorAction,
                 org.apache.commons.validator.Field,
                 org.apache.struts.action.ActionErrors,
                 javax.servlet.http.HttpServletRequest "
  msg="errors.drinkingAge" />
```

In the Validation.xml the entry would be like the following:

```
<field property="age" depends="drinking">
  <arg0 key="label.age" />
  <var>
    <var-name>drinkingAge</var-name>
    <var-value>drinkingAge</var-value>
  </var>
</field>
```

Client-Side Versus Server-Side Validation

Finally, we should briefly mention the JavaScript support provided by the Validator. Because some applications would rather validate on the client side, it's necessary to sometimes write JavaScript to perform simple checks on the client.

The Validator provides the support for dynamically and automatically creating JavaScript validation rules based on rules within the configuration file. For each `<validator>` element, you can also include a `<javascript>` child element and include the JavaScript code right in the file. When a JSP is rendered that contains the appropriate custom tags, JavaScript will be rendered, as well, and perform validation on submittal of the form. The JSP custom tag is included in the set of Struts tags and is called `JavaScriptValidatorTag`. The inclusion of the tag would look like:

```
<html:form action="/orderForm" onsubmit="return validateorderForm(this);">

<html:javascript formName="orderForm" />
```

Conclusion

The Struts Validator is a powerful addition to the framework. It allows validations to be managed in a separate configuration file, where they can be reviewed and modified without changing Java or Java Server Page code. This is important since, like localization, validations are tied to the business tier and should not be mixed with presentation code.

The framework ships with several basic validators that will meet most of your routine needs. You can easily add custom validators for special requirements. If required, the original Struts validate method can be used in tandem with the Struts Validator, to be sure all your needs are met.

Like the main Struts framework, the Struts Validator is built for localization from the ground up. It can even share the standard message resource file with the main framework, providing a seamless solution for your translators.

Validating input is an essential service that a web application framework must provide, and the Struts Validator is a flexible solution that can scale to meet the needs of even the most complex applications.

Resources

- "[Validating user input](#)" from the book "Struts in Action", by Ted Husted.
- An O'Reilly Article by Chuck Cavaness - <http://www.onjava.com/pub/a/onjava/2002/12/11/jakartastruts.html>
- Home of the Validator Creator - <http://home.earthlink.net/~dwinterfeldt/>
- Struts User Guide Info - http://jakarta.apache.org/struts/userGuide/dev_validator.html