Extracted from:

Stripes

. . . and Java Web Development Is Fun Again

This PDF file contains pages extracted from Stripes, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2008 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

wouldn't be nice to welcome the user to the form with validation error messages! With on="*save*", required-field validations are restricted to save() and so do not cause errors in form().

Once in the form, the user may very well enter invalid values and then click the Cancel button. You need to turn off all validations by annotating cancel() with @DontValidate so that the user will be allowed to cancel the form even if the input is not valid.

Whew. . . enough theory. Let's look at some examples.

4.2 Using Built-in Validations

Let's get back to our webmail application. We have a form to enter a contact's information, displayed by contact_form.jsp:

Download email_06/web/WEB-INF/jsp/contact_form.jsp

```
<s:form beanclass="${actionBean.class}">
  <div><s:hidden name="contact.id"/></div>
  <table class="form">
    <tr>
      <td>Email:</td>
      <td>
        <s:text name="contact.email" class="required"/>
      </td>
    </tr>
    <!--Same for First and Last name, Phone number, Birth date-->
    <tr>
      <td> </td>
      <td>
        <s:submit name="save" value="Save"/>
        <s:submit name="cancel" value="Cancel"/>
      </td>
    </tr>
  </table>
</s:form>
```

ContactFormActionBean sends the user to the form and handles the form submission:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
package stripesbook.action;
public class ContactFormActionBean extends ContactBaseActionBean {
    private static final String FORM="/WEB-INF/jsp/contact_form.jsp";

    @DefaultHandler
    public Resolution form() {
        return new ForwardResolution(FORM);
    }
```

CLICK HERE to purchase this book now.

```
    public Resolution save() {
        Contact contact = getContact();
        getContactDao().save(contact);
        getContext().getMessages().add(
            new SimpleMessage("{0} has been saved.", contact)
        );
        return new RedirectResolution(ContactListActionBean.class);
    }
    public Resolution cancel() {
        getContext().getMessages().add(
            new SimpleMessage("Action cancelled.")
        );
        return new RedirectResolution(ContactListActionBean.class);
    }
}
```

We'll now add some validations to this form.

Making a Field Required

Let's begin by making the contact's email address a *required* field. First, it's better to let the user know up front about required fields. One way is to make the field border thicker by adding a *"required"* class and styling it in the CSS file:

Download email_06/web/WEB-INF/jsp/contact_form.jsp

```
<s:text name="contact.email" class="required"/>
```

Download email_06/web/css/style.css

```
input.required {
  border-width: 2px;
}
```

Next, adding @ValidateNestedProperties with @Validate(field="email") to contact validates the *"contact.email"* nested property. Remember that the contact property moved to the parent ContactBaseActionBean, so the validation must override either the getter or the setter method in ContactFormActionBean:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
@ValidateNestedProperties({
    @Validate(field="email", required=true, on="save")
})
@Override
public void setContact(Contact contact) {
    super.setContact(contact);
}
```

CLICK HERE to purchase this book now.

Figure 4.3: A validation error for a required field

As we discussed, the on="*save*" restricts the validation to the save() event handler. Now, if the user saves the form with the email field left blank, a validation error occurs, and Stripes redisplays contact_form.jsp. To show the error message to the user as in Figure 4.3, add the <*s:errors/*> tag:

Download email_06/web/WEB-INF/jsp/contact_form.jsp

```
<s:form beanclass="${actionBean.class}">
  <s:errors/>
  <div><s:hidden name="contact.id"/></div>
  <table class="form">
```

Just like information messages, Stripes has a default way of displaying error messages: with a header message followed by the validation errors in a numbered list. A reasonable effort is made to construct error messages using the name of the field and the type of validation that failed, so we get something quite decent just by adding the <*s:errors/*> tag. In Chapter 6, *Customizing Stripes Messages*, on page 123, we'll talk about how to customize both the text and the presentation of error messages.

Email Addresses

We've made the email a required field, but this validates only that the user entered something in the field. It does not actually validate what the user entered. How about making sure that the email *format* is valid?

CLICK HERE to purchase this book now.

Joe Asks...

Where Should I Put the <s:errors/> Tag?

Placing <*s:errors/*> within the <*s:form*> tag displays the error messages associated with that form. When you have more than one form in a single page, you can display the errors for each form or place the <*s:errors/*> outside the <*s:form*> tag to display the error messages that occurred in the current action bean.

I mentioned that in Stripes validations can be implemented as type converters. To use a type converter, you indicate its class in the converter= attribute of @Validate. The EmailTypeConverter validates that the input is of email address format, so we can use it with converter= to validate the contact email:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
@ValidateNestedProperties({
    @Validate(field="email", required=true, on="save",
        converter=EmailTypeConverter.class)
})
@Override
public void setContact(Contact contact) {
    super.setContact(contact);
}
```

The EmailTypeConverter uses JavaMail to validate the email address, so we'll have to add the library to the WEB-INF/lib directory. Unless you are using Java 6, you will also have to add the JavaBeans Activation Framework:

```
WEB-INF/lib/javamail.jar
WEB-INF/lib/activation.jar
```

Now, entering an invalid email address such as "hello" displays this error message: "The value (hello) entered is not a valid email address."

Limiting the Length of Input

Let's add validation rules for the first and last name fields. These fields are optional, but if a value is entered, we'll enforce these restrictions:

- The first name cannot exceed twenty-five characters.
- The last name cannot exceed forty characters.
- The last name must be at least two characters.

CLICK HERE to purchase this book now.

Required Fields and the on Parameter

You can restrict the required=true validation to a *list* of event handlers, such as on={"save", "update"}. Another option is to specify the event handler(s) for which *not* to apply the validation using the ! negation symbol. For example, on="!save" executes the required=true validation for every event handler of the action bean except save(). You can also use a list with negations, as in on={"!save", "!update"}.

Do not mix "positive" and "negative" event handler names in the on= attribute, such as on={"save", "!update"}, because logically it doesn't make sense. (Think about it.)

As we can see in the following code, it's very simple to add these validations with the minlength= and maxlength= attributes:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
@ValidateNestedProperties({
    /* previous validations... */
    @Validate(field="firstName", maxlength=25),
    @Validate(field="lastName",  minlength=2, maxlength=40)
})
```

Since the first and last name fields are optional, each validation is executed *only if the user enters a value for that field*. Now, entering a single character in the last name field produces the error shown in Figure 4.4, on the next page. Notice that Stripes used the value of minlength= to make the message more helpful.

As a bonus, Stripes automatically generates the maxlength= attribute in the form's HTML <*input*> tags to match the value in the maxlength= attribute of @Validate:

```
<tr>
  <td>First name:</td>
  <td><input maxlength="25" type="text" name="contact.firstName"/></td>
</tr>
<tr>
  <td>Last name:</td>
  <td><input maxlength="40" type="text" name="contact.lastName"/></td>
</tr>
```

Any decent browser stops accepting characters in the text field after the maximum length has been reached.

CLICK HERE to purchase this book now.

Figure 4.4: A validation error for minimum input length

Of course, the validation in the action bean is still executed—we can't rely only on client-side validation, because users could sent input in other ways than using the form. It's still nice to immediately let the well-intentioned user know when they've reached the limit as they are typing a value into the text field.

Another nice feature is that Stripes does not stop at the first encountered validation error. Instead, as many errors as possible are accumulated during the validation process to provide more information to the user.

Validating with EL Expressions

We can also validate user input by using an EL expression in the expression= attribute of @Validate. The **boolean** value of the expression determines whether the validation passed. This gives us an easy way to add a validation based on a conditional expression.

Within the expression, we can refer to the field that we are validating using the keyword **this** and to other properties of the action bean by their names. The action bean context, the request scope, and the session scopes are available with context, request, and session.

The birth date already benefits from the implicit validation of converting the input to a java.util.Date. Now that we've added the <*s:errors/*> tag to the JSP, the user sees an error message after entering an invalid date. Let's use an expression to also validate that the birth date in the contact

CLICK HERE to purchase this book now.

form is *before the current date*. In other words, no unborn people in the contact list, please!

The key to this validation is that the current date is not a static value. So, we add a simple method in the action bean to provide it:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
public Date getToday() {
    return new Date();
}
```

Now, using an expression makes it a cinch to validate that the birth date is in the past:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
@ValidateNestedProperties({
    /* previous validations... */
    @Validate(field="birthDate", expression="${this < today}")
})
```

In the expression ${this < today}, **this** refers to the birthDate property, and today calls getToday() to obtain the current date.

Armed with this validation, submitting the form with a birth date in the future, such as 2040-01-27,[2] causes the action bean to return the error "The value supplied (Fri Jan 27 00:00:00 EST 2040) for field Contact Birth Date is invalid."

As you can see, using expressions gives you a concise and effective way of adding validations that are based on other fields or on values produced by any helper method.

Using Regular Expression Masks

Another way to validate user input is to use a regular expression mask.[3] To be considered valid, the entire input must match the mask. By placing the regular expression in the mask= attribute of @Validate, you can validate patterns that would otherwise require gobs of tedious code.

Consider the "Phone number" field in the contact form. For the sake of the example, let's say that the phone number should be in the format used in North America: a three-digit area code, followed by a three-digit

2. I'll be happy, but very surprised, if someone reads this book after 2040!
3. Refer to the java.util.regex.Pattern Javadocs for the regular expression syntax that Stripes uses.

CLICK HERE to purchase this book now.

Using ${ } in Expressions

Enclose the validation expression within ${ }, or don't—the choice is yours. Indeed, expression="this < today" and expression="${this < today}" are equivalent. Stripes automatically adds ${ } for you if you leave it out.

Personally, I prefer using ${ } because I find it makes it clearer that an EL expression is being used. Whichever format you choose, being consistent will certainly make your code more readable.

prefix and a four-digit suffix, as in (654) 456-4567. To be lenient with our users, we'll allow some flexibility with the input format:

- The parentheses around the area code are optional.
- The separators between each part of the phone number can be hyphens, periods, or spaces, or they can be omitted altogether.

For example, all these phone numbers are acceptable:

(654) 456-4567	654-456-4567	654 456 4567	654.456.4567
(654)456 4567	6544564567	654 4564567	654.456-4567

Adding this validation is easy by building a regular expression mask with the following constructs:

- \(? and \)? to represent an optional opening and closing parenthesis
- [-.]? to accept an optional hyphen, period or space
- \d to represent a digit
- {*N*} to indicate the previous construct repeated *N* times

With these constructs, we can validate the phone number by adding the following mask. Since the regular expression is in a Java String, we have to use \\ to represent \.

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
@ValidateNestedProperties({
    /* previous validations... */
    @Validate(field="phoneNumber",
        mask="\\(?\\d{3}\\)?[-. ]?\\d{3}[-. ]?\\d{4}")
})
```

CLICK HERE to purchase this book now.

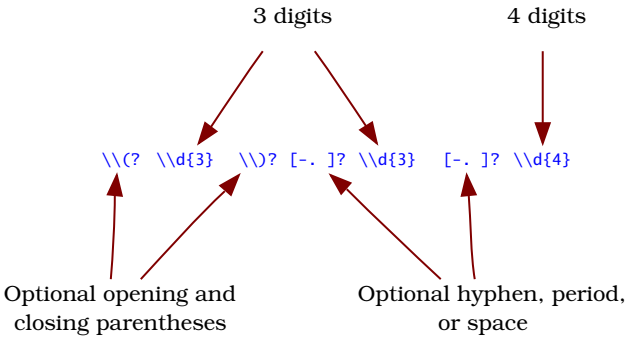

Figure 4.5: A regular expression to validate a phone number

OK, regular expressions are rarely pleasing to the eye, so I've tried to make it clearer by breaking it down as shown in Figure 4.5.

The *entire* input must match the regular expression, so incomplete phone numbers are also rejected. An example of entering an invalid phone number is shown in Figure 4.6, on the next page.

We've added a fairly sophisticated validation for the phone number with a @Validate annotation and a regular expression mask. Think about how much more code we'd need to implement this validation by parsing the input string ourselves!

The Cancel Button

The last thing we need to do in the contact form is to turn all validations off for the `Cancel` button. Otherwise, canceling the form won't work if there are any invalid values that were entered by the user. We just need to add the @DontValidate annotation to the cancel() event handler:

Download email_06/src/stripesbook/action/ContactFormActionBean.java

```
@DontValidate
public Resolution cancel() {
    getContext().getMessages().add(
        new SimpleMessage("Action cancelled.")
    );
    return new RedirectResolution(ContactListActionBean.class);
}
```

CLICK HERE to purchase this book now.

Figure 4.6: A validation error using a regular expression mask

Pretty good. We've added validation to the contact form, and all we needed were annotations in the action bean and a single *<s:errors/>* tag in the JSP.

We didn't use the minimum/maximum numerical value and credit card validations in the contact form because we don't have any fields that are relevant to those validations. Nevertheless, let's look at them briefly before continuing.

Minimum and Maximum Numerical Values

Stripes provides validation of minimum and maximum numerical values with the minvalue= and maxvalue= attributes of @Validate. These attributes accept values of type **double**, and they work for properties of any primitive numerical type as well as all subclasses of Number.

Suppose you wanted to restrict some field to a value between 0 and 7, inclusive. You would use this:

```
@Validate(minvalue=0, maxvalue=7)
private int someField;
```

Now, entering an invalid value for this field would give an error message such as this:

- "The minimum allowed value for Some Field is 0."
- "The maximum allowed value for Some Field is 7."

Again, Stripes is smart enough to use the values that we specify in the minvalue= and maxvalue= attributes to construct the error messages.

CLICK HERE to purchase this book now.

A Note About Trimming Input

After some discussion, the Stripes community agreed that user input should be trimmed before validating. This makes validations such as required fields, minimum length, and so on, behave as most developers expect: entering two spaces in a required field should not be valid, and it shouldn't pass a minlength=2 validation.

Because trimming the input is so often desirable, it is the default behavior in Stripes. You can disable trimming for a field by annotating it with @Validate(trim=false).

Credit Card Numbers

CreditCardTypeConverter checks that the input *could* be a valid credit card number, without actually connecting to anything to check whether an account with that number actually exists. Here's what the type converter does:

- Starts by removing all nondigit characters from the input
- Checks that the card corresponds to AMEX, Diners Club, Discover Card, enRoute, JCB, MasterCard, or Visa, based on the prefixes and the number of digits that these cards use
- Validates the Luhn algorithm[4] on the number

CreditCardTypeConverter is similar to EmailTypeConverter in that it validates the input without converting it to a different type. To use it, just add @Validate(converter=CreditCardTypeConverter.class) on the "Credit card number" field.

How Stripes Processes Built-in Validations

Now that we've seen examples of each built-in validation, let's take a closer look at how Stripes executes these validations. I've illustrated the process in Figure 4.7, on the following page. Validations are run on a list of fields, which initially contains every field. After performing a validation, only the fields that are valid are kept in the list for the next validation. The validations are arranged in order such that later validations are worth running only if previous validations have passed. Validation errors are accumulated and made available for the JSP to display with <*s:errors/*>.

4. See http://en.wikipedia.org/wiki/Luhn if you really want to know how that works.

CLICK HERE to purchase this book now.

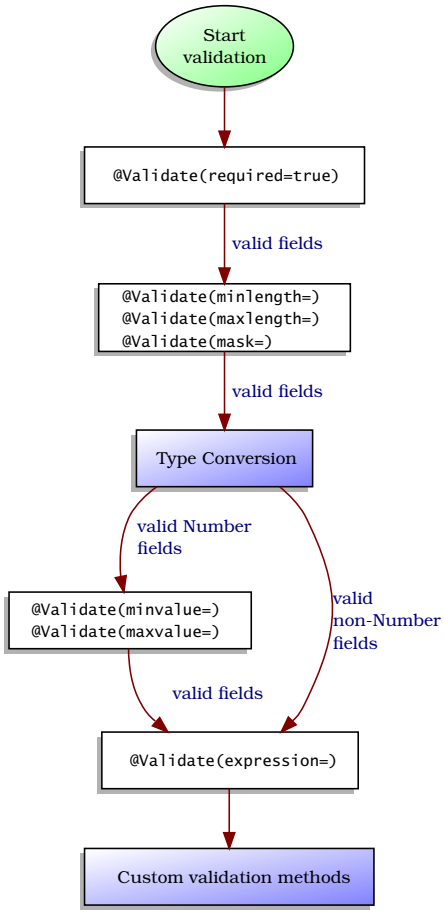

Figure 4.7: Processing validations in order of priority

In the middle of the diagram, notice the box labeled "Type Conversion." I've briefly touched on the subject that Stripes performs type conversion for all basic data types. If the type conversion passes and the property type extends Number, then the minimum and maximum numerical value validations are executed. We'll talk about type conversion in more detail in Chapter 5, *There's More to Life Than Strings: Working with Data Types*, on page 100.

After processing all built-in validations, Stripes moves on to *custom validation methods*. This is where you get to do pretty much anything you need to do to validate the input.

CLICK HERE to purchase this book now.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Stripes...and Java Web Development Is Fun Again's Home Page
http://pragprog.com/titles/fdstr
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments in the news.

Buy the Book

If you liked this PDF, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/fdstr.

Contact Us

Phone Orders:	1-800-699-PROG (+1 919 847 3884)
Online Orders:	www.pragprog.com/catalog
Customer Service:	orders@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com