Struts 2 :Validation Framework

Exploring the validation framework

- ***The validation framework architecture:***
  - There are three main components at play in the validation framework:
    - the *domain data*,
    - *validation metadata*, and
    - the *validators*.
  - Each plays a vital role in the work of validation.
  - DOMAIN DATA :
    - We must have some data to validate.
    - These DOMAIN DATA are assumed to hold the data our action will work with when it begins execution.

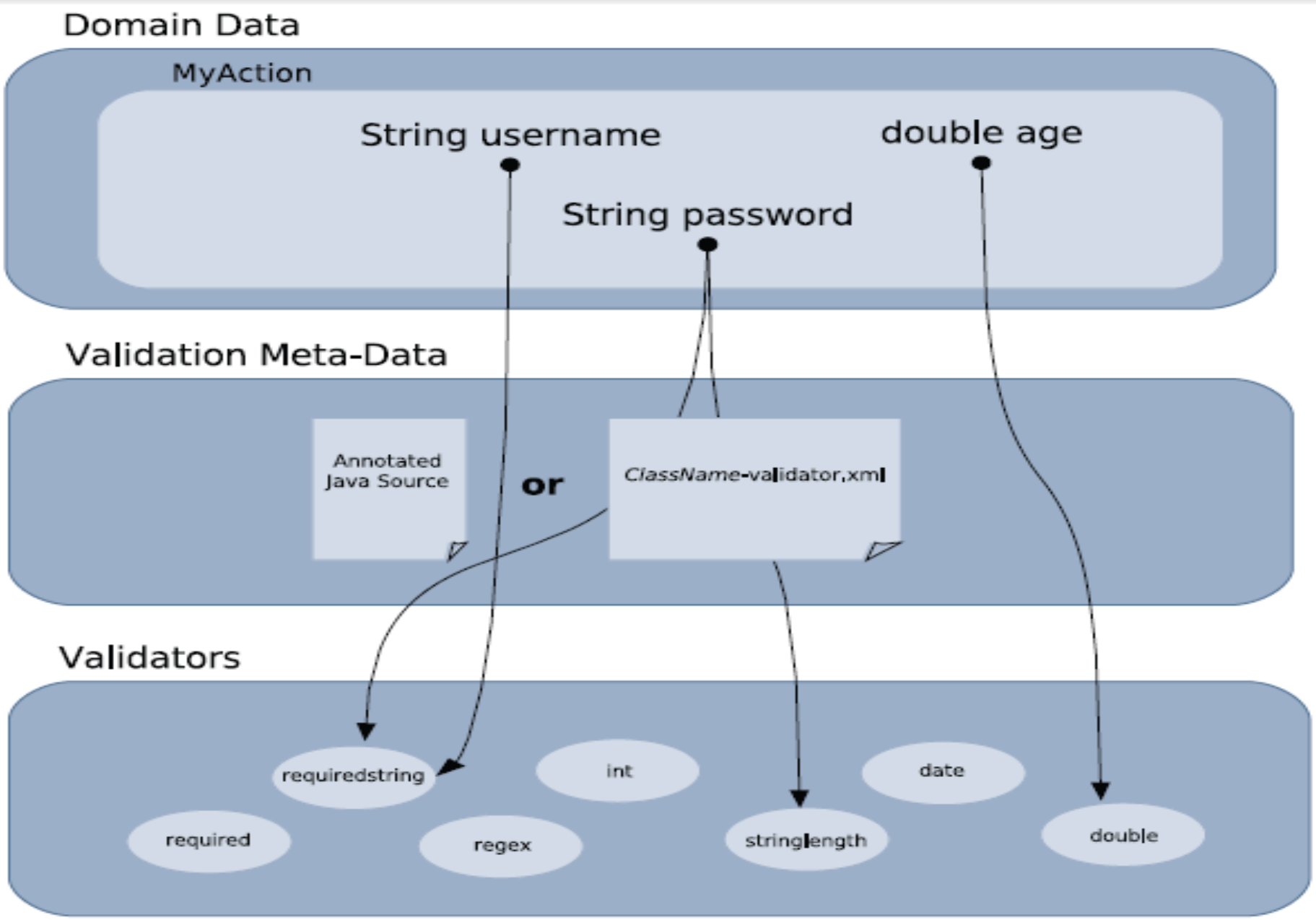


Figure 10.1 The validation framework uses metadata to associate validators with data properties.

Exploring the validation framework

- VALIDATION METADATA:
  - **A middle component lies between the validators and the data properties themselves**.
  - This middle component is the metadata that associates individual data properties with the validators that should be used to verify the correctness of the values in those properties at runtime.
  - You can associate as many validators with each property.
  - The developer can map data properties to validators with XML files or with Java annotations.

Exploring the validation framework

- VALIDATORS:
  - A validator is a reusable component that contains the logic for performing some fine-grained act of validation.
  - The framework comes with a rich set of built-in validators, and you can even write your own.
  - To have your data validated by these validators, you simply wire up your properties to the desired validators via some XML or Java annotations.
  - When the validation executes, each property is validated by the set of validators with which it's been associated by the metadata layer.

The validation framework in the Struts 2 workflow

- REVIEWING BASIC VALIDATION
  - When we place our validation in the **validate() method** on our actions, we recall here that how that validation works.
  - These interfaces are com. opensymphony.xwork2.Validateable and com.opensymphony.xwork2.ValidationAware.
  - Validateable exposes the validate() method, in which we've been stuffing our validation code,
  - ValidationAware exposes methods for storing error messages generated when validation finds invalid data.

The validation framework in the Struts 2 workflow

- REVIEWING BASIC VALIDATION
  - These interfaces work in cycle with an important interceptor known as the **workflow interceptor**.
    - When the workflow interceptor fires, it first checks to see whether the action implements Validateable.
    - If it does, the workflow interceptor invokes the validate() method.
    - If our validation code finds that some piece of data isn't valid, an error message is created and added to one of the ValidationAware methods that store error messages.
    - When the validate() method returns, the workflow interceptor still has another task.
    - It calls ValidationAware's hasErrors() method to see if there were any problems with validation. If errors exist, the workflow interceptor intervenes by stopping further execution of the action by returning the input result, which returns the user back to the form that was submitted.

The validation framework in the Struts 2 workflow

- INTRODUCING THE VALIDATION FRAMEWORK WORKFLOW
  - The validation framework actually shares quite a bit of the same functionality we've previously outlined for basic validation; it uses the ValidationAware interface to store errors and the workflow interceptor to route back to the input page if necessary.
  - In fact, the only thing that changes is the validation itself.
  - The workflow, as dictated by this stack of interceptors, remains constant regardless of which type of validation you choose to use.
  - In particular, note the following sequence of interceptors from the defaultStack, as defined in struts-default.xml:

```
<interceptor-ref name="params"/>
<interceptor-ref name="conversionError"/>
<interceptor-ref name="validation"/>
<interceptor-ref name="workflow"/>
```

The validation framework in the Struts 2 workflow

- INTRODUCING THE VALIDATION FRAMEWORK WORKFLOW
  - The params interceptor and the conversionError interceptor both fire before we get to the validation-related interceptors.
  - These two interceptors finish up the work of transferring the data from the request and converting it to the correct Java types of the target properties.
  - The validation interceptor has nothing to do with that form of validation.
  - Recall that the workflow interceptor invokes the validate() method to conduct basic validation.
  - Now we need to take note of the validation interceptor because it’s the entry into the validation framework.

The validation framework in the Struts 2 workflow

- INTRODUCING THE VALIDATION FRAMEWORK WORKFLOW
  - When this interceptor fires, it conducts all the validation that's been defined via the validation metadata we mentioned in the previous section.
  - The first functional unit in the pipeline is the data transfer and type conversion process.
  - This process, conducted by the params and conversionError interceptors, moves the data from the request parameters onto the properties exposed on the ValueStack.
  - In this Figure 10.2 The validation framework runs after **data transfer/ type conversion** and before the **workflow interceptor.**

The validation framework in the Struts 2 workflow

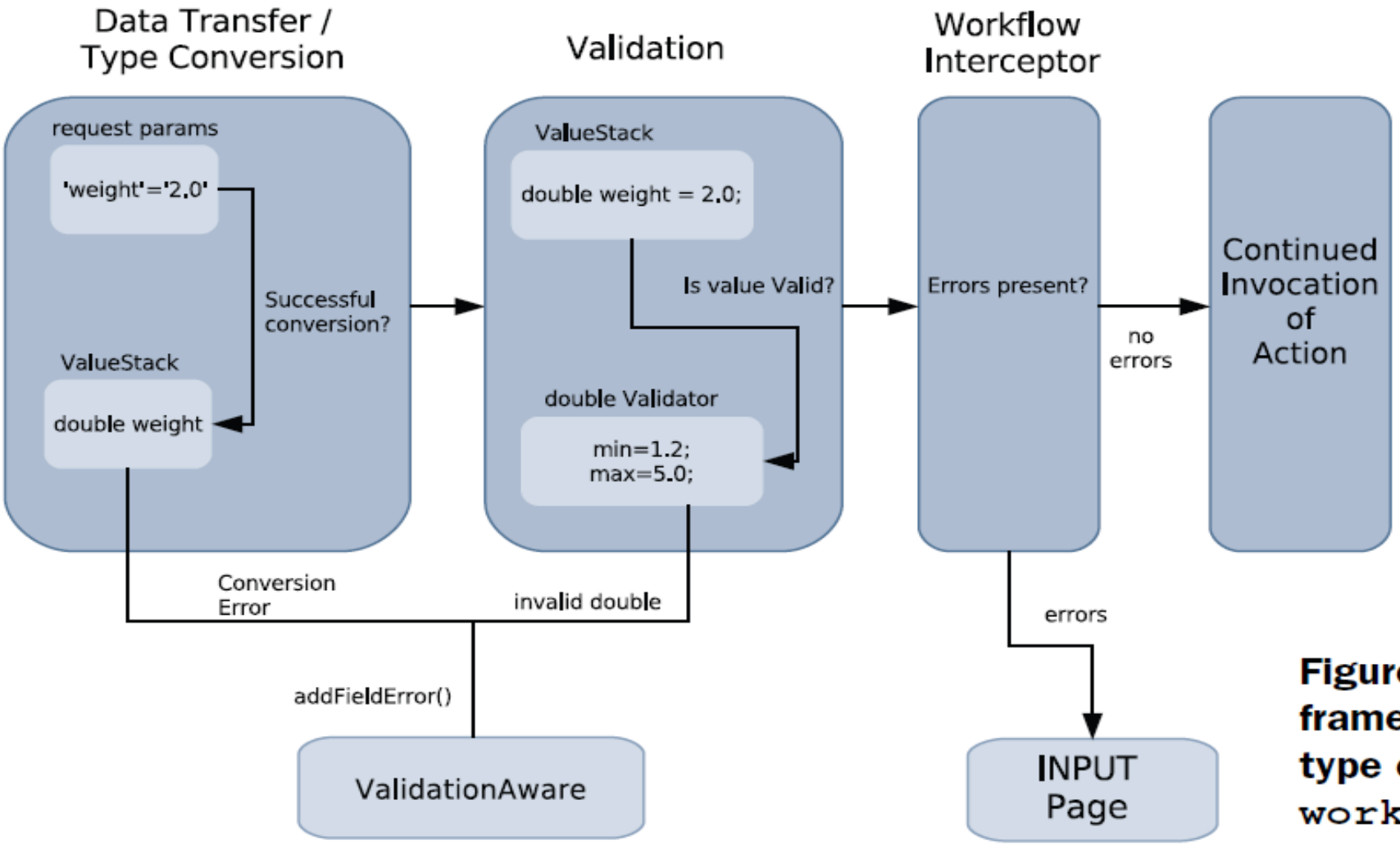


Figure
framev
type c
workf

Exploring the validation framework

- In fact, the only thing that changes is the validation itself.
- Sequence of interceptors from the defaultStack, as defined in struts-default.xml:

```
<interceptor-ref name="params"/>
<interceptor-ref name="conversionError"/>
<interceptor-ref name="validation"/>
<interceptor-ref name="workflow"/>
```

Exploring the validation framework

- We know that workflow it has two phases.
  - The **first phase** is to invoke the validate() method, if exposed by the current action. This is the entry point into basic validation.
  - The **second phase** is checking for errors.
  - The workflow interceptor checks the ValidationAware method hasErrors().
  - If there are none, it passes control on to the rest of the action invocation process.
  - If errors are found, workflow is diverted and we return to the input page and present the user with error messages so she can correct the form data.
- We can use both of methods at same time validate() method and validation framework also.

Declaring your validation metadata with ActionClass-validations.xml

Listing 10.2 Declares the validators that validate each exposed property

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2//
      EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">     ❶
<validators>                                                                  ❷
  <field name="password">                                                     ❸
     <field-validator type="requiredstring">                                  ❹
       <message>You must enter a value for password.</message>                ❺
     </field-validator>
  </field>
  <field name="username">
    <field-validator type="stringlength">
      <param name="maxLength">8</param>
      <param name="minLength">5</param>
      <message>While ${username} is a nice name, a valid username must
            be between ${minLength} and ${maxLength} characters long.
      </message>
    </field-validator>
  </field>
  <field name="portfolioName">
    <field-validator type="requiredstring">
      <message key="portfolioName.required"/>
    </field-validator>
  </field>
```

Declaring your validation metadata with ActionClass-validations.xml

```
    <field name="email">
      <field-validator type="requiredstring">
         <message>You must enter a value for email.</message>
      </field-validator>
      <field-validator type="email">
        <message key="email.invalid"/>
      </field-validator>
    </field>
    <validator type="expression">
      <param name="expression">username != password</param>
      <message>Username and password can't be the same.</message>
    </validator>
</validators>
```

6

7

Message Element Options

- The message element is used to specify the message that the user should see in the event of a validation error.
- In the simplest form, we simply embed the message text itself in the message element .
- However, several more options present themselves. First, we can use OGNL to make the message dynamic.
- An example of this can be seen in Register-validation.xml's declaration of the username field.

Message Element Options

```
<field name="username">
    <field-validator type="stringlength">
        <param name="maxLength">8</param>
        <param name="minLength">5</param>
        <message>While ${username} is a nice name, a valid username
                must be between ${minLength} and ${maxLength}
                characters long.
        </message>
    </field-validator>
</field>
```

Message Element Options

```
<field name="portfolioName">
    <field-validator type="requiredstring">
        <message key="portfolioName.required"/>
    </field-validator>
</field>
```

```
user.exists=This user ${username} already exists.
portfolioName.required=You must enter a name for your initial portfolio.
email.invalid=Your email address was not a valid email address.
```

Surveying the built-in validators

Validator name	Params	Function	Type
`required`	None	Verifies that value is non-null.	field
`requiredstring`	`trim` (default = true, trims white space)	Verifies that value is non-null, and not an empty string.	field
`stringlength`	`trim` (default=true, trims prior to length check), `minLength`, `maxLength`	Verifies that the string length falls within the specified parameters. No checks are made for unspecified length params—if you give no minimum, then an empty string would pass validation.	field
`int`	`Min`, `max`	Verifies that the integer value falls between the specified minimum and maximum.	field
`double`	`minInclusive`, `maxInclusive`, `minExclusive`, `maxExclusive`	Verifies that the double value falls between the inclusively or exclusively specified parameters.	field
`date`	`Min`, `max`	Verifies that the date value falls between the specified minimum and maximum. Date should be specified as `MM/DD/YYYY`.	field

Surveying the built-in validators

Validator name	Params	Function	Type
`email`	None	Verifies email address format.	field
`url`	None	Verifies URL format.	field
`fieldexpression`	`expression` (required)	Evaluates an OGNL expression against current `ValueStack`. Expression must return either `true` or `false` to determine whether validation is successful.	field
`expression`	`expression` (required)	Same as `fieldexpression`, but used at action level.	action
`visitor`	`Context`, `appendPrefix`	Defers validation of a domain object property, such as `User`, to validation declarations made local to that domain object.	field
`regex`	`expression` (required), `caseSensitive`, `trim`	Verifies that a `String` conforms to the given regular expression.	field

Surveying the annotations

Annotation	Description
ConversionErrorFieldValidator Annotation	Checks if there are any conversion errors for a field.
DateRangeFieldValidator Annotation	Checks that a date field has a value within a specified range.
DoubleRangeFieldValidator Annotation	Checks that a double field has a value within a specified range.
EmailValidator Annotation	Checks that a field is a valid e-mail address.
ExpressionValidator Annotation	Validates an expression.
FieldExpressionValidator Annotation	Uses an OGNL expression to perform its validator.
IntRangeFieldValidator Annotation	Checks that a numeric field has a value within a specified range.

Surveying the annotations

Annotation	Description
RegexFieldValidator Annotation	Validates a regular expression for a field.
RequiredFieldValidator Annotation	Checks that a field is non-null.
RequiredStringValidator Annotation	Checks that a String field is not empty.
StringLengthFieldValidator Annotation	Checks that a String field is of the right length.
StringRegexValidator Annotation	Invokes a regular expression to validate a String field.
UrlValidator Annotation	Checks that a field is a valid URL.
Validation Annotation	Marker annotation for validation at Type level.
Validations Annotation	Used to group validation annotations.
VisitorFieldValidator Annotation	Invokes the validation for a property's object type.
CustomValidator Annotation	Use this annotation for your custom validator types.

Validation Annotations

- **RequiredFieldValidator Annotation** : This validator checks that a field is non-null.
- **Usage :** The annotation must be applied at method level.
- **Parameters**

Parameter	Required	Default	Notes
message	yes		field error message
key	no		i18n key from language specific properties file.
fieldName	no		
shortCircuit	no	false	If this validator should be used as shortCircuit.
type	yes	ValidatorType. FIELD	Enum value from ValidatorType. Either FIELD or SIMPLE can be used here.

Examples

@RequiredFieldValidator(message = "Default message", key = "i18n.key", shortCircuit = true)

Using Struts 2 Int Validator

Step 1: Create the xml file and adds the following xml snippet in the **struts.xml** file.

struts.xml

```
<struts>
  <package name="example" namespace="/example" extends="struts-default">
      <action name="intValidation" class="example.NumAction">
        <result name="input">/pages/intInputForm.jsp</result>
        <result name="error">/pages/intInputForm.jsp</result>
        <result>/pages/intSuccess.jsp</result>
      </action>
  </package>
</struts>
```

Using Struts 2 Int Validator

Step 2 : Create the input form.
intInputForm.jsp

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head> <title>Input form</title></head>
<body>
      <s:form method="POST" action="intValidation">
          <s:textfield label="Enter Number" name="userinput" />
          <s:submit />
      </s:form>
</body>
</html>
```

Using Struts 2 Int Validator

Step 3 : Create the Action class.
NumAction.java

```
public class NumAction extends ActionSupport{
    private int userinput=0;
    public String execute() throws Exception{
          /*  if (getUserinput() >= 10 && getUserinput() <= 80){
                return SUCCESS;
              }
            else{ return ERROR;} */
      return SUCCESS;
  }
  public void setUserinput(int userinput)
  {     this.userinput = userinput;  }
  public int getUserinput()
  {     return userinput;  }
}
```

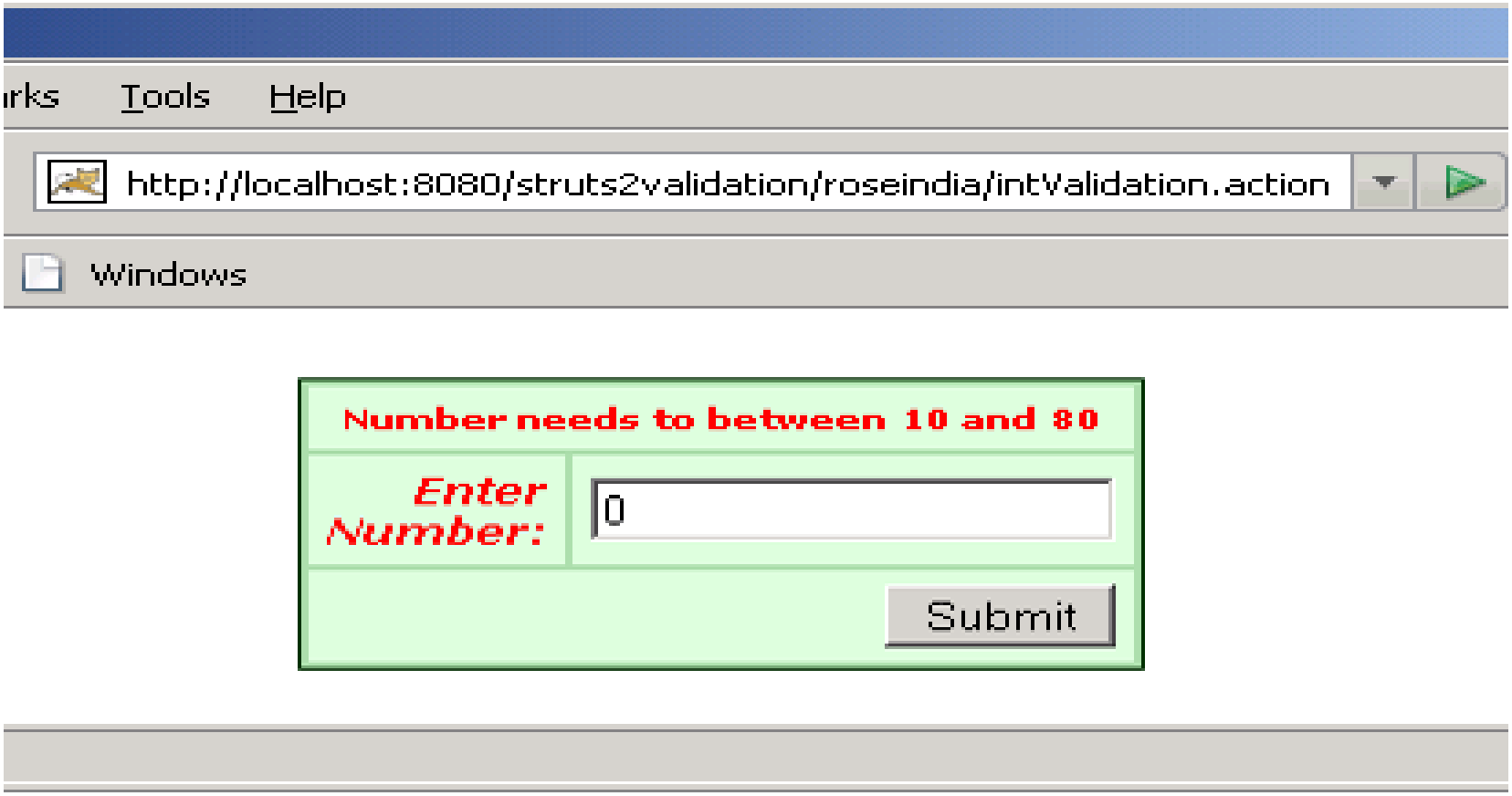
Using Struts 2 Int Validator

intSuccess.jsp

```
<%@page language="java" %>
<html>
<head> <title>Correct entry</title> </head>
<body>
<b>
Correct Input Number :</b>
<%=request.getParameter("userinput") %>
</body>
</html>
```

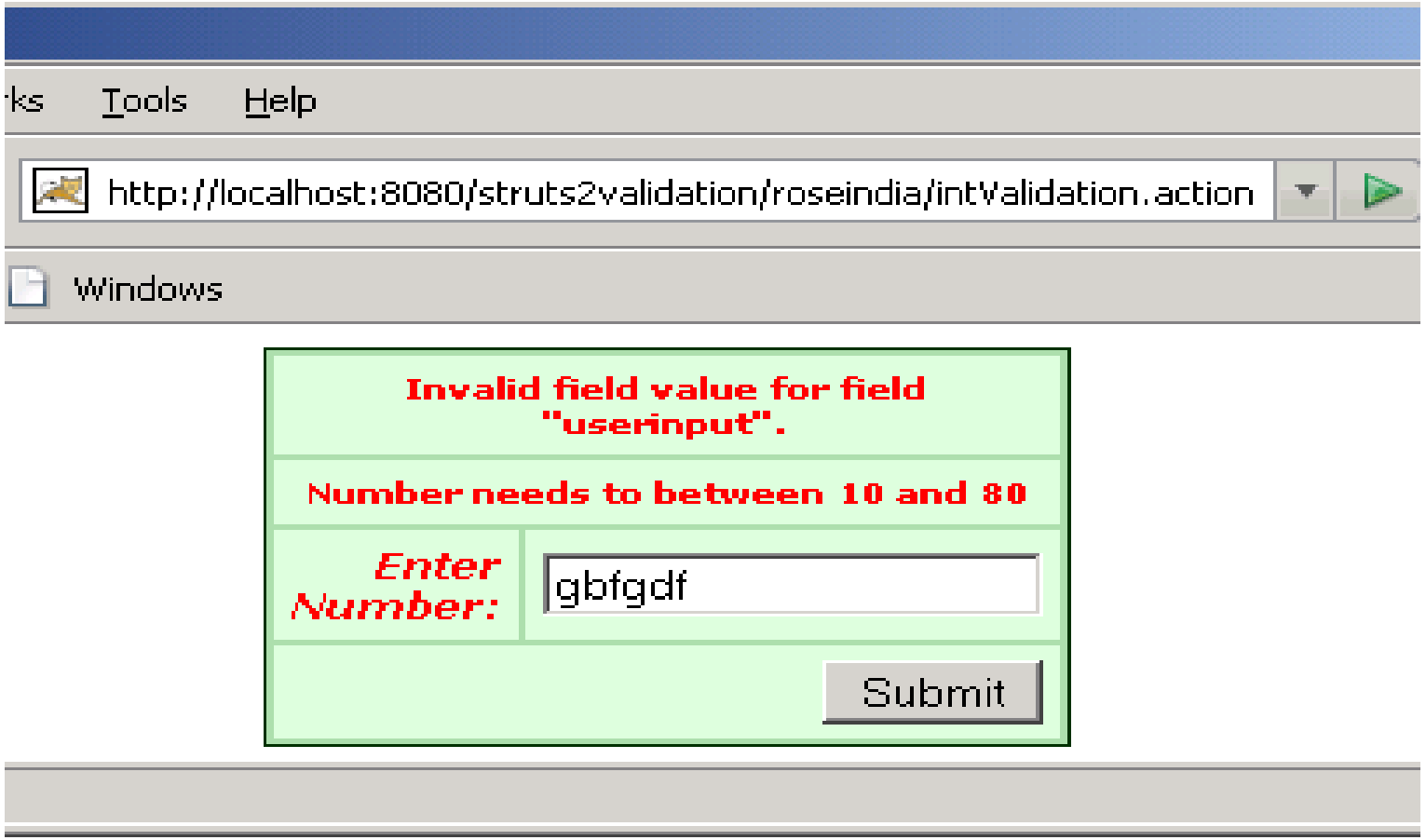
Using Struts 2 Int Validator

Output



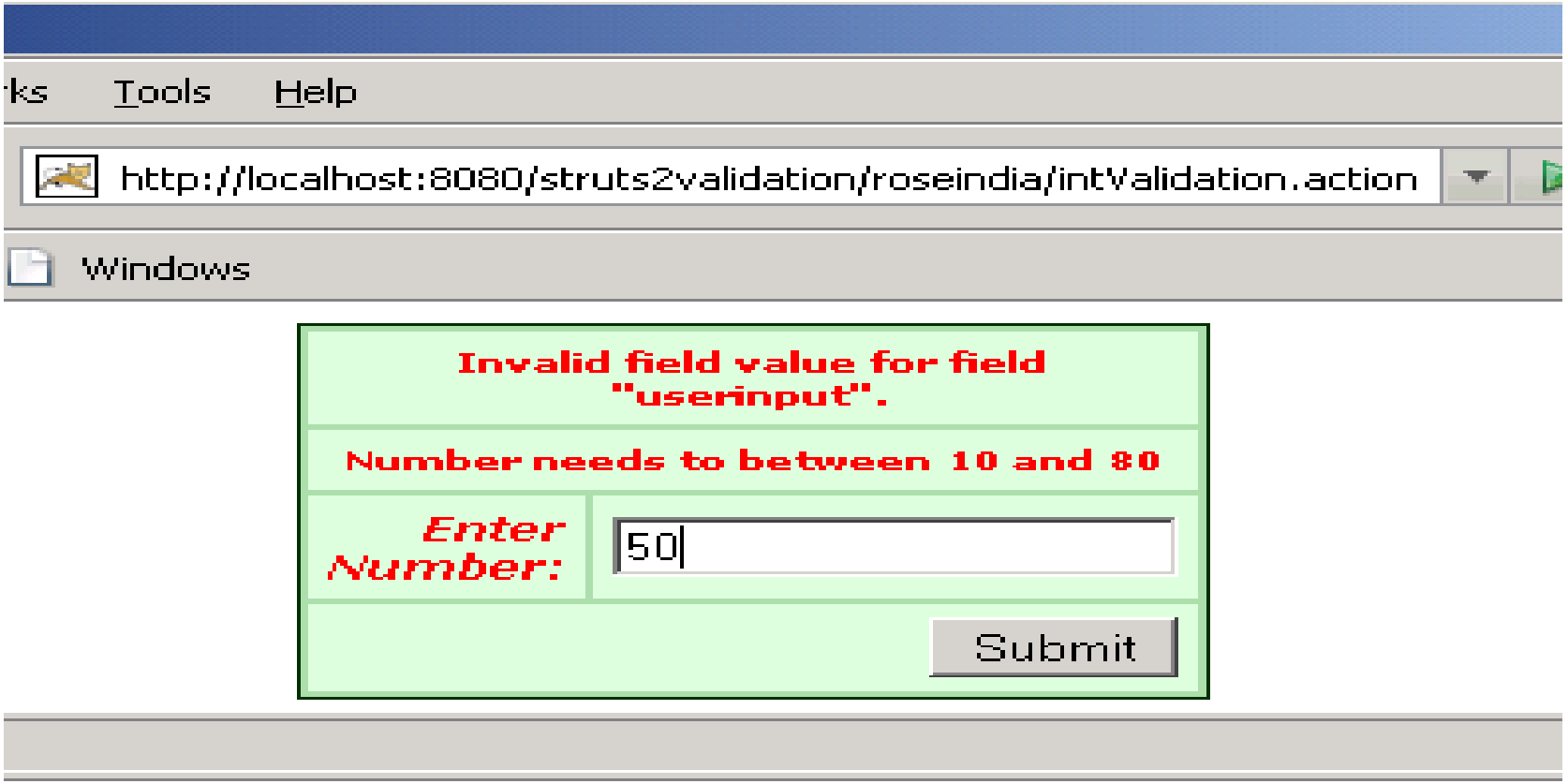
Using Struts 2 Int Validator

Output



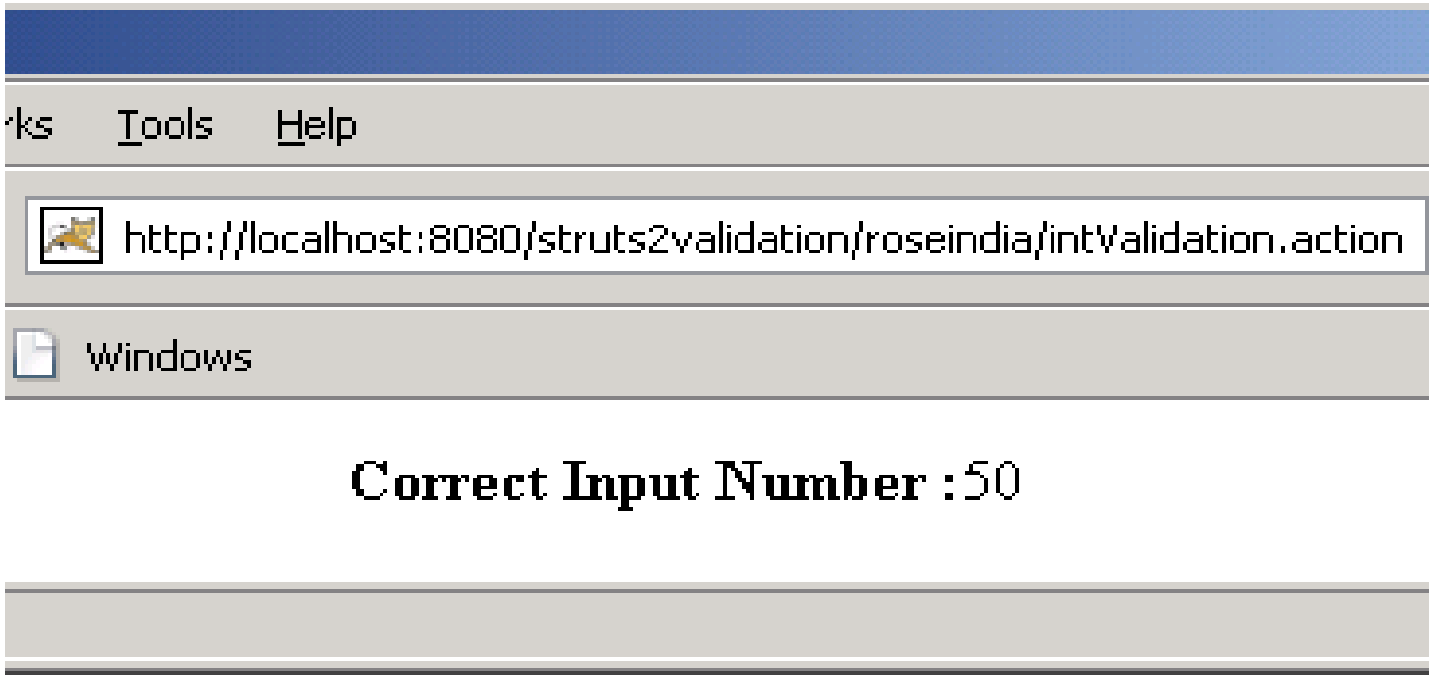
Using Struts 2 Int Validator

Output



Using Struts 2 Int Validator

Output



Writing a custom validator

- Writing your own custom validator is little different than writing any of the other custom Struts 2 components.
- For our example, we write a custom validator that checks for a certain level of password integrity.
- After we implement it, we add it to the Struts 2 Portfolio Register action to make sure that people are using strong passwords for their accounts.

Example : A custom validator to check password strength

- As with other custom components, a custom validator must **implement** a certain **interface**.
- In this case, all validators are obligated to implement the **Validator or FieldValidator interface**. The two interfaces, found in the **com.opensymphony.xwork2.validator** package, represent the two types of Validators as described earlier, field and nonfield.
- The framework also provides some convenience **classes** to make the task of writing custom validators all the more agreeable. Typically, you'll extend either **ValidatorSupport or FieldValidatorSupport**, both from the com.opensymphony.xwork2.validator.validators package.

Example : A custom validator to check password strength

- In our case, we extend the **FieldValidatorSupport class** because our validator, like most, operates on a given field.
- We design our password validator to make three checks:
  - The password must contain a letter, uppercase or lower.
  - The password must contain a digit, 0–9.
  - The password must contain at least one of a **set** of "special characters."

Example : A custom validator to check password strength (Validator Class)

```
public class PasswordValidator extends FieldValidatorSupport {
        static Pattern dp=Pattern.compile("[0-9]");
        static Pattern alpha =Pattern.compile("[A-Za-z]");
        static Pattern sp=Pattern.compile("[#$]");

        public void validate(Object o) throws ValidationException {
                String fieldname=getFieldName();
                String fieldvalue= (String)getFieldValue(fieldname, o);

                Matcher mdp=dp.matcher(fieldvalue);
                Matcher malpha=alpha.matcher(fieldvalue);
                Matcher msp=sp.matcher(fieldvalue);

                if(!mdp.find())  addFieldError(fieldname, o);
                if(!malpha.find())  addFieldError(fieldname, o);
                if(!msp.find())  addFieldError(fieldname, o);
        }
}
```

Example : A custom validator to check password strength (validators.xml)

```
<validators>

    <validator
        name="int"
        class="com.opensymphony.xwork2.validator.validators.IntRangeFieldValidator"/>
    <validator
        name="passwordvalidator"
        class="example.PasswordValidator"/> <!-- Custom Validator Mapping -->

</validators>
```

Example : A custom validator to check password strength(Action-validator.xml file)

```
<field name="password">
        <field-validator type="requiredstring">
            <message key="requiredstring"/>
        </field-validator>
        <field-validator type="stringlength">
            <param name="minLength">6</param>
            <param name="trim">true</param>
            <message key="requiredpassword"/>
        </field-validator>
        <field-validator type="passwordvalidator">
          <message> ALPHANUMERIC PASSWORD REQUIRED.</message>
        </field-validator>
    </field>
```