Developing Applications Using Java Web Frameworks

Session - 4

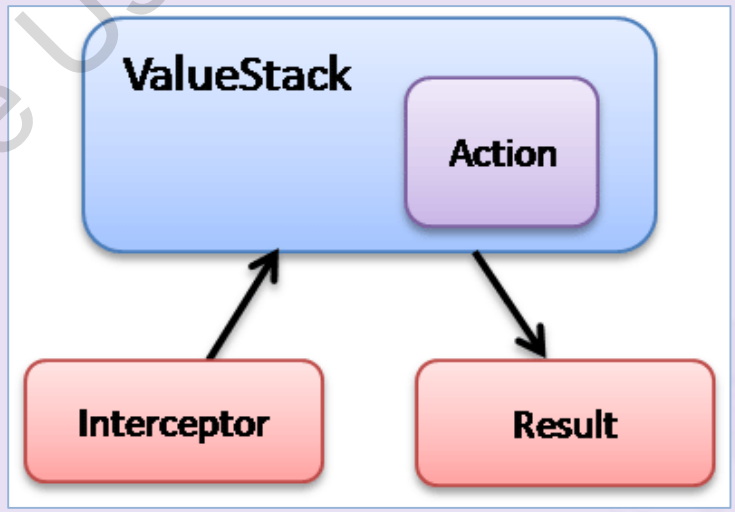
Struts 2 - OGNL, Validation, and Internationalization

Objectives

- Explain the use of ValueStack
- Explain OGNL expression language
- Explain the flow of data in and out of the Struts 2 framework
- Explain the use of converters and their types
- Explain Validator framework in Struts 2
- Explain different types of validators present in the Validator framework
- Explain the implementation of validation framework in Struts 2
- Explain internationalization
- Explain the use of i18N interceptor and resource bundle in internationalization
- Explain how to create the Struts 2 Web application with internationalization

Introduction 1-2

- ValueStack is a storage area that holds the data associated with the processing of a request.
- It is a stack of objects in the Struts 2 framework.
- All the core components interact with it to provide access to context information as well as elements of the execution environment.



Introduction 2-2

- The ValueStack is made up of the following objects:

Temporary Object

- It requires temporary storage during request processing. For example, current iteration value for a collection object.

Model Object

- It represents a place of the current object on the stack before the action is being executed.

Action Object

- It represents the action that is being currently executed.

Named Object

- It represents any object that is identified by an identifier.
- It can be developer created or existing such as `#application`, `#session`, `#request`, `#attr`, and `#parameters`.

ValueStack and OGNL

- **Object Graph Notation Language (OGNL)**
  - Allows the developer to refer and manipulate the data present on the ValueStack.
  - Helps in data transfer and type conversion between data forms and action class.
  - Provides a mechanism to navigate object graphs using dot notation and evaluate expression.

- For ValueStack, OGNL expression is tested at each level and a result is returned after evaluation of the expression.

OGNL 1-3

- It is integrated into the Struts 2 framework to provide data transfer and type conversion.
- It is an expression and binding language used for getting and setting the properties of the Java objects.
- It acts as a glue between the frameworks string-based input and output and the Java-based internal processing.
- It is a binding language between the GUI elements and the model objects.
- OGNL is an open source framework from Apache Commons project.
- Using OGNL, you can set and get properties from Java Beans.

OGNL 2-3

- OGNL can be used as:
  - A binding language between GUI elements.
  - A TypeConverter to convert values from one type to another.
  - A data source language to map between table columns and a Swing TableModel.
  - A binding language between Web components and the underlying model objects.
- The three different OGNL expression parts are as follows:
  - Property name, such as **name** in a JavaBean.
  - Method call such as `toString()` which returns the current object in a string format.
  - Array indices such as **`employee[0]`**, which returns the first value of the current object.

OGNL 3-3

- OGNL expressions are evaluated in the context of current object.
- Evaluating the expression chain, the previous link in the chain acts as the current object for the next one.
  - For example, consider the following expression and its order of evaluation:
    `Name.toCharArray()[0].numericValue.toString()`
- The steps followed to evaluate this expression are as follows:
  - The value stored in the name property of the object is extracted.
  - The `toCharArray()` method is invoked on the resulting `String`.
  - The first character from the resulting character array is extracted.
  - The numeric value is obtained from the character by invoking the `getNumericValue()` method of the `Character` class.
  - The `toString()` method is finally invoked on the Integer object to obtain the final result of the expression as `String`.
- The two most important parts of OGNL are expression language and type converters.

Expression Language

- OGNL's expression language is used in the form input field name and in JSP tags.
- Following code snippet shows the use of OGNL expression in Struts 2 tag:

```
...
<h5>Congratulations! You have successfully created
your login </h5>
<h3>The <s:property value="loginName" /> Login
Details </h3>
...
```

- In the code:
  - A congratulation message is displayed after successful creation of the login page.
  - The OGNL expression language is specified within the double quotes of the `value` attribute.
  - The `<s:property>` tag of Struts 2 takes the value from the property of one of the Java objects and writes it into the HTML in place of the tag.

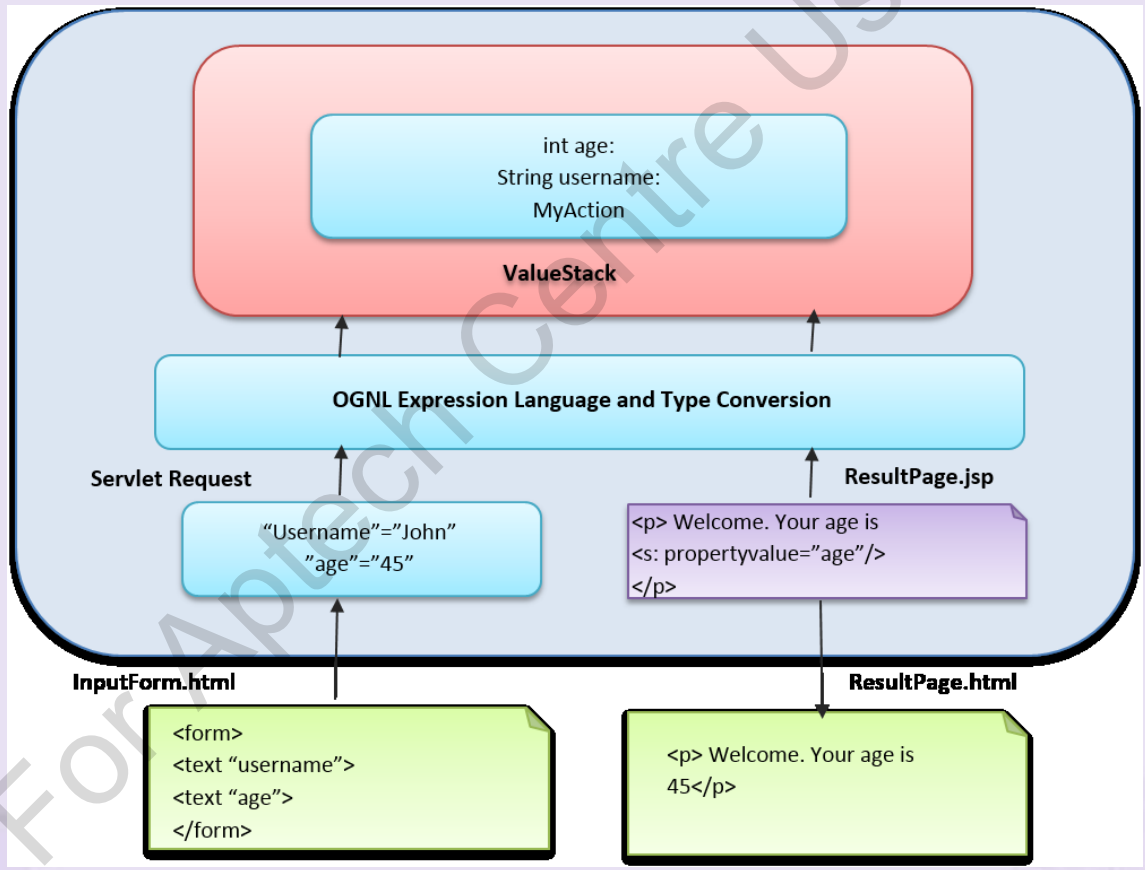
Type Converter 1-4

- Struts 2 framework provides built-in OGNL type converters.

- **Data Type Conversion**
  - Is made from the Java type of the property referred by the OGNL expression language to the string format of the HTML output.

Type Converter 2-4

- Following figure shows how data moves in and out of the Struts 2 framework, helps to bind and convert the data:



Type Converter 3-4

- The flow of data in and out of the Struts 2 framework is as follows:

Data Input

The request parameters are received by the framework as an `HttpServletRequest` object and are stored as name/value pairs.

The `param` interceptor transfers this data from the request object to ValueStack because the action object is present in the ValueStack.

Data Output

Once the data has been processed by the action, the result will be invoked.

When the rendering process of the result starts, the result will access the ValueStack using the OGNL expression language tags.

Type Converter 4-4

Data Input

The `params` interceptor interprets the parameter name as an OGNL expression to locate the correct destination property for the value.

If the property exists in the ValueStack, the data needs to be moved.

The data is moved to the property by invoking the setter method of the property.

Data Output

The tags will retrieve the value from the ValueStack by referring to each individual values with OGNL expressions.

Finally, the `Result.jsp` page displays the result by retrieving the value stored in the property using the `property` tag.

The value is converted again from the Java type of the property on the ValueStack to a string, so that it can be rendered as an HTML output.

Built-in Type Converters 1-2

- Struts 2 framework provides built-in converters for converting an HTTP string data type to any of the Java data types.

- Following table shows the different data types in Java that are supported by built-in converters:

Built-in Type Converters	Description
String	Represents a string data type.
boolean/Boolean	Converts true and false strings to primitive or object version of Boolean data type.
char/Character	Converts string to primitive or object version of character data type.
int/Integer	Converts string to primitive or object version of integer data type.
float/Float	Converts string to primitive or object version of float data type.
long/Long	Converts string to primitive or object version of long data type.

Built-in Type Converters 2-2

Built-in Type Converters	Description
double/Double	Converts string to primitive or object version of double data type.
Date	Converts string to SHORT format of current locale.
Arrays	Converts each string element to the array's type.
Lists	Populated with strings.
Maps	Populated with strings.

- The type converter does the data type conversion between the string-based HTTP form and the strictly typed Java objects.

Mapping Form Fields 1-12

- For automatic type conversion to take place, the developer needs to link the form field names with the Java properties of the action class and vice versa.
- This is a two-step process:

Developer writes the OGNL expression for the name attribute of the form field.

Developer creates the properties in Java code (Action class) that will receive the data.

- The different types of data type conversions are as follows:
  - **Primitive and Wrapper Class:**
    - The built-in conversion between the Java primitive and wrapper classes and OGNL expressions in the form fields are quite simple.

Mapping Form Fields 2-12

- Following code snippet shows the use of OGNL expressions in the form fields:

```
...
<h3> Registration Field </h3>
<s:form action="Register">
<s:textfield label="Username" name="username" />
<s:password label="Password" name="password" />
<s:textfield label="Enter your age...!" name="age" />
<s:textfield label="Enter your birthday.(mm/dd/yy)" name="birthdate"
/>
<s:submit/>
</s:form>
...
```

- In the code:
  - Each of the input field is an OGNL expression.
  - These expressions are resolved with respect to the action objects present in the ValueStack.

Mapping Form Fields 3-12

- Following code snippet shows the code in **`Register.java`**:

```
...
public class Register extends ActionSupport {
 private String username;
 private String password;
 private String portfolioName;
 private Double age;
 private Date birthdate;

@Override
 public String execute(){
 return SUCCESS;
}

} public String getPassword() {
 return password;
}
public void setPassword(String password) {
```

Mapping Form Fields 4-12

```
this.password = password;
}
public String getUsername() {  return username;
public void setUsername(String username) {  this.username = username;
}
public Double getAge() { return age;
}
public void setAge(Double age) {  this.age = age;
}
public Date getBirthdate() {  return birthday;
}
public void setBirthdate(Date birthday) {  this.birthday = birthday;
}
}
...
```

- In the code:
  - When the **Submit** button on the form is clicked, the request is sent to the framework as a name-value pair.
  - The framework places the `Action` object in the ValueStack.

Mapping Form Fields 5-12

- **Handling Multiple Values:**
  - Multiple parameter values coming from a request can be mapped to a single parameter name.
  - These multiple values can be mapped to Arrays, Lists, or Maps.

Array	Lists	Maps
• Struts 2 provides support for converting data to Java arrays. • OGNL executes the type conversion for each element of the array. • The framework provides unique names to OGNL, and are correctly interpreted as specific elements in an array.	• Struts 2 framework supports the conversion of sets of request parameter to properties of various collection types such as Lists. • Lists need not be initialized. • Elements in a List will be treated as String objects if the type is not specified.	• Struts 2 also support the conversion of a set of values from the request parameter to a Map property. • Map stores the value in a key-value pair. • The data types can be specified for the key-value pair in the Map.

Mapping Form Fields 6-12

- Following code snippet shows the use of the array in the struts form:

```
...
<s:form action="ArraysDataTransferTest">
<s:textfield name="marks" label="Marks"/>
<s:textfield name="marks" label="Marks"/>
<s:textfield name="marks" label="Marks"/>
<s:textfield name="names[0]" label="names"/>
<s:textfield name="names[1]" label="names"/>
<s:textfield name="names[2]" label="names"/>
<s:submit/> </s:form>
...
```

- In the code:
  - There are two arrays named, **marks** and **names**.
  - The two array properties named, **marks** and **names** will receive data from the first and the last three form fields respectively.

Mapping Form Fields 7-12

- Following table displays the request parameter name with its value:

RequestParameter Name	Parameter Value
`Marks`	75, 65, 55
`name[0]`	Angel
`name[1]`	Jessica
`name[2]`	John

Mapping Form Fields 8-12

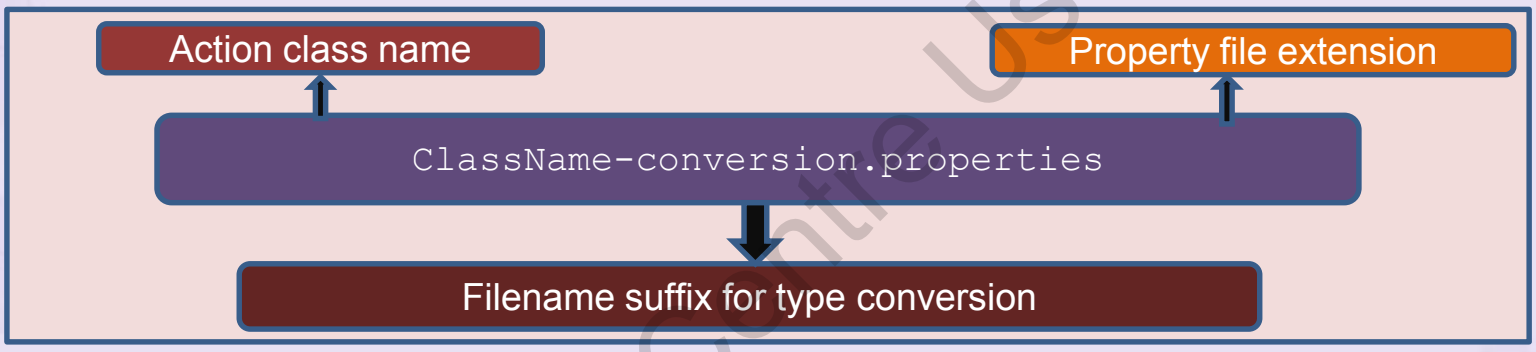
- Following code snippet shows the implementation of the properties in the action class that will receive the data:

```
...
private int[] marks ;
public int[] getMarks() { return marks;
}
public void setMarks(int[] marks){ this.marks = marks; }
private String[] names = new String[10];
public String[] getNames() { return names;
}
public void setNames(String[] names){ this.names = names; }
...
```

- In the code:
  - Arrays are displayed using the getter and setter methods.
  - The framework, when it transfers the **`marks`** property, first, searches for the **`marks`** property and resolves the mapping of the property with the request parameter.

Mapping Form Fields 9-12

- Following figure displays the creation of a properties file according to the naming convention:



- Following code snippet shows the JSP page containing the weights property, which will be used as a List in the action class:

```
...
<s:textfield name="weights[0]" label="weights"/>
<s:textfield name="weights[1]" label="weights"/>
<s:textfield name="weights[2]" label="weights"/>
...
```

Mapping Form Fields 10-12

- Following code snippet demonstrates the use of the `List` property in the `Action` class:

```
...
private List weights;
public List getWeights() {
return weights;
}
public void setWeights(List weight) {
this.weights = weight;
}
...
```

- If you are using generics to specify the Collection type, then the properties file configuration need not be used.
- With type specific List conversion, you cannot pre-initialize the List.

Mapping Form Fields 11-12

- Following code snippet demonstrates the implementation of the `Map` property in a JSP page:

```
...
<s:textfield name="maidenNames.mary" label="Maiden Name"/>
<s:textfield name="maidenNames.jane" label="Maiden Name"/>
<s:textfield name="maidenNames.hellen" label="Maiden Name"/>
<s:textfield name="maidenNames['beth']" label="Maiden Name"/>
<s:textfield name="maidenNames['sharon']" label="Maiden Name"/>
<s:textfield name="maidenNames['martha']" label="Maiden Name"/>
...
```

- In the code:
  - The specified key-value pair is of String data type.
  - The key-value pair can either be specified using the dot-notation expression or in an array format.
  - The **`maidenNames`** is the key and the different values such as **`mary`**, **`jane`**, and so on are the values.

Mapping Form Fields 12-12

- Following code snippet demonstrates the implementation of the `Map` property in the Java action page:

```
...
private Map maidenNames ;
public Map getMaidenNames(){
return maidenNames;
}
public void setMaidenNames ( Map maidenNames ) {
this.maidenNames=maidenNames;
}
...
```

- In the code:
  - The getter and the setter method for the map property have been defined.
  - You do not have to initialize the `Map` object explicitly.

Type Conversion Annotations

- The type conversion annotations are used for generics defined in Maps and Collections.
- It is configured in the ***-conversion.properties** file.
- Following table shows some of the type conversion annotations:

Annotation Name	Description
`CreateIfNull`	Checks if the new element should be created if it currently does not exist in the list or map.
`Type Conversion`	Determines the converter class to use. For collections, a rule of PROPERTY, MAP, KEY, KEY_PROPERTY, or ELEMENT can be used to specify which part to be converted.
`Element`	Used for generic type to specify the element type for Collection types and Map values.
`Key`	Used for generic types to specify the key type for Map keys.
`KeyProperty`	Used for generic types to specify the key property name value.

Validation 1-2

- User input needs to be validated so as to ensure that correct data has been entered as required by the application.
- One of the main features of the Struts 2 framework is its built-in validation support.
- **Validation Framework:**
  - The Struts 2 framework supports both server and client-side validation.
  - Validation is implemented by the framework using validation Interceptor which is configured in the default Interceptor stack.
  - Three layers present in the Validation Framework are Domain Data, Validation MetaData, and Validators.

Validation 2-2

Domain Data	Validation Metadata	Validators
• The data entered by the user using a browser exists as properties in an action class in Struts 2. • For example, in the **Login Form**, a user is required to enter username and password.	• This component will associate the individual data properties with the validators which are used for verifying the correctness of the values during runtime. • Mapping of validators to data properties can be done using Java annotations or XML files.	• Validators can be defined as a reusable component that contains the logic for performing fine-grained validations. • The validators have to be associated with the properties either through an XML file or through Java annotations.

Working of a Validator 1-2

- An action class extends `ActionSupport` class which implements the `Validateable` and `ValidationAware` interfaces of the `com.opensymphony.xwork2` package.
- **Validateable Interface:**
  - The `validate()` method in the `Validateable` interface contains the validation code.
  - The validator interfaces work with workflow Interceptor.
  - Once the execution of the `validate()` method is completed, the workflow Interceptor will invoke the `hasErrors()` method of the `ValidationAware` interface.
- **ValidationAware Interface:**
  - The `ValidationAware` interface contains methods for storing the error messages generated when validation of a property fails.

Working of a Validator 2-2

- Following code snippet shows the sequence of Interceptors from the `defaultStack` as defined in `struts-default.xml` file:

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

- In the code:
  - The `params` and `conversionError` Interceptor are responsible for transferring and converting the values in the HTTP request parameter to correct Java types.
  - Both these Interceptors are fired before the validation Interceptors are fired.
  - The `validation` Interceptor is used for making an entry into the validation framework.

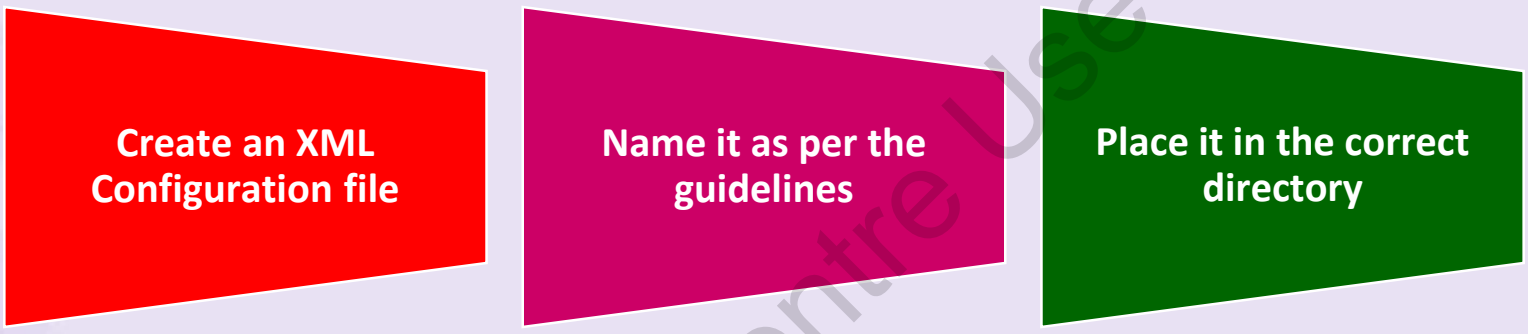
Validation Scope 1-4

- The two types of validators present in the Validator Framework are as follows:

Field Validators	Non-Field Validators
• These validators operate on a single field accessible through an action. • It can perform validations in the full action context involving more than one field in validation rule. • It can be defined on each field. • There can be more than one validator on a single field. • The name of the validator file is derived from the name of the class that implements the action for which the validation rules apply.	• These are not restricted to a single specific field. • It applied to the whole action and often contain checks that apply to more than one field values. • The `expression` validator is the only non-field validator available in the built-in validators. • Complex validation logic can be written in the expression validator using OGNL EL.

Validation Scope 2-4

- Steps for applying pre-defined validators:



- The naming rule for the XML validation metadata file is **ActionClass-validation.xml**.
- The action class is named as **`Registration`**, then the validation XML file is named as **`Registration-validation.xml`**.
- The file is placed in the package directory structure next to the action class and each `<field-validator>` is executed in the order of definition.

Validation Scope 3-4

- Following code snippet demonstrates the implementation of validators in an XML file:

```
...
<!DOCTYPE validators PUBLIC
 . . .
<validators>
<field name="username">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message key="errors.required" />
</field-validator>
</field>
<field name="password">
<field-validator type="requiredstring">
<param name="trim">true</param>
<message key="errors.required" />
</field-validator>
</field>
```

Validation Scope 4-4

```
<field name="age">
<field-validator type="required">
<message key="errors.required" />
<!-- <message> You must enter age </message> -->
</field-validator>
<field-validator type="int">
<param name="min">1</param>
<param name="max">100</param>
<message key="errors.range"/>
</field-validator>
</field>
...
</validators>
```

- In the code:
  - The `doctype` element includes all the validation XML files.
  - The `<validators>` element includes the declarations of the individual validators that should run when this action is invoked.

Built-in Validators 1-2

- Following table displays a list of built-in validators:

Validator Name	Parameters	Description
`required`		Checks that the value is not null.
`requiredstring`	`trim`	Checks that the value is not null and is not empty. The default value for the attribute trim is true indicating that it trims white space.
`double`	`minInclusive`, `maxInclusive`, `minExclusive`, `maxExclusive`	Checks that the double value is within the inclusive or exclusive specified parameters.
`int`	`min`, `max`	Checks that the integer value is within the value specified by min and max.
`email`	-	Checks the format of the email address.
`url`		Checks the URL format.

Built-in Validators 2-2

Validator Name	Parameters	Description
`stringlength`	`trim, minlength, maxlength`	Checks that the length of the string is within the parameters specified by minlength and maxlength.
`date`	`min, max`	Checks that the date value is within the specified date as specified by min and max attribute. Format of the date value is MM/DD/YYYY.
`regex`	`expression(), caseSensitive, trim`	Checks that the string confirms to the regular expression.
`expression`	`expression()`	Used at action level and is same as fieldexpression.

Using a Validator 1-9

- To demonstrate the use of Validators, a **NewsletterRegistration** application is created in NetBeans.
- Modify the default `index.jsp` page to contain a link to `NewsLetterRegistration.jsp` page.
- Create a registration form `NewsLetterRegistration.jsp` that accepts user details such as `name`, `password`, `age`, `email`, and `telephone number`.
- The steps are as follows:
  - Creation of the JSP page
  - Creation of an Action class
  - Declaration of the validation metadata with ActionClass-validations.xml
  - Creation of Properties file

Using a Validator 2-9

- Following code snippet demonstrates the code for **NewsLetterRegistration.jsp** page:

```
...
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>NewsLetter Registration Page</title>
</head> <body> <h1>News Letter Registration Page!</h1>
<s:form action="NewsLetter" method="post" >
<s:textfield name="username" key="username" size="20" />
<s:textfield name="password" key="password" size="20" />
<s:textfield name="age" key="age" size="20" />
<s:textfield name="email" key="email" size="20" />
<s:textfield name="telephone" key="telephone" size="20" />
<s:submit /> </s:form> </body>
</html>
...
```

- In the code:
  - Five text fields have been defined to accept the username, password, age, email address, and telephone number respectively.

Using a Validator 3-9

- Following code snippet shows the code for the action form:

```
...
public class NewsLetter extends ActionSupport {
public String execute() throws Exception {
return SUCCESS;
}
/**
* Provide default value for Message property.
*/
public static final String MESSAGE = "HelloWorld.message";
/**
* Field for Message property.
*/
 private String message;
* Return Message property.
* @return Message property
*/
public String getMessage() {
return message;
```

Using a Validator 4-9

```
}
public void setMessage(String message) {
this.message = message;
}
private String username;
private String password;
private String telephone;
private String email;
private int age;
public String getUsername() {
return username;
}
public void setUsername(String userName) {
this.username = userName;
}
public String getPassword() {
return password;
}
public String getTelephone() {
return telephone;
}
public void setTelephone(String telephone) {
```


Contd.

Using a Validator 5-9

```
this.telephone = telephone;
}
public String getEmail() {
return email;
}
public void setEmail(String email) {
this.email = email;
}
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
}
...
```

- In the code, the action class receives the data from the JSP page and sets the property value.

Using a Validator 6-9

- Following code snippet shows the NewsLetter action's validation metadata file, **NewsLetter-validation.xml:**

```
...
<!DOCTYPE validators PUBLIC

<validators> <field name="username">
<field-validator type="requiredstring">
  <param name="trim">true</param>
  <message key="errors.required" />
</field-validator> </field> <field name="password">
<field-validator type="requiredstring">
   <param name="trim">true</param>
  <message key="errors.required" /> </field-validator>
</field> <field name="age">
<field-validator type="required"> <message key="errors.required" />
<!-- <message> You must enter age </message> -->
</field-validator>
<field-validator type="int">
<param name="min">1</param>
<param name="max">100</param>
<message key="errors.range"/>
```

Using a Validator 7-9

```
</field-validator>
</field>
<field name="email">
<field-validator type="requiredstring">
<message key="errors.required" />
</field-validator>
<field-validator type="email">
<message key="errors.invalid" />
</field-validator>
</field>
<field name="telephone">
<field-validator type="requiredstring">
<message key="errors.required" />
</field-validator>
</field>
</validators>
...
```

- In the code:
  - Name of the file is derived from the name of the class that implements the action for which the validation rules apply.
  - The file is placed in the package directory next to the action class.
  - The `doc` type element must be included in the validation XML file.

Using a Validator 8-9

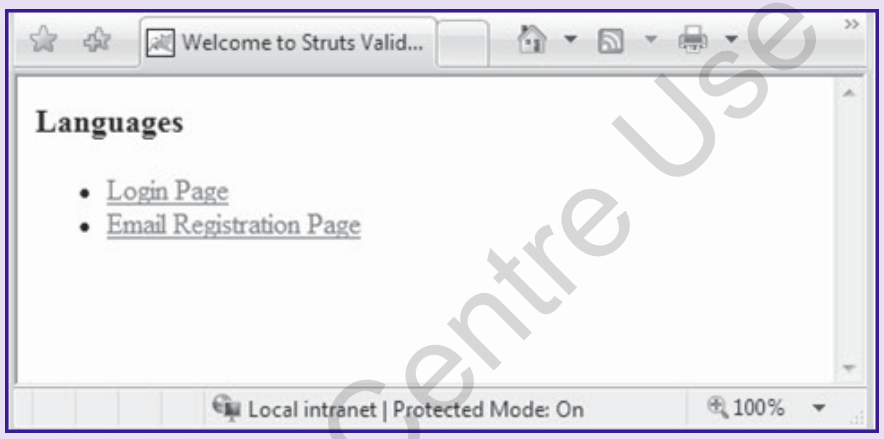
- Following code snippet demonstrates the implementation of **`properties`** file:

```
HelloWorld.message= Welcome to Struts Validation ...
username=Username
age=Age
email=Email
telephone=Telephone
password=Password
errors.invalid=${getText(fieldName)} is invalid.
errors.required=${getText(fieldName)} is required.
errors.number=${getText(fieldName)} must be a number.
errors.range=${getText(fieldName)} is not in the range ${min} and ${max}.
```

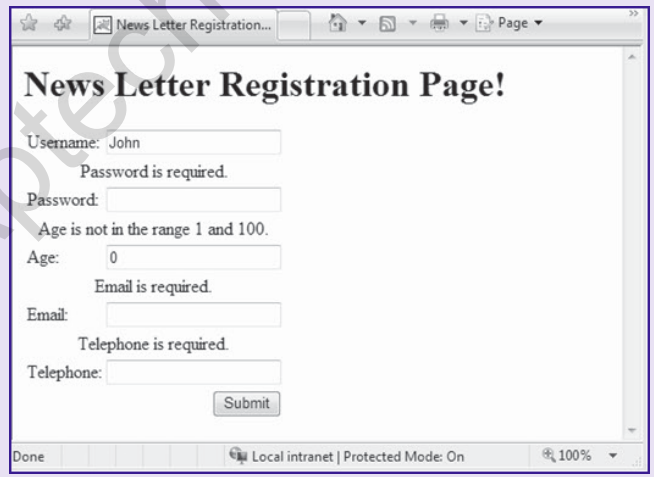
- In the code:
  - The different error messages are stored for each of the text fields to be validated.
  - For example, if the `requiredstring` validator encounters a null value for the `username` property of the action class, the respective error message is retrieved and displayed.

Using a Validator 9-9

- Following figure shows the application startup page:



- Following figure displays the error page showing the validation messages:



Creating a Custom Validator

- A custom validator must implement the `Validator` or `FieldValidator` interface.
- These custom validator classes normally extend either the `ValidatorSupport` or the `FieldValidatorSupport` classes.

Validation Annotation 1-2

- Validation annotation is used for configuring custom validators and to group validators for a property or a class.
- Following table describes the different validation annotations:

Annotation Name	XML Equivalent	Description
`RequiredFieldValidator`	Required	Ensures that the property is not null.
`RequiredStringValidator`	Requiredstring	Ensures that the String property is not null or empty.
`StringLengthFieldValidator`	stringlength	Checks that the String length is within the specific range.
`IntRangeFieldValidator`	Int	Checks that the integer property is within the specific range.

Validation Annotation 2-2

Annotation Name	XML Equivalent	Description
`DateRangeFieldValidator`	Date	Checks that the date property is within the specific range.
`ExpressionValidator`	Expression	Evaluates the OGNL expression using the ValueStack.
`FieldExpressionValidator`	FieldExpression	Validates a field using the OGNL expression.
`EmailValidator`	Email	Ensures that the property is a valid email address.
`UrlValidator`	url	Ensures that the property is a valid URL.
`RegexFieldValidator`	Regex	Checks whether the property matches the regular expression.
`Validation`		Signifies that the class using annotation based validation can be used on interfaces or classes.
`Validations`		Groups multiple validations for a property or a class.

Internationalization in Struts 2 1-3

- **Internationalization**
  - Is the technique for developing applications that support multiple languages and data formats without the need to re-engineer the application.
  - Sometimes, the term 'Internationalization' is abbreviated as `I18N`, because there are 18 letters between the first 'I' and the last 'N.'
- Following table shows the list of few country and language codes:

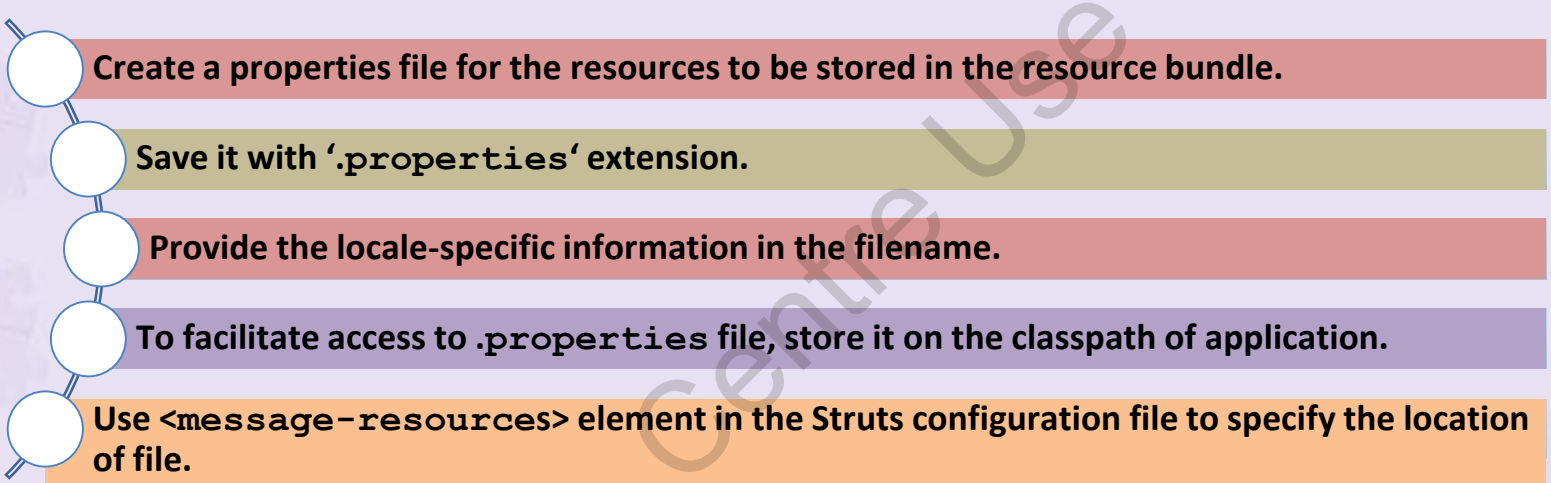
Country Name	Country Code	Language	Language Code
Italy	IT	Italian	it
China	CN	Chinese	zh
France	FR	French	fr
Germany	DE	German	de
Japan	JP	Japanese	ja
Spain	ES	Spanish	es
United States	US	English	en

Internationalization in Struts 2 2-3

- Struts 2 supports internationalization through:
  - il8n Interceptors
  - Message Resource Bundles
  - Tag Libraries

Internationalization in Struts 2 3-3

- Steps to use message resources are as follows:



- Following code snippet shows the use of `text` tag to retrieve the message from the resource bundle:

```
<s:text name="some.key" />
```

Implementing Internationalization 1-5

- Following code snippet creates the common `index.jsp` page which will be displayed based on the selected language:

```
. . .
<%@ taglib prefix="s" uri="/struts-tags"%>
. . .
<body>
  <h1><s:text name="global.heading"/></h1>
  <s:url id="indexEN" namespace="/" action="locale" >
     <s:param name="request_locale" >en</s:param>  </s:url>
 <s:url id="indexFR" namespace="/" action="locale" >
    <s:param name="request_locale" >fr</s:param>
  </s:url>
<s:a href="%{indexEN}" >English</s:a>
<s:a href="%{indexFR}" >France</s:a> <s:form action="login" method="post"
namespace="/">
     <s:textfield name="name" key="global.name" size="20" />
     <s:textfield name="password" key="global.password" size="20" />
     <s:submit name="submit" key="global.submit" />
  </s:form>
</body>
```

- In the code, the Struts 2 `text` tag retrieves a `ResourceBundle` message based on the key passed into the `name` attribute.

Implementing Internationalization 2-5

- Following code snippet shows the different resource bundle properties created for different languages:

```
# global.properties
global.name = Name
global.password = Password
global.submit = Submit
global.heading = Select Locale
global.success = Successfully authenticated

#global_fr.properties
global.name = Nom d'utilisateur
global.age = l'pâssws
global.submit = Soumettre des
global.heading = Sé lectionnez Local
global.success = Authentifié  avec succès
#global_fr.properties
global.name = Nom d'utilisateur
global.age = l'pâssws
global.submit = Soumettre des
global.heading = Sé lectionnez Local
global.success = Authentifié  avec succès
```

Implementing Internationalization 3-5

- Following code snippet shows the configuration of resource bundle in the `struts.xml` file:

```
. . .
<struts>
  <constant name="struts.custom.i18n.resources" value="global" />

  <package name="example" extends="struts-default" namespace="/">
     <action name="login"
        class="com.example.LoginAction”
        method="execute">
        <result name="input">/index.jsp</result>
<result name="success">/success.jsp</result>
     </action>

     <action name="locale"
        class="com.example.Locale" method="execute">
        <result name="success">/index.jsp</result>
     </action>
  </package>
</struts>
```

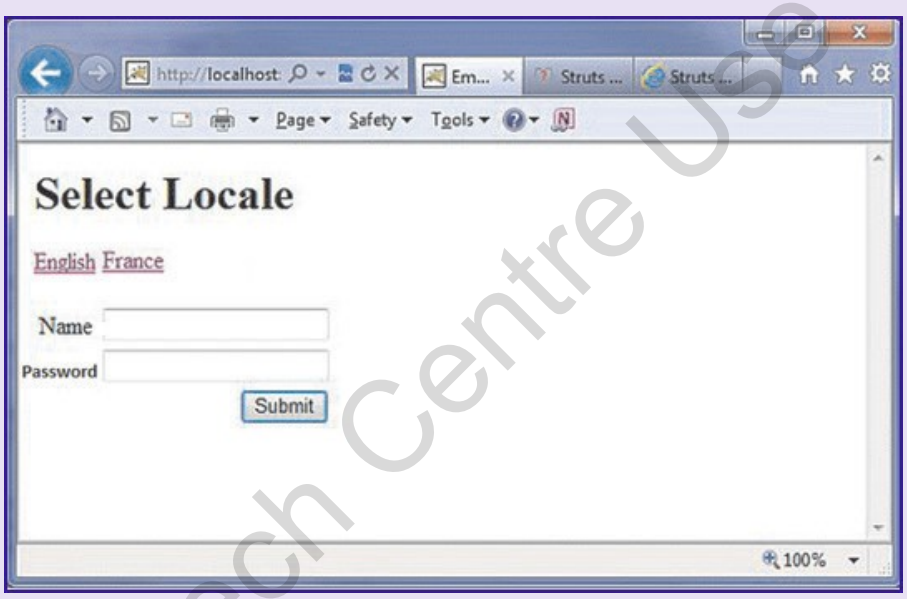
Implementing Internationalization 4-5

- Following code snippet shows the `LoginAction` and `Locale` class:

```
/*
* LoginAction.java class */
public class LoginAction{
  private String name;
  private String password;
  public String execute()
  {   return SUCCESS;   }
   // getter and setter methods
  . . .
}
/*
* Locale class */
public class Locale extends ActionSupport{
  public String execute()
  {        return SUCCESS;    }
}
```

Implementing Internationalization 5-5

- Following figure shows the output of the code:



Lifecycle Callback Annotations 1-4

- They are invoked at a specified time during the processing of an action.
- There are three lifecycle callback annotations:

`@Before`

- It marks an action method that should be executed before the main action method.
- It is used at the method level.
- It helps in pre-processing of common tasks.

`@After`

- Marks an action that should be executed after the logic in the main action method.
- Result is executed but before the result is returned to the user.
- Applicable at the method level.

`@BeforeResult`

- It invoked after the method that executes the logic for the action but before the result is executed.
- The return value is ignored.
- This annotation is applied at the method level.

Lifecycle Callback Annotations 2-4

- Following code snippet shows the use of the `@Before` annotation:

```
...
public class SampleAction extends ActionSupport {
@Before
public void isAuthorized() throws AuthenticationException {
// authorize request, throw exception if failed
}
public String execute() {
 // perform secure action
 return SUCCESS;
}
}
...
```

Lifecycle Callback Annotations 3-4

- Following code snippet shows the use of the `@After` annotation:

```
...
public class SampleAction extends ActionSupport {
@After
public void isValid() throws ValidationException {
// validate model object, throw exception if failed
}
 public String execute() {
// perform action
 return SUCCESS;
 }
}
...
```

- In the code, the `isValid()` method will be executed after the `execute()` method.

Lifecycle Callback Annotations 4-4

- Following code snippet shows the use of the `@BeforeResult` annotation:

```
...
public class SampleAction extends ActionSupport {
@BeforeResult
public void isValid() throws ValidationException {
 // validate model object, throw exception if failed
}
public String execute() {
 // perform action
return SUCCESS;
}
}
...
```

- In the code, the `isValid()` method will be executed after the `execute()` method but before the execution of the result.

Summary

- OGNL is integrated into the Struts 2 framework to provide data transfer and type conversion.
- OGNL's expression language is used in the form input field name and in JSP tags.
- Struts 2 framework provides built-in converters for converting an HTTP string data type to any of the Java data types.
- Two types of validators present in the Validator Framework are Field Validators and Non-Field Validators.
- One of the main features of the Struts 2 framework is its built-in validation support.
- The validate() method in the Validateable interface contains the validation code whereas the ValidationAware interfaces contains methods for storing the error messages generated when validation of a property fails.
- Two types of validators present in the Validator Framework are Field Validators and Non-Field Validators.
- Struts 2 Web application support internationalization.
- Struts 2 supports internationalization through interceptors, resource bundles, and tag libraries, in the Web application.
- Life cycle callback annotations are invoked at a specified time during the processing of an action.