Transform & Execute Apache Struts 1.x based Validations to Bean Validation through JSF

Vijay Kumar Pandey
Director of Technology Solutions, Intueor Consulting, Inc.
Irvine, CA (United States of America)

Abstract.
The paper is intended to provide an understanding of how to transform and execute Apache Struts 1.x based validation to Bean Validation and then execute the transformed validate method through Java Server Faces (JSF) runtime. JSF by default executes the validation after the APPLY_REQUEST_VALUES phase and if there are no validation issues then the control passes to UPDATE_MODEL_VALUES phase else it goes to the RENDER_RESPONSE phase. While in the Struts 1.x application, first the Struts action form is populated with the request parameter values and then only its ‘validate’ method is executed. So direct mapping of struts validation execution process will fail in JSF since model (action form) will not be populated with the request parameters when the validation is executed.

Keywords – Struts, JSF, ‘validate’ method, Bean Validation (BV), OmniFaces, ThreadLocal, ActionForm, ControllerBean, TagHandler.

I. INTRODUCTION

Struts based validation of the action form is defined in the class ValidatorForm and executed through the validate method public ActionErrors validate (Action Mapping mapping, HttpServletRequest request). The implementation here executes all the configured struts-based validation on various properties of the form. Developers must execute this super method from their respective action forms validate method. Most of the time application will also have various non-configured validations that will be present in the validate method. For every validation failure in struts, an ActionError will be created and stored in request/response for it to be displayed during the rendering of the page. If the transformed JSF application from Struts transforms the struts-based validation config to Bean Validation, based constraint annotations and would like to execute the transformed validate method in JSF only after the UPDATE_MODEL_VALUES phase, then the default lifecycle of JSF should be changed to provide this feature. JSF is a feature rich framework and this unique problem can be solved through the usage of a JSF based tag handler. A JSF based tag handler can execute the transformed validate method after the UPDATE_MODEL_VALUES phase to ensure that the action form bean (will call controller bean in JSF) is populated with the request parameters. The downside to this is it will be populated with the invalidated data. Developers need to ensure that their forms do get validated before the data is sent for further processing through the INVOKE_APPLICATION phase.

II. STRUTS VALIDATE TO JSF VALIDATE

Most of the struts-based validators are based on the open source project Apache Commons Validator. A standard Struts required validator maps to Bean Validator’s NotNull constraint. For better understanding, a validated struts action form UserForm is created with a String property firstName, which will be configured with the required validation.

A. Struts based UserForm with firstName

```java
public class UserForm extends ValidatorForm {

    private String firstName;

    public String getFirstName(){
        return firstName;
    }

    public void setFirstName(String firstName){
        this.firstName = firstName;
    }

    public ActionErrors validate(ActionMapping mapping, ServletRequest request) {
        ActionErrors errors = super.validate(mapping, request);
        if (errors.size() > 0) {
            return errors;
        }
        //some other specific validations
    }
}
```

B. Struts Validation Configuration

```xml
<form name="userform">
    <field property="firstName" depends="required">
        <msg name="required" key="error.required.firstName"/>
    </field>
</form>
```

The userform above provides the struts validation configuration, based on the validate method. The code super.validate(mapping, request) will make use of the validation config and if any validation error, it gets added to the ActionErrors.
C. JSF based User Controller Bean with first Name

Instead of calling this class as a form, which was mainly a Struts way of naming them, let’s call this form class as UserControllerBean. This will extend from a base class, that can provide all the common functionality and the implementation of the Bean Validation.

public class UserControllerBean extends BaseControllerBean {
    @NotNull(message = "{error.required.firstName}")
    private String firstName;

    public String getFirstName(){
        return firstName;
    }

    public void setFirstName(String firstName){
        this.firstName = firstName;
    }

    public void validate() {
        FacesContext facesContext = FacesContext.getCurrentInstance();
        super.validate();
        if (facesContext.isPostback()) {
            return;
        }
    }
}

In the above class, the field firstName is annotated with theNotNullBean Validation constraint annotation, which is likerequiredvalidation from Struts. The validate method has been transformed by removing all the Struts based parameters and managing any request/response-based invocation throughFacesContext. In the super class, validate method will be implemented to provide Bean Validation based implementation.

D. Bean Validation JSF based implementation

Below is a code excerpt on how to take aControllerBean and validate the bean validation constraint annotations annotated on the field or the property.

```java
javax.validation.Validator validator = // fetch the validator
for (String property : propertyList) {
    Set violationsRaw = validator.validateValue(this.getClass(),
    property, propertyValue, beanValidatorGroup);
    //add the violations as JSF error message
}
```

III. JSF TAG HANDLER MECHANISM TO CHANGE LIFECYCLE

Transforming Struts based validation to Bean Validation and executing it through JSF runtime, will require execution of validate method of theControllerBean, after it has been populated with the request parameters. To ensure this, validate method needs to be executed after theUPDATE_MODEL phase. The section below will describe how to set up this mechanism in JSF.

A. Stopping Bean Validation during PROCESS_VALIDATIONS

A JSF based tag handler will be designed to manoeuvre the mechanism of moving thePROCESS_VALIDATIONSPhase after theUPDATE_MODEL_VALUESphase.

```java
public class ValidateTagHandler extends TagHandler {

    This being a tag handler for processing validation mainly on html form submission, it should only be set up during postBack processing for theRESTORE_VIEWphase.

    if (! (ComponentHandler.isNew(parent) && facesContext.isPostback()) && facesContext.getCurrentPhaseId() == RESTORE_VIEW)) {
        return;
    }
}

Once theRESTORE_VIEWphase has completed, all the JSF components will be set up. So During the tag handler processing, add a phase listener that will be executed after theRESTORE_VIEWphase is completed. The main operations that will happen in this dynamic phase listener are

- Remove the JSF basedBeanValidator from the JSF component or else the validation will get invoked during thePROCESS_VALIDATION

```java
Validator[] validators = component.getValidators();
for (Validator validator : validators) {
    if (validator instanceof BeanValidator) {
        return (BeanValidator) validator;
    }
}

//create a dummy validation group
private interface NoValidationGroup {

    //set a dummy validation group so that actual validation does not invokes
    String nonValGroup = NoValidationGroup.class.getName();

    //keep the original validation group – that will be reset after this phase
    beanValidator.setValidationGroups(nonValGroup);
}

- Add a new JSF validatorCollect Submit Val Validator to the JSF component that basically collects the submitted and converted value through itsvalidate method but doesn’t perform the actual validation. This collection of submitted/converted value will later be validated through Bean Validation.

    component.addValidator(new <<SomeValidator to extract submitted value>>);

Both above processes of removing the bean validation by adding a dummy validation group, along with adding a new validator to collect the submitted and converted values, must be set to be executed through a phase listener. This should be invoked before thePROCESS_VALIDATIONSPhase.
A new phase listener should also be setup to be invoked after the PROCESS_VALIDATIONS phase, to ensure to reset the original validation group related to bean validation and remove the new validator CollectSubmitValValidatdor from the component that was added to collect the submitted values.

```java
//restore the original bean validation group
BeanValidator beanValidator = getBeanValidator(component);
if (beanValidator != null) {
    String originalValiGrp = component.getAttributes().remove("original_bean_val_group");
    beanValidator.setValidationGroups("original_bean_val_group");
}

//remove the ‘CollectSubmitValValidatdor’
EditableValueHolder valueHolder = //reference component
Validator colValidator = null;
for (Validator validator : valueHolder.getValidators()) {
    if (validator instanceof CollectSubmitValValidatdor) {
        colValidator = validator;
        break;
    }
}
if (colValidator != null) {
    valueHolder.removeValidator(colValidator);
}
```

**B. Execution of validate method**

In the same tag handler, a new phase listener must be added that will invoke the actual validate method on the controller bean. This phase listener should be added after the UPDATE_MODEL_VALUES phase.

```java
//find the actual ‘validate’ method of the controller bean
Method method = //either through reflection or some reflection lib
    method.invoke(controllerBean);
//If any exception in the above invocation
facesContext.validationFailed();
facesContext.renderResponse();
```

**C. Omni Faces**

Omni Faces is a JSF utility library and its contribution to the JSF world is immense. This paper’s tag handler concept is based on the OmniFacesValidateBean tag handler.

**D. Test User.xhtml**

The below code provides the usage of the tag handler (tag: valTag) in an xhtml page that will invoke the validate method on the UserControllerBean

```java
<omnifaces:valTag name="f:firstName"/>
```

**E. User Controller Bean**

The code in the bean below shows the validate method and annotated firstName field with the NotNull constraint annotation and shows a manual custom error condition.

```java
public class UserControllerBean extends BaseControllerBean implements Serializable{
    private String firstName;
    public String getFirstName() { return firstName; }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public boolean validate() {
        if (firstName == null || firstName.isEmpty()) {
            return false;
        }
        //add custom error handling
        if (firstName.equals("John Doe")) {
            String errorMessage = "User First name cannot be John Doe!";
            facesContext.addMessage(new FacesMessage(errorMessage, FacesMessage.SEVERITY_ERROR, errorTag, errorTag));
            facesContext.validationFailed();
        }
    }
}
```

**F. Base Controller Bean**

This bean can be used as a super class for all the controller beans (basically JSF’s counterpart of the struts ActionForms). This class will provide the base implementation of the Bean Validation and add the constraint violations as jsf error messages.
package test;

import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Set;
import javax.faces.application.FacesMessage;
import javax.faces.componentEditableValueHolder;
import javax.faces.context.FacesContext;
import javax.validation.ConstraintViolation;
import javax.validation.Path;
import javax.validation.PathHelper;
import javax.validation.Validator;
import javax.validation.groups.Default;

public class BaseControllersBean {
	public static final Class<?>[] DEFAULT_VALIDATION_GROUPS = new Class[] { Default.class };
	private void validate() {

		FacesContext faceContext = FacesContext.getCurrentInstance();
		javax.faces.application.FacesMessage validator = (Validator) faceContext.getExternalContext().
			getApplicationMap().get("java.faces.validator.BeanValidator.ValidatorFactory");
		ValidationContext valContext = ValidationContextHolder.getInstance().
			getBeanValidationContext();


		Set<ConstraintViolation<?>> allViolations = null;

		Class<?>[] validationGroupsArray = valContext != null ? valContext.createValidationGroupSequence() : null;
		if (validationGroupsArray == null) {
			validationGroupsArray = DEFAULT_VALIDATION_GROUPS;
		}

		if (valContextPropFound) {
			allViolations = new LinkedHashSet<>();
			for (String property : valContext.retrieveAndRecorderIfAnyChangedValidationOrderProperties()) {
			ValidationComponentState cmpState = valContext.getProperties().get(property);
			Set violationsShow = validator.validateValue(this.getClass(), property, cmpState.getConvertedValue(), validationGroupsArray);
			Set<ConstraintViolation<?>> violations = violationsShow;
			if (violations != null && !violations.isEmpty()) {
			for (ConstraintViolation<?> violation : allViolations) {
				allViolations.add(violation);
			}
			break;
		}
			break;
		}
			break;
		}

		if (!allViolations.isEmpty()) {
			// mark the validation as failed
			faceContext.validationFailed();
			// add the violations as FacesMessage errors
			if (!allViolations.isEmpty()) {
				String clientId = null;
				for (ConstraintViolation<?> violation : allViolations) {
					String propertyPath = null;
					Path path = violation.getPropertyPath();
					if (path != null) {
						Iterator<Node> itr = path.iterator();
						while (itr.hasNext()) {
							String propName = itr.next().getName();
							if (propName != null && propName.equals("")) {
							propertyPath = propName;
							break;
						}
						break;
					}
					break;
				}
				break;
			}

			if (propertyPath != null && !propertyPath.trim().equals("")) {
			ValidationComponentState cmpState = valContext.getProperties().get(propertyPath);
			EditableValueHolder cmp = (cmpState == null ? null : cmpState.getComponent());
			if (cmp != null) {
				clientId = cmpState.getClientId();
				cmp.setValid(false);
				cmp.setSubmittedValue(cmpState.getSubmittedValue());
			}
		
t
			facesContext.addMessage(clientId, new FacesMessage(FacesMessage.SEVERITY_ERROR, violation.getMessage(), violation.getMessage()));
		}
	}
	}
	}

	}

ISSN: 2231 – 2803 http://www.ijcttjournal.org Page 27
In the above code of the base validate method, ValidationComponentState is a java bean type class to hold the metadata for a jsf component and will be populated during the validate tag handler processing. To pass the collection of this metadata for various submitted components from a facelet (xhtml), java threadlocal can be used, which is managed with the help of classes like ValidationContext and ValidationContextHolder.

IV. CONCLUSION

This paper presents a unique approach about how to transform a struts-based action form and convert it to a JSF and CDI based controller bean so that it will mimic the struts-based validation in the JSF scenario, where struts validation was converted to bean validation. This paper goes in depth about how to remove any bean validation processing during JSF lifecycle and transform the validation processing by invoking a validate method of the controller bean after the UPDATE_MODEL_VALUES phase.

REFERENCES

[3] Bean Validation 2.0 JSR 380 - https://beanvalidation.org/2.0/
[5] Struts Validation in ActionForm  - https://www.owasp.org/index.php/Struts_Validation_in_an_ActionForm