International Journal of Computer Trends and Technology
ISSN: 2231-2803 / https://doi.org/10.14445/22312803/IJCTT-V67I4P105
Volume 67 Issue 4, 24-28, April 2019

Original Article

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 PROCESS_VALIDATIONS after the APPLY_REQUEST_VALUES phase, and if there are no validation issues, then the control passes to the UPDATE_MODEL_VALUES phase. Else it goes to the RENDER_RESPONSE phase. While in the Struts 1. x application, 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 the 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 Action Errors validate (Action Mapping mapping, Http Servlet Request request)*. The implementation here executes all the configured struts-based validation on various form properties. Developers must execute this super method from their respective action forms validate method. Most of the time, the application will also have various non-configured validations present in the validate method. For every validation failure in struts, an *ActionError*will is created and stored in the request/response for it to be displayed during the rendering of the page. Suppose 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. In that case, 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 by using 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 active form bean (will call controller bean in JSF) is populated with the request parameters. The downside to this is it will be populated with invalidated data. Developers need to ensure that their forms are validated before the data is sent for further processing through the INVOKE_APPLICATION phase.

II. STRUTS VALIDATE TO JSF VALIDATE

Most 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

```
public class UserForm extends ValidatorForm {

    private String firstName;

    public String getFirstName(){
        return firstName;
    }

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

public ActionErrors
validate(ActionMappingmapping,HttpServletRequest request) {

ActionErrors errors = super.validate(mapping, request)

        if (errors.size() > 0) {
        return errors;
        }
    //some other specific validations

}
```

B. Struts Validation Configuration

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

The *user form* above provides the struts validation configuration based on the *validat*ion method. The code *super. Validate (mapping, request)*will use the validation config, and if any validation error gets added to the *ActionErrors.*

C. JSF based User Controller Bean with the first Name

Instead of calling this class 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 voidvalidate() {
FacesContextfacesContext = FacesContext.getCurrentInstance();
    super.validate();
    if (facesContext.getMessageList(). size() > 0) {
             return;
    }
        //some other specific validations

    }
```

In the above class, the field *firstName* is annotated with the *NotNull* Bean Validation constraint annotation, which is like *the required* validation from Struts. The *validate* method has been transformed by removing all the Struts-based parameters and managing any request/response-based invocation through *FacesContext*. In the superclass, *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 a *ControllerBean* and validate the bean validation constraint annotations annotated on the field or the property.

```
javax.validation.Validator validator = // fetch the validator
for (String property : propertyList) {
    Set violationsRaw = validator.validateValue(this.getClass(),
property, propertyValue, beanValGroups);
             //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 the *ControllerBean* after it has been populated with the request parameters. To ensure this, validate method needs to be executed after the *UPDATE_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 maneuver the mechanism of moving the PROCESS_VALIDATIONS phase after the UPDATE_MODEL_VALUES phase.

```
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 the RESTORE_VIEW phase.

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

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

- Remove the JSF-based *BeanValidator* from the JSF component, or else the validation will get invoked during the PROCESS_VALIDATION.

```
Validator[] validators = component.getValidators();
for (Validator validator : validators) {
    if (validator instanceofBeanValidator) {
             return (BeanValidator) validator;
    }
}//create a dummy validation group
private interface NoValidationGroup {}

//set a dummay 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 validator *Collect Submit Val Validator* to the JSF component that collects the submitted and converted value through its *validate* method but doesn't perform the actual validation.

This submitted/converted values collection 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 and adding a new validator to collect the submitted and converted values must be executed through a phase listener. This should be invoked before the PROCESS_VALIDATIONS phase.

A new phase listener should also be set up to be invoked after the PROCESS_VALIDATIONS phase 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.

```
//restore the original bean validation group

BeanValidatorbeanValidator = getBeanValidator(component);
if (beanValidator != null) {
String originalValiGrp =  component.getAttributes()
    .remove("original_bean_val_group");
    beanValidator.setValidationGroups("original_bean_val_group")
;
}

//remove the 'CollectSubmitValValidatdor'

EditableValueHoldervalueHolder = //reference component

Validator colValidator = null;
for (Validator validator : valueHolder.getValidators()) {
    if (validator instanceofCollectSubmitValValidatdor)
            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 to invoke the actual *validate* method on the controller bean. This phase listener should be added after the UPDATE_MODEL_VALUES phase.

```
//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 OmniFaces*ValidateBean* 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*

```
<ui:composition      xmlns="http://www.w3.org/1999/xhtml"
                     xmlns:h="http://xmlns.jcp.org/jsf/html"
                     xmlns:f="http://xmlns.jcp.org/jsf/core"
                     xmlns:ui="http://xmlns.jcp.org/jsf/facelets"
                     xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
                     xmlns:pt="http://xmlns.jcp.org/jsf/passthrough"
                     xmlns:p="http://primefaces.org/ui"
                     xmlns:o="http://omnifaces.org/ui"
                     xmlns:poc="http://test/ui"
                     xmlns:tag="http://test/tag"
                     xmlns:func="http://test/functions"
                  template="/WEB-INF/faces/template/template.xhtml">

        <ui:define name="title">
            Test Title
        </ui:define>

        <ui:define name="content">
            <h:form >
                <p:panelGrid columns="2" >
                    <f:facet name="header">
                        Header
                    </f:facet>
                   <p:outputLabel value="First Name" for = "firstName" />
                   <p:inputText value="#{userController.userControllerBean.firstName}"
                            id="firstName" />

                    <f:facet name="footer">
                        <p:commandButton value="Save"
                             action="#{userController.save}" update="@form" />
                    </f:facet>
                </p:panelGrid>
                <tag:valTag value="#{userController.userControllerBean}"/>
            </h:form>
        </ui:define>
</ui:composition>
```

E. *User Controller Bean*

The code in the Bean below shows the *validate* method and annotated *first name* field with the *NotNull* constraint annotation and shows a manual custom error condition.

```
package test;

import java.io.Serializable;

import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.validation.constraints.NotNull;

@SuppressWarnings("serial")
public class UserControllerBean extends BaseControllerBean
            implements Serializable{

    @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.getMessageList(). size() > 0) {
            return;
        }
        //add custom error handling
        if("John Doe".equalsIgnoreCase(firstName)) {
            String errorMsg = "User First name cannot be John Doe";
            facesContext.addMessage(firstName,new FacesMessage(
                    FacesMessage.SEVERITY_ERROR,
                    errorMsg,errorMsg));
            facesContext.validationFailed();
        }
    }
}
```

F. *Base Controller Bean*

This Bean can be used as a superclass 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.component.EditableValueHolder;
import javax.faces.context.FacesContext;
import javax.validation.ConstraintViolation;
import javax.validation.Path;
import javax.validation.Path.Node;
import javax.validation.Validator;
import javax.validation.groups.Default;

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

        FacesContext facesContext = FacesContext.getCurrentInstance();
        javax.validation.Validator validator = (Validator) facesContext.getExternalContext().
                getApplicationMap().get("javax.faces.validator.beanValidator.ValidatorFactory");
        ValidationContext valContext =  ValidationContextHolder.getBeanValidationContext();

        boolean valContextPropFound = valContext != null && valContext.getProperties() != null
                && !valContext.getProperties().isEmpty() ? true : false;

        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.retrieveAndReorderIfAnyChangedValidationOrderProperties()) {
                 ValidationComponentState cmpState = valContext.getProperties().get(property);
                 Set violationsRaw = validator.validateValue(this.getClass(), property,
                             cmpState.getConvertedValue(), validationGroupsArray);
                 Set<ConstraintViolation<?>> violations = violationsRaw;
                 if(violations!=null && !violations.isEmpty()){
                     allViolations.addAll(violations);
                 }
             }
         }
         if (allViolations != null && !allViolations.isEmpty()) {
             //mark the validation as failed
             facesContext.validationFailed();
             //add the violations as FacesMessage errors
             if (allViolations != null && !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;
                             }
                         }
                     }
                     clientId = null;
                     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());
                         }
                     }
                     facesContext.addMessage(clientId,new FacesMessage(FacesMessage.SEVERITY_ERROR,
                             violation.getMessage(),violation.getMessage()));
                 }
             }
         }
    }
}
```

In the above code of the base validate method, *ValidationComponentState* is a java bean type class to hold the metadata for a jsf component. It 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 *thread-local* can be used, which is managed with the help of classes like *ValidationContext* and *ValidationContextHolder*.

IV. CONCLUSION

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

REFERENCES

[1] Apache Commons Validator - https://commons.apache.org/proper/commons-validator/
[2] OmniFaces - http://omnifaces.org/
[3] Bean Validation 2.0 JSR 380- https://beanvalidation.org/2.0/
[4] Java Server Faces - https://javaee.github.io/javaserverfaces-spec/
[5] Struts Validation in ActionForm - https://www.owasp.org/index.php/Struts_Validation_in_an_ActionForm.