Bean validation for JSF Components (Valerie)

Rudy De Busscher

Version 0.9, ??/??/2017

Table of Contents

Introduction

Origin

As mentioned, Jerry was derived from the Apache MyFaces extensions Validation framework (ExtVal). Not have to define the validation requirements on the JSF components, like required and the maximum size of a field, was for us a main important feature of that Framework. That is why they are recreated with Valerie (Validation -> Val -> Valerie as I would like to have a name companion to Jerry).

Besides some features which are available in ExtVal, some other nice features are added like the Date provider concept for the Date range validation.

Since Valerie is solely targeted to the Java EE ecosystem, only Bean Validation annotations are considered in this framework (this in contrast to ExtVal). But a plugin can be created to support other annotations as well.

Atbash migration

Mapping between the old and the new maven artifacts.

Rubus artifact	Atbash Artifact
<groupId>be.rubus.web</groupId> <artifactId>valerie</artifactId>	<groupId>be.atbash.ee.jsf</groupId> <artifactId>valerie</artifactId>

Features

1. MetaDataExtractor, MetaDataTransformer

   Key components in extracting the annotations found on properties referenced from JSF components and determining the meta data (required, max length, …)

2. Initializer's

   With the help of the ComponentInitializer concept of Jerry, places the information on the JSF Components.

3. Custom bean validation annotations

   Like *@ValSize* which combines the *@NotNull* and *@Size* standard annotations.

4. Date provider concept

   @ValPast and *@ValFuture* are retrieving the current date from a provider so that it becomes easier to test time related aspects in your application.

5. Support for custom annotations

Valerie allows the integration of your custom annotations.

6. Cross Field validation in Validation JSF Phase

   With the help of the @RecordValue we are able to perform class level bean validations in the Process Validations JSF Phase. And this without placing the screen values in the bean.

Releases

v0.9 (??/??/2017)

1. Migrate to Atbash namespace (Mvn artefact and package names)

v0.4 (17/12/2016)

1. Renamed the PrimeFaces plugin of valerie to artifact-id valerie-primefaces

Configure your project

In case you don't use maven, you can just download the jar file and put in the `lib` folder of your project.

1. Open the project `pom.xml` file for editing.
2. Add the JCenter Maven repository where the dependency can be found.

   ```
   <repository>
       <id>Bintray_JCenter</id>
       <url>https://jcenter.bintray.com</url>
   </repository>
   ```

3. Add the Valerie module to the list of dependencies.

   ```
   <dependency>
       <groupId>be.atbash.ee.jsf</groupId>
       <artifactId>valerie</artifactId>
       <version>0.9</version>
   </dependency>
   ```

   There is no need to define the Jerry dependency as it is included through the transitive dependencies.

4. You are ready to use Valerie.

PrimeFaces support

There is a plugin of Valerie which supports the PrimeFaces components. You only have to add the

following dependency to your POM file.

```
<dependency>
    <groupId>be.atbash.ee.jsf.valerie</groupId>
    <artifactId>valerie-primefaces</artifactId>
    <version>0.9</version>
</dependency>
```

This plugin just contains a few Component Initializers to put the correct information on the JSF PrimeFaces components.

Usage scenarios

Indicate required field

When a JSF component is linked to a Java property in a bean which indicate that it is required, for example it has a *@NotNull* annotation, the JSF component will have his required property set.

Java property definition of a required value.

```
@Model
public class RequiredBean {

    @NotNull
    private String required;

   // Getter and setter
}
```

Jsf component linked to this required property.

```
   <p:outputLabel id="requiredLabel" value="required" for="required"/>
   <p:inputText id="required" value="#{requiredBean.required}"/>
```

The above example uses PrimeFaces and the PrimeFaces Valerie plugin. With PrimeFaces it is easy to see that a field is required because the label gets an additional * at the end.

Maximum size for input fields

When you specify the @Size attribute on a String property in a bean, the linked JSF component will set the size property so that no more then the indicated number of characters can be inputted into the field.

```
@Model
public class MaxSize {

    @Size(max = 5)
    private String value2;

    // Getter and setter
}
```

A JSF component linked to this property will only allow 5 characters to be entered.

Remaining characters support for PrimeFaces TextArea

The PrimeFaces textArea component has support for a label which indicates how many characters can be entered (total - already entered).

The *@Size* (and *@ValSize*) max() attribute is integrated with this feature.

```
@Model
public class DescriptionBean {

    @Size(max = 500)
    private String description;

    // Getter and setter
}
```

Using the following fragment on the screen

```
        <p:inputTextarea id="description" value="#{descriptionBean.description}"
                         counter="remaining" counterTemplate="{0} characters
remaining"/>
        <h:outputText id="remaining"/>
```

Will result in the text (initially when no character is in the text Area typed) *500 characters remaining*.

Custom defined Bean validation @ValSize

@ValSize is a custom defined Bean validation, which is almost identical to the standard *@Size* version. Except that the default value for the min attribute is 1. You can see @ValSize as the non optional version of @Size.

```
@Model
public class RequiredBean {

    @ValSize
    private String required;

   // Getter and setter
}
```

So defining the annotation without any values, like the above example, makes the field required.

Combine Bean validations annotations

The official name for this is *Constraint composition*. You can combine several bean validation annotations together and define a new name for them. Valerie has also support for this type of validation.

When a JSF component refers to a Java property which has the @CombinedValidation annotation, it will be required and have a maximum length of 14 characters.

Example definition of a constraint composition.

```
@NotNull
@Size(min = 2, max = 14)
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
@Constraint(validatedBy = {})
@Documented
public @interface CombinedValidation {

    String message() default "Must be between 2 and 14 characters";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

DateProvider concept for ValPast and ValFuture

Testing date en time related aspects of your applications can be painful. It is hardly an option to change the system clock of your test server to see what happens next month.

Therefor an alternative version of *@Past* and *@Future* is defined which allows to provide a date for the current point in time.

The interface DateProvider is defined as follows

Definition of DateProvider within Valerie

```
public interface DateProvider {
    Date now();
}
```

When a CDI bean is found which implements this interface, the *now()* method is called instead of asking for the system time.

The example of this feature uses a CDI managed JSF bean so that the user can change the date on screen which used in the checks by *@ValPast* and *@ValFuture*.

Example of a CDI managed JSF bean as DateProvider

```
@ApplicationScoped
@Named
public class DateProviderBean implements DateProvider {

    private Date fixedNow = new Date(); // default is equal to system date.

    public Date getFixedNow() {
        return fixedNow;
    }

    public void setFixedNow(Date fixedNow) {
        this.fixedNow = fixedNow;
    }

    @Override
    public Date now() {
        return fixedNow;
    }

}
```

@DateRange class level validation and @RecordValue

With the @DateRange class level bean validation annotations, you can verify if the start date comes before the end date.

Example usage of @DateRange

```
@DateRange(start = "startDate", end = "endDate")
public class DateRangeBean {

    private Date startDate;

    private Date endDate;

    // getters and setters
}
```

The properties containing the *start date* and the *end date* must always be indicated.

This is a regular class level bean validation and will be executed when other validations are verified.

But when we add the @RecordValue annotations to the properties **startDate** and **endDate**, the values from the screen will be recorded during the Process Validation JSF Lifecycle phase. A phase listener will perform the validation at the end of the phase without putting the values into the bean.

Advanced usages

Custom validation

When you create a custom Bean validation annotation and validator, you can integrate it with Valerie by implementing the MetaDataTransformer interface.

As example we take a Validator for the Belgian zip codes (4 digits). The annotation looks like this

ZipCode Bean validation annotation

```
@Target({METHOD, FIELD, ANNOTATION_TYPE})
@Retention(RUNTIME)
@Constraint(validatedBy = {ZipCodeValidator.class})
@Documented
public @interface ZipCode {

    String message() default "Zip code is not valid (1000 - 9999)";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};
}
```

Whenever we use this annotation on a String property, we like to have the JSF component to have the required attribute set and at maximum 4 characters that can be entered. This metadata information is defined by the MetaDataTransformer implementation we make for ZipCode.

MetaDataTransformer for ZipCode

```
@ApplicationScoped
public class ZipCodeMetaDataTransformer implements MetaDataTransformer {
    @Override
    public Map<String, Object> convertMetaData(MetaDataEntry metaData) {
        Map<String, Object> result = new HashMap<>();
        if (ZipCode.class.getName().equals(metaData.getKey())) {
            result.put(CommonMetaDataKeys.REQUIRED.getKey(), Boolean.TRUE);
            result.put(CommonMetaDataKeys.SIZE.getKey(), 4);
            result.put(ZipCode.class.getName(), Boolean.TRUE);
        }
        return result;
    }
}
```

When this *Transformer* sees the ZipCode annotation, it adds 3 values to the metaData.

1. It identifies it as Required
2. The size is set to 4 characters maximum
3. The ZipCode class name is added so that ComponentInitializers can use it if needed.

Remark: It is important that we mark this class with a CDI scope in order to be picked up by Valerie.

The following code shows how you can use the metaData info about the ZipCode in a *ComponentInitializer* to add/update the mask attribute of the PrimeFaces Mask component.

```
@ApplicationScoped
@InvocationOrder(101)
public class ZipCodeComponentInitializer implements ComponentInitializer {
    @Override
    public void configureComponent(FacesContext facesContext, UIComponent uiComponent,
Map<String, Object> metaData) {
        if (metaData.containsKey(ZipCode.class.getName())) {
            InputMask input = (InputMask) uiComponent;
            input.setMask("9999");
        }
    }

    @Override
    public boolean isSupportedComponent(UIComponent uiComponent) {
        return  uiComponent instanceof InputMask;
    }
}
```