



JSF Component that accepts only digits with maxlength option

Technical White Paper

NumberInput

Table of Contents

1	SYNOPSIS / EXECUTIVE SUMMARY	3
2	KEYWORDS / CATEGORY	3
3	PROBLEM STATEMENT / HYPOTHESIS.....	3
4	ANALYSIS OF OBSERVATIONS.....	4
5	ALTERNATIVE SOLUTIONS	4
6	PROPOSED SOLUTION	4
6.1	ARCHITECTURAL / DESIGN CONSIDERATIONS.....	4
6.2	ADVANTAGES OF THE PROPOSED APPROACH.....	4
6.3	ADAPTABILITY / PORTABILITY OF THE SOLUTION.....	5
6.4	IMPLEMENTATION OF THE SOLUTION.....	5
7	CONCLUSIONS.....	8

1 Synopsis / Executive Summary

- The HTMLInputText component provided by JSF accepts the given input as text and hence it has various limitations
- Suppose if the form requires to get only digits, for example mobile number, developer has to handle this by writing an exclusive validator or user has to handle in presentation layer.
- NumberInput is a JSF Custom Component, which accepts only digits.
- Since it accepts only digits, developer need not worry about writing a validator or handling it in the presentation layer
- The other benefit of this NumberInput component, the developer is given an option of defining the maximum length of the number
- NumberInput will throw a faces error message when the user enters other than number saying "*Invalid Number*"
- NumberInput will throw a faces error message when the user enters digits more than the given max length saying "*Maximum number of digits allowed is <max length>*"
- These faces error messages can be displayed at presentation layer using the JSF component `<h:message/>`.

2 Keywords / Category

- JEE, JSF, JSF Custom Component, Custom Component, Faces Custom Component.

3 Problem Statement / Hypothesis

- The current HTMLInputText component provided by JSF accepts the given input as text
- If the form requires accepting only limited number of digits, since HTMLInputText accepts alphabets, numbers, special characters, etc., developer has to write extra logic to validate the input
- Now the developer has two options. First option, developer has to write a custom re-usable validator and then bind it with HTMLInputText. In the second option, developer has to handle this validation in the presentation layer
- If the developer uses the second option, he/she has to follow the same steps for all the input fields in the form, wherever they need only digits to be accepted

4 Analysis of Observations

Since HTMLInputText component accepts alphabets, numbers, special characters, etc., developer has to write extra logic to validate the input. There are two options; a developer can follow for validating an input.

In the first option, developer has to write a custom re-usable validator and then bind it with HTMLInputText. In this case, this validator can be used for other input fields also, since validators are re-usable.

In the second option, developer has to handle this validation in the presentation layer.

If the developer uses the second option, he has to follow the same steps for all the input fields in the form, wherever he needs only digits to be accepted.

5 Alternative Solutions

Developer has to, manually validate the given input by developing a custom validator or by handling it the backing bean.

6 Proposed Solution

Creating a custom component, which accepts only digits with an option for controlling number of digits to be entered. We will call this custom component as “**NumberInput**”.

6.1 Architectural / Design Considerations.

Following are the design steps to be followed while creating a custom component.

- ✓ Create a component class that extends javax.faces.component.UIInput
- ✓ Create a renderer class that extends javax.faces.render.Renderer
- ✓ Create a tag class that extends javax.faces.webapp.UIComponentTag
- ✓ Create a description to TLD
- ✓ Register the component in faces-config
- ✓ Register the renderer in faces-config
- ✓ Create a custom validator and register it to the component class.

6.2 Advantages of the Proposed Approach

- NumberInput is a JSF Custom Component, which accepts only digits
- Since it accepts only digits, developer need not worry about writing a validator or handling it in the presentation layer

- The other benefit of this NumberInput component is the developer is given an option of defining the max length of the number
- NumberInput will throw a faces error message when the user enters other than number saying "Invalid Number" and a faces error message when the user enters digits more than the given max length saying, "Maximum number of digits allowed is <max length>"
- These error messages can be customized by using the attributes "errorMessage" and "errorMessageForLength"
- These faces error messages can be displayed at presentation layer using the JSF component <h:message/>
- NumberInput can be used with any of JSF frameworks with or without minor modifications.

6.3 Adaptability / Portability of the Solution

NumberInput custom component can be used in any of JSF frameworks like faces, my-faces, rich-faces, ADF Faces, IceFaces, etc, without or with minor modification.

6.4 Implementation of the Solution

1 Create a component class that extends javax.faces.component.UIInput:

Our custom component class has to extend UIInput since it is an input component. Below is the portion of the class.

```
public class NumberInput extends UIInput {
    public static final String NUMBER_FAMILY = "NUMBERFAMILY";
    private int maxLength = 0;
    private String errorMessage = "Invalid Number";
    private String errorMessageForLength = "Number of digits exceeds the allowed
limit";
    public NumberInput() {
        super();
        addValidator(new NumberValidator());
    }
    public String getFamily() {
        return NUMBER_FAMILY;
    }
}
```

2 Create a renderer class that extends javax.faces.render.Renderer.

Create a renderer java class by extending Rendered provided by jsf, and implement decode and encode behaviors for the component. Below code shows a implementation of decode and encode methods.

```
public void decode(FacesContext context, UIComponent component) {
    isValidInput(context, component);
    if (component instanceof UIInput) {
```

```

        UIInput input = (UIInput) component;
        String clientId = input.getClientId(context);
        Map requestMap = context.getExternalContext()
            .getRequestParameterMap();
        String newValue = (String) requestMap.get(clientId);
        if (null != newValue) {
            input.setSubmittedValue(newValue);
        }
        NumberInput numberInput = (NumberInput) component;
        String maxLength = (String)
component.getAttributes().get("maxlength");
        if(maxLength != null){

            numberInput.setMaxLength(Integer.parseInt(maxLength));
        }
    }

    public void encodeEnd(FacesContext ctx, UIComponent component)
        throws IOException {
        isValidInput(ctx, component);
        ResponseWriter writer = ctx.getResponseWriter();
        writer.startElement("input", component);
        writer.writeAttribute("type", "text", "text");
        String id = (String) component.getClientId(ctx);
        writer.writeAttribute("id", id, "id");
        writer.writeAttribute("name", id, "id");
        String size = (String) component.getAttributes().get("size");
        if (size != null) {
            writer.writeAttribute("size", size, "size");
        }
        Object currentValue = getValue(component);
        writer.writeAttribute("value", formatValue(currentValue),
"value");
        writer.endElement("input");
    }

```

3 Create a tag class that extends javax.faces.webapp.UIComponentTag.

In order Create a tag java class by extending UIComponentTag provided by jsf, and add the custom attributes like errorMessage, errorMessageForLenght as its properties.

4 Create a description to TLD.

Following is the tag descriptor code for the our custom component.

```

.....
<tag>
  <name>numberInput</name>
  <tag-class>com.icesoft.component.inputnumber.NumberInputTag</tag-class>
  <body-content>empty</body-content>
  <description>
    This is the tag for the credit card input component.
  </description>

```

```

<attribute>
  <name>errorMessage</name>
  <required>>false</required>
  <rtexprvalue>>false</rtexprvalue>
  <description>
    General error message when the number is invalid
  </description>
</attribute>
<attribute>
  <name>errorMessageForLength</name>
  <required>>false</required>
  <rtexprvalue>>false</rtexprvalue>
  <description>
    Error message when the number of digits exceeds the max length.
  </description>
</attribute>
</tag>

```

5 Register the component in faces-config.

```

<component>
  <component-type>
    NUMBER_INPUT
  </component-type>
  <component-class>
    com.icesoft.component.inputnumber.NumberInput
  </component-class>
</component>

```

6 Register the renderer in faces-config.

```

<renderer>
  <description>
    Renderer for the credit card component.
  </description>
  <component-family>NUMBERFAMILY</component-family>
  <renderer-type>
    NUMBER_RENDERER
  </renderer-type>
  <renderer-class>
    com.icesoft.component.inputnumber.NumberInputRenderer
  </renderer-class>
</renderer>

```

7 Create a custom validator and register it to the component class.

Create a validator by implementing standard Validator of JSF and implement the validate method as shown below. Registering to the component class is done in the constructor of the NumberInput component by invoking addValidator(new NumberValidator())

```

public void validate(FacesContext context, UIComponent component,
    Object value) throws ValidatorException {
    if (null != value) {

```

```
        if (!(value instanceof String)) {
            throw new IllegalArgumentException("The value must
be a String");
        }
        String ccNum = (String) value;
        NumberInput numberInput = (NumberInput) component;
        try {
            int num = Integer.parseInt(ccNum);
            if(numberInput.getMaxLength()!=0 &&
ccNum.length()>numberInput.getMaxLength()){
                throw new ValidatorException(new
FacesMessage(numberInput.getErrorMessageForLength()));
            }
        } catch (NumberFormatException e) {
            throw new ValidatorException(new
FacesMessage(numberInput.getErrorMessage()));
        }
    }
}
```

7 Conclusions

Now we are ready to use this component in JSF applications. Same component will work with IceFaces, ADF Faces, My Faces, Rich Faces, etc, without any modifications. If we need to make use the features provided by these frameworks, for example partial submit of icefaces, we need to implement corresponding classes provided by them.

Following is the sample code to be used in the jsp files:

```
<myComponents:numberInput id="test" maxlength="2"
value="#{numberBean.number}" errorMessage="Invalid Number"
errorMessageForLength="Maximum allowed digits is 2" />
```