JSF Renderer Interceptor (Jerry)

Rudy De Busscher

Version 0.9, ??/??/2018

Table of Contents

Introduction

Origin

On various occasions, I needed the *RendererInterceptor* concept of Apache MyFaces extensions Validation framework. I did not always need the great validation features of the framework, so I found it an overhead, in terms of performance and memory consumption.

So I extracted that interface and the required classes and transformed it to use CDI to have the benefits of it. That is how Jerry (JSF Renderer Interceptor, JRI) was started. Jerry is the basis for **Octopus** from version 0.9.5 on but has also some usage scenarios on his own.

Atbash migration

Mapping between the old and the new maven artifacts.

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

Features

1. RendererInterceptor

   With the **RendererInterceptor** you have a before and after method for every major method in a Renderer which is `decode`, `encodeBegin`, `encodeChildren`, `encodeEnd` and `getConvertedValue`.

2. ComponentInitializer

   With **ComponentInitializer** it becomes easier to adjust the JSF components. They use a *RendererInterceptor* in the background and work is done in the `beforeEncodeBegin` phase.

3. RepeatableComponentInitializer

   A **RepeatableComponentInitializer** is like a *ComponentInitializer* but the initializer is executed every time the component is rendered. This in contrast with *componentInitializer* which is only executed once for each component. See also Initializers executions further on.

There are some other small CDI features which are useful in any project

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 *Jerry* module to the list of dependencies.

```
<dependency>
    <groupId>be.atbash.ee.jsf</groupId>
    <artifactId>jerry</artifactId>
    <version>${atbash.jerry.version}</version>
</dependency>
```

3. You are ready to use Jerry.

Releases

v0.9 (??/??/2018)

1. Migrate to Atbash namespace (Mvn artefact and package names).
2. RepeatableComponentInitializer
3. Use other Atbash projects.
4. Remove Java EE 6 compliance.

v0.4.1 (25/06/2017)

1. Support for OWB proxies

v0.4 (17/12/2016)

1. Helper for dynamic configuration values (DynamicConfigValueHelper)
2. Warning when for attribute value not found (RequiredMarkerInitializer)

v0.3 (08/03/2016)

1. @ConfigEntry has a member noLogging so that content isn't logged, except when -Djerry.log.all=true

v0.2.3 (15/02/2016)

1. Bug fixing (#8)

v0.2.2 (25/01/2016)

1. Bug fixing (#1)

v0.2.1 (26/11/2015)

1. @DataRange has now the equalsAllowed member to indicate that start date can be equal to end date.
2. Bug fixing (#6)

Initializers executions

ComponentInitializer and **RepeatableComponentInitializer** are executed once or every time the component is rendered according to the following rules.

A *ComponentInitializer* is in general only executed once. So when the component due to AJAX functionality is executed a second time, the initializer is not executed again.

The exception is when the component, on which the initializer is executing, is placed within the facelets repeat component (ui:repeat) or a component based on *UIData* like h:datatable. Then it is also executed multiple times to ensure proper functionality within that repeating component.

A *RepeatableComponentInitializer* is by default always executed when the component is rendered within a repeating component or not.

ValueExpression usage

Developers must be aware that setting *properties*, like UIComponent.setValue for a label, will make the EL expression no longer used. This is especially important when the initializer is executed multiple times (within a repeating component or the *RepeatableComponentInitializer*

For example, we have following label definition

```
<h:label value="#{msg['label.key']}" for="inputField" />
```

When you add an * at the end of the label to indicate it is required, you could do something like this

```
label.setValue(label.getValue() + " * ");
```

However, when this snippet is executed multiple times, each time an asteriks is added at the end, also when the value of the label is defined as an EL expression.

This is because the setValue sets a 'hardcoded' value, just as we do it within the JSF XHTML page which will ignore the EL expression.

For retrieving the correct vale of the attribute, one can use the methods provided in the **ComponentUtils** class. It checks first for an EL Expression and then for a fixed, hardcoded value.

However, the developer is still responsible for checking the value within the *ComponentInitializer* because

```
label.setValue(ComponentUtils.getValue(label, facesContext) + " * ");
```

still results in multiple asteriks when a fixed value si defined in the value attribute of the label.

Usage scenarios

Component Initializer

Jerry can initialize any JSF component just before it will be rendered.

As example, the code is shown for setting the background color of required fields.

ComponentInitializer which makes each PrimeFaces InputText component with a reddish background color when it is required.

```
@ApplicationScoped
public class RequiredInitializer implements ComponentInitializer {
    @Override
    public void configureComponent(FacesContext facesContext, UIComponent uiComponent,
Map<String, Object> metaData) {
        InputText inputText = (InputText) uiComponent;
        if (inputText.isRequired()) {
            String style = inputText.getStyle();
            if (style == null) {
                style = "";
            }
            inputText.setStyle(style + " background-color: #B04A4A;");
        }
    }

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

This are the important aspects of the code.

1. Implement the `ComponentInitializer` interface.
2. Annotate the class with `@ApplicationScoped` CDI scope.
3. Define in the `isSupportedComponent` method if this ComponentInitializer should handle the component.
4. Perform the required functionality in the `configureComponent` method.

The metaData parameter is filled up by Valerie, the (Bean) validation companion of Jerry. In the advanced use case scenarios, there is also an example how you can use it using only Jerry features.

Startup Event

You can use the CDI event StartupEvent to perform any initialization when your application is deployed and ready on the server.

Log some message when application is ready

```
    public void onStartup(@Observes StartupEvent startupEvent) {
       System.out.println("Ready to roll"); // Please use logger !
    }
```

You can also using the startup EJB singleton beans to perform some initialization. This is preferred if the initialization does some database actions.

Injectable Logger

Jerry uses SLF4J as logging facade. You can inject such loggers by creating a simple Producer method. That method is available within Jerry and thus injectable loggers can be used.

Usage of injectable logger.

```
@Inject
private Logger logger;

public void doSomething() {
   logger.info("Performed the doSomething");
}
```

The type of logger is `org.slf4j.Logger`.

Configuration

In very rare situations you need to change the configuration of Jerry. The values can be specified in files with the basename *jerry* or defined in other configuration files as long as they are known to the Atbash configuration system.

jerry.renderkit.wrapper.class

Defines the RenderKitWrapper which is responsible for creating custom *Renderer*s so that we can 'intercept' the calls to `decode`, `encodeBegin`, `encodeChildren`, `encodeEnd` and `getConvertedValue`.

If you need your own custom version of these *Renderer*s, another wrapper is not an issue we are discussing the situation where you ant to replace the functionality within the *JerryRendererWrapper*, you can specify the class name of the RenderKitWrapper with this parameter.

Advanced usages

RendererInterceptor

TODO

Override configuration

TODO

Technical aspects

How RendererInterceptor is applied

TODO

Ordering of RendererInterceptors

TODO

ComponentStorage, MetaDataHolder, MetaDataEntry and MetaDataTransformer

TODO