Attached Object, UI Component Wovening of Attributes using Meta Rule API - JSF

Vijay Kumar Pandey

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

Abstract*. The article is intended to provide an in-depth understanding to one of the complex and obscure part of the JavaServer Faces (JSF) known as AttachedObject. It also details the types of AttachedObjectsand how theirand UIComponent attributes are woven with the help of MetaRule API, keeping the JSF Facelet handler design cohesive but decoupled. The intended audience for this article include application architects, software designers and software programmers who participate in the design, architecting and development of robust and complex enterprise-wide web-based applications using JSF 2.2. This document assumes that the reader has a basic understanding of JSF 2.2.*

Keywords – *AttachedObjects, MetaRule, Validator, Convertor, Behavior, UIComponent, JSF, JSF 2.2, MyFaces*

I. Introduction

This article will dive deep into JSF 2.2 AttachedObjects, MetaRule, UIComponent and AttachedObjectsstate management. AttachedObjects in JSF 2.2 are non-UIComponent objects that provide additional functionality to the UIComponent. These objects type include Validator, Convertor, Behavior, and Events such as ValueChangeListener andActionListener. The word "Attached" in their name is a hint to the fact that these objects do not exist on their own;but they are tied to a certain UIComponent.JSF provides a base interface, javax.faces.view.AttachedObjectHandler,to handlethe processing of AttachedObjects during the transformation of XHTML to Facelets, and their interaction with UIComponents.

This article discusses AttachedObjects in detail, and will utilize code samples from Apache MyFaces 2.2.12 JSF Implementationfor a better understanding of certain scenarios.

II. Important JSF API Classes Associated With AttachedObjects

A pre-requisite to understanding AttachedObjectsin JSF 2.2, is knowledge of the different API Classes that are associated with AttachedObjects:

A. AttachedObjectHandler

This is the base interface for the AttachedObject, like the base interface FaceletHandler, which is implemented by TagHandler. These handler classes take part in building of the view, conversion of XHTML (basically XML)to handlers and then the component tree inside theUIViewRoot.

The AttachedObjectHandlerprovides the following methods, which are implemented by the specific type of attached object handler.

```
public void applyAttachedObject(FacesContext context, UIComponent parent);
public String getFor();
```

The set of sub interfaces that extend this handler class are:

- BehaviorHolderAttachedObjectHandler – to handle tags for behaviors
- EditableValueHolderAttachedObjectHandler – to handle tags for validators, valueChangeListeners. UIInput implements EditableValueHolder, which provides the features to handle those components that can submit values for HTML "input" type elements
- ValueHolderAttachedObjectHandler – to handle tags for converters. UIOutput implements ValueHolder, which provides features to handle those components that render HTML label or textual type elements; they donot submit values but do have converters to alter their values
- ActionSource2AttachedObjectHandler – This is mainly for ActionListener type elements such as f:actionListener

The set of classes that implement these interfaces are:

- FaceletsAttachedObjectHandler – This is the base class that implements AttachedObjectHandler and provides implementation for both its methods. Its implementation is final and cannot be overridden by any sub classes.
  This class also extends from DelegatingMetaTagHandler (this in turn is a subclass of MetaTagHandler, TagHandler, FaceletHandler).

- ConverterHandler – extends from FaceletsAttachedObjectHandler
- ValidatorHandler – extends from FaceletsAttachedObjectHandler
- BehaviorHandler – extends from FaceletsAttachedObjectHandler

B. Classes to help XML elements get wired into UIComponent or AttachedObject

The following is a set of JSF API Base Handler classes that help various XML elements get wired into UIComponent or AttachedObject:

- MetaTagHandler - This class extends from TagHandler and is the base class for any kind of handlers that need to deal with XHTML markup attributes,UIComponent and AttachedObject. This class provides

```
protected abstract MetaRuleset createMetaRuleset(Class type);
```

- DelegatingMetaTagHandler – This class extends MetaTagHandler and provides the base implementation of the above method by simply providing a convenient way of creating the MetaRuleset

```
protected MetaRuleset createMetaRuleset(Class type) {
    return getTagHandlerDelegate().createMetaRuleset(type);
  }
protected abstract TagHandlerDelegate getTagHandlerDelegate();
```

By providing the above mechanism of overriding the TagHandlerDelegate, JSF allows application developers to override the implementation. However, JSF provides the default implementation which is more than adequate for almost all situations.

- TagHandlerDelegateFactory – this is the mechanism that JSF implementation providesfor custom TagHandlerDelegate using which, application developers can provide their own custom implementation to configure in faces config, in the element <tag-handler-delegate-factory>. The main methods that provide the mechanism to create proper TagHandlerDelgate are:

```
public abstract TagHandlerDelegate
createComponentHandlerDelegate(ComponentHandler owner);
public abstract TagHandlerDelegate
createValidatorHandlerDelegate(ValidatorHandler owner);
public abstract TagHandlerDelegate
createConverterHandlerDelegate(ConverterHandler owner);
public abstract TagHandlerDelegate
createBehaviorHandlerDelegate(BehaviorHandler owner);
```

FaceletsAttachedObjectHandler extends DelegatingMetaTagHandler – and this is the way AttachedObjectHandler's are wired with the TagHandlerDelegate.

C. Wovening of XML attributes – ConverterHandler

This section describes how a converter element is attached with its parent component. JSF provides a set of classes that help in setting various attributes from an XHTML into its attributes. To understand this feature, refer to the example below:

XHTML Fragement

```
<h:outputText value="#{testController.dateOfBirth}">
<f:convertDateTime pattern="MMM-dd-yyyy"/>
</h:outputText>
```

Java Controller Class

```
@Named
@RequestScoped
public class TestController implements Serializable {
        private Date dateOfBirth;
        public TestController() {
                Calendar cal = Calendar.getInstance();
                cal.set(Calendar.DAY_OF_MONTH, 10);
                cal.set(Calendar.MONTH, 2);
                cal.set(Calendar.YEAR, 1965);
                dateOfBirth = cal.getTime();
        }
        public Date getDateOfBirth() {
                return dateOfBirth;
        }
        public void setDateOfBirth(Date dateOfBirth) {
                this.dateOfBirth = dateOfBirth;
        }
}
```

In the above XHTML fragment, this validator could also be set by its id

```
<f:converter converterId="javax.faces.DateTime"
pattern="MMM-dd-yyyy"/>
```

This XML element needs to set its converterId and pattern (for this converter, there are some other attributes also). JSF implementation provides SAX mechanism to convert XML elements and attributes into FaceletHandlers (TagHandler, ComponentHandler, AttachedObjectHandler). These FaceletHandlers are then executed in recursive order to build the whole UIViewRoot with chained UIComponent's and wherever needed, with AttachedObject. The Converter attachedObject needs to be set up with its component i.e. h:outputText and this component is javax.faces.component.html.HtmlOutputText.

HtmlOutputText extends from UIOutput and that also implements ValueHolder, which also has two methods

```
public Converter getConverter();
public void setConverter(Converter converter);
```

D. MetaRule, MetaData, MetaRuleSet

JSF provides for several types of Converters, and each with its own set of attributes. In the previous example, theConverter is set as part of the HtmlOutputText. JSF provides various classes and interfaces to allow application developers to set attributes and set AttachedObject into its parent

UIComponent, and to extend this feature without worrying about how XHTML is converted into Facelet and various Handlers and then into a tree of components set inside UIViewRoot. These set of classes is also how component attributes are set into its instances when they are parsed from the XHTML.

- MetaRule - This is the main abstraction class for setting a specific attribute on the primary element. The 'value' attribute is set on the HtmlOutputText; and,converterId and pattern attribute are set on the actual Converter instance which is created via f:converter.Then this Converter instance is set in the HtmlOutputText. The main method of this abstract class is

```
public abstract Metadata applyRule(String name, TagAttribute attribute,MetadataTarget meta);
```

In this method, name parameter is the attribute name;attribute is the XML attribute of that element and,meta is the JSF implementation specific class providing several utility methods over the actual instance on which this rule is applied, i.e.,UIComponent, AttachedObject.

- Metadata – These are the JSF implementation dependent set of classes that provide the mechanism for an attribute to be set on the actual instance. Creating a specific MetaRule, also requires the developer to provide an implementation of the Metadata.

```
public abstract void applyMetadata(FaceletContext ctx, Object instance);
```

- MetaRuleset – This is a collection of MetaRules. Different properties are handled differently and for each property, there is a MetaRule. TheseMetaRules together when applied on an instance of UIComponent orAttachedObjects, lead to a fully populated object. Along with this, it has several utility methods that help if a specific attribute is to be ignored or to be addedas an alias for an attribute.

XHTML is converted into a Facelet, a tree of various FaceletHandlers, which in turn creates UIComponents andAttachedObjects. Therefore, when a component instance is created during buildView process, an empty UIComponentis created with nothing set and the default handler class where UIComponent is created is basically ComponentHandler, one of its super classes is MetaTagHandler (extends TagHandler). Two of methods in this class are:

```
protected abstract MetaRuleset createMetaRuleset(Class type);
```

```
protected void setAttributes(FaceletContext ctx, Object instance);
```

Now, once the component is created (without any attributes set), tag handlers call setAttributes and that is where all Meta related information comes into play to get everything wired together properly. This is what happens inside setAttributes (code excerpt shown below):

```
Class type = instance.getClass();
Metadata mapper = this.createMetaRuleset(type).finish();
mapper.applyMetadata(ctx, instance);
```

E. Java Beans Property Mapper Rule

One of the JSF implementations, related to setting the XHTMLattributes of the element markup for UIComponent and AttachedObject is a MetaRule that helps find the setter method (write method, as per the JavaBeans specs) of that property and sets the value. The value itself can be a literal or dynamic value (EL expression). The default implementation of MetaRuleset provided almost always adds this MetaRule in the MetaRuleset.

F. Wiring of Converter into the Component

As per the FaceletTag Handler composition, there are only two types of FaceletHandler i.e.,TagHandler and CompositionHandler.

These handlers chain up together to create UIComponents and AttachedObjects, through the method:

```
public void apply(FaceletContext ctx, UIComponent parent) throws IOException
```

In this case where ConverterHandler is used, the parent is an instance of HtmlOutputText which is itself a sub class of ValueHolder, as described above. A sample code excerpt (from MyFaces) outlining how this is all wired together, is provided below:

```
public void apply(FaceletContext ctx, UIComponent parent) throws IOException
if (parent instanceof ValueHolder){
applyAttachedObject(ctx.getFacesContext(), parent);
}
}

public void applyAttachedObject(FacesContext context, UIComponent parent){
FaceletContext faceletContext = (FaceletContext) context.getAttributes().get(

        FaceletContext.FACELET_CONTEXT_KEY);

        ValueHolder vh = (ValueHolder) parent;
        ValueExpression ve = null;
        Converter c = null;
        if (<<ConvertHandler>>.getBinding() != null){
                ve = _delegate.getBinding().getValueExpression(faceletContext, Converter.class);
                c = (Converter) ve.getValue(faceletContext);
        }
        if (c == null){
```

```
                c                                                    =
context.getApplication().createConverter(<<converterId>>)
                if (ve != null){
                        ve.setValue(faceletContext, c);
                }
        }
        if (c == null){
                throw                                        new
TagException(_delegate.getTag(), "No Converter was created");
        }
        <<ConvertHandler>>.setAttributes(faceletContext, c);
        vh.setConverter(c);
}
```

III. MetaRule, MetaRuleset And A Dynamic Converter With Complex Properties

A. MetaRule – MethodExpressionMetaRule:

Below is detailed code ofa dynamic converter with attributes of type javax.el.MethodExpression, which provides the real delegated conversion capability. A MetaRule is used to properly set these attribute in the dynamic converter, and to help understand, how any XHTML markup attributes (complex type) can be set in its markup element (for UIComponents and AttachedObjects).

Since specific attributes need to be handled, this requires the creation of a custom ConverterHandler that would be declared in a taglib file, where this handler can be specified (JSF 2.2 does not provide the capability to annotate the converter with a handler).

```
package test;

import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

import javax.el.MethodExpression;
import javax.faces.view.facelets.FaceletContext;
import javax.faces.view.facelets.MetaRule;
import javax.faces.view.facelets.Metadata;
import javax.faces.view.facelets.MetadataTarget;
import javax.faces.view.facelets.TagAttribute;
import javax.faces.view.facelets.TagAttributeException;

/**
 * Based on similar MetaRule from MyFaces code
 *
 */
public class MethodExpressionMetaRule extends MetaRule {

        private final String methodName;

        private final Class<?> returnTypeClass;

        private final Class<?>[] params;

        public MethodExpressionMetaRule(String methodName, Class<?> returnTypeClass, Class<?>[] params) {
                this.methodName = methodName;
                this.returnTypeClass = returnTypeClass;
                this.params = params;
        }

        @Override
        public Metadata applyRule(String name, TagAttribute attribute, MetadataTarget meta) {
                if (!name.equals(this.methodName)) {
                        return null;
                }

                if (MethodExpression.class.equals(meta.getPropertyType(name))) {
                        Method method = meta.getWriteMethod(name);
                        if (method != null) {
                                return new MethodExpressionMetadata(method, attribute, this.returnTypeClass, this.params);
                        }
                }
                return null;
        }

        private static class MethodExpressionMetadata extends Metadata {
                private final Method _method;

                private final TagAttribute _attribute;

                private Class<?>[] _paramList;

                private Class<?> _returnType;
```

```
		public MethodExpressionMetadata(Method method, TagAttribute attribute, Class<?> returnType,
				Class<?>[] paramList) {
			_method = method;
			_attribute = attribute;
			_paramList = paramList;
			_returnType = returnType;
		}

		public void applyMetadata(FaceletContext ctx, Object instance) {
			MethodExpression expr = _attribute.getMethodExpression(ctx, _returnType, _paramList);
			try {
				_method.invoke(instance, new Object[] { expr });
			} catch (IllegalAccessException | IllegalArgumentException | InvocationTargetException e) {
				throw new TagAttributeException(_attribute, e.getCause());
			}
		}
	}
}
```

B. Converter – DynamicConverter

```
package test;

import javax.el.MethodExpression;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

@FacesConverter(DynamicConverter.CONVERTER_ID)
public class DynamicConverter implements Converter {

	public static final String CONVERTER_ID = "test.dynamicConverter";

	private MethodExpression delegateDecodeProcessor;

	private MethodExpression delegateEncodeProcessor;

	public DynamicConverter() {
	}

	@Override
	public Object getAsObject(FacesContext context, UIComponent component, String value) {
		if (delegateDecodeProcessor != null) {
			return delegateDecodeProcessor.invoke(context.getELContext(), new Object[] { context, component,
value });
		}
		return value;
	}

	@Override
	public String getAsString(FacesContext context, UIComponent component, Object value) {
		if (delegateEncodeProcessor != null) {
			return (String) delegateEncodeProcessor.invoke(context.getELContext(),
					new Object[] { context, component, value });
		} else {
			return value == null ? null : value.toString();
		}
	}

	public MethodExpression getDelegateDecodeProcessor() {
		return delegateDecodeProcessor;
	}

	public void setDelegateDecodeProcessor(MethodExpression delegateDecodeProcessor) {
		this.delegateDecodeProcessor = delegateDecodeProcessor;
	}

	public MethodExpression getDelegateEncodeProcessor() {
		return delegateEncodeProcessor;
	}

	public void setDelegateEncodeProcessor(MethodExpression delegateEncodeProcessor) {
		this.delegateEncodeProcessor = delegateEncodeProcessor;
```

```
		}

}
```

C. ConverterHandler – DynamicConverterHandler

```
package test;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.view.facelets.ConverterConfig;
import javax.faces.view.facelets.ConverterHandler;
import javax.faces.view.facelets.MetaRule;
import javax.faces.view.facelets.MetaRuleset;
public class DynamicConveterHandler extends ConverterHandler {

    private static final MetaRule DELEGATE_DECODE_PROCESSOR = new MethodExpressionMetaRule("delegateDecodeProcessor",
                    Object.class, new Class[] { FacesContext.class, UIComponent.class, String.class });

    private static final MetaRule DELEGATE_ENCODE_PROCESSOR = new MethodExpressionMetaRule("delegateEncodeProcessor",
                    String.class, new Class[] { FacesContext.class, UIComponent.class, Object.class });

    public DynamicConveterHandler(ConverterConfig config) {
            super(config);
    }
    @Override
    protected MetaRuleset createMetaRuleset(Class type) {
            MetaRuleset metaRuleset = super.createMetaRuleset(type);
            //super class ConverterHandler will directly handle this, so no need for any MetaRule to act upon it
            metaRuleset.ignore("converterId");
            metaRuleset.addRule(DELEGATE_DECODE_PROCESSOR);
            metaRuleset.addRule(DELEGATE_ENCODE_PROCESSOR);
            return metaRuleset;
    }
}
```

D. Taglib

```
<tag>
            <tag-name>dynamicConverter</tag-name>
            <converter>
                        <converter-id>test.dynamicConverter</converter-id>
                        <handler-class>test.DynamicConveterHandler</handler-class>
            </converter>
            <attribute>
                        <description>Converter Id against which Converter will be created</description>
                        <name>converterId</name>
                        <required>true</required>
                        <type>java.lang.String</type>
            </attribute>
            <attribute>
                        <description>Delegated Method Expression to perform the actual conversion during decode</description>
                        <name>delegateDecodeProcessor</name>
                        <required>false</required>
                        <type>javax.el.MethodExpression</type>
            </attribute>
            <attribute>
                        <description>Delegated Method Expression to perform the actual conversion during encode</description>
                        <name>delegateEncodeProcessor</name>
                        <required>false</required>
                        <type>javax.el.MethodExpression</type>
            </attribute>
</tag>
```

E. Delegated Dynamic Bean working as a Converter

```
package test;

import javax.enterprise.context.RequestScoped;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.inject.Named;

@Named
@RequestScoped
public class DelegatedDynamicConverter {
```

```
	public Object getAsObject(FacesContext context, UIComponent component, String value) {
		return value;
	}

	public String getAsString(FacesContext context, UIComponent component, Object value) {
		if (value == null)
			return "";
		return "Test" + value.toString();
	}

}
```

Xhtml Usage

```
<h:outputText value="#{testController.firstName}">
<tag:dynamicConverter converterId="test.dynamicConverter" delegateDecodeProcessor="#{delegatedDynamicConverter.getAsObject}"
                                                          delegateEncodeProcessor="#{delegatedDynamicConverter.getAsString}"/>
</h:outputText>
```

In the above element, there is no need for delegateDecodeProcessorbecause it is a UIOutput type component; it is included just to show its usage.

The above mechanism can be applied to properties of UIComponent through their ComponentHandler.

IV. State Management of AttachedObject

By default,Converter type AttachedObjectis set in UIOutput and that is where their state is managed via saveState and restoreState. In this context, save and restore statesare only usedwhen a converter has at least one attribute. JSF implementation provided converters such as javax.faces.Long and javax.faces.Doublethat donot have any attributes so they donot need any state management. But, other converters such as "javax.faces.DateTime" and "javax.faces.Number" have attributes for it to manage the conversion process, and hence both MyFaces and Mojarra provide state management on these converters.State Managementis done by implementing these converters with interface PartialStateHolder (which itself extends from StateHolder).

A. DynamicConverter - StateManagement

For a better understanding of State Management using DynamicConverter, PartialStateHolder and associated attributes, refer to sample source code below.

```
package test;

import javax.el.MethodExpression;
import javax.faces.component.PartialStateHolder;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.FacesConverter;

@FacesConverter(DynamicConverter.CONVERTER_ID)
public class DynamicConverter implements Converter, PartialStateHolder {

	public static final String CONVERTER_ID = "test.dynamicConverter";

	private MethodExpression delegateDecodeProcessor;

	private MethodExpression delegateEncodeProcessor;

	//properties for State Management
	private boolean initialStateMarked = false;
	private boolean _transient = false;

	public DynamicConverter() {
	}

	@Override
	public Object getAsObject(FacesContext context, UIComponent component, String value) {
		if (delegateDecodeProcessor != null) {
			return delegateDecodeProcessor.invoke(context.getELContext(), new Object[] { context, component,
value });
		}
		return value;
	}

	@Override
	public String getAsString(FacesContext context, UIComponent component, Object value) {
		if (delegateEncodeProcessor != null) {
			return (String) delegateEncodeProcessor.invoke(context.getELContext(),
```

```
                                                    new Object[] { context, component, value });
                } else {
                        return value == null ? null : value.toString();
                }
        }

        public MethodExpression getDelegateDecodeProcessor() {
                return delegateDecodeProcessor;
        }

        public void setDelegateDecodeProcessor(MethodExpression delegateDecodeProcessor) {
                this.delegateDecodeProcessor = delegateDecodeProcessor;

                /*      This ensure to clear the marked initial state - if this setter got invoked after the view was build
                        We know that initial state is marked true after view is build
                        In most cases these setter methods are only invoked during buildView
                */
                clearInitialState();
        }

        public MethodExpression getDelegateEncodeProcessor() {
                return delegateEncodeProcessor;
        }

        public void setDelegateEncodeProcessor(MethodExpression delegateEncodeProcessor) {
                this.delegateEncodeProcessor = delegateEncodeProcessor;
                clearInitialState();
        }

        //Methods below are related to State Management

        @Override
        public Object saveState(FacesContext context) {

/*In normal scenario using Partial State saving - no state will be saved since these attribute setter methods won't be invoked
by application developers other than by JSF implementation when buildView occurs - and after buildView initial state is marked true
*/
                if (!initialStateMarked()) {
                        Object[] values = new Object[2];
                        values[0] = delegateDecodeProcessor;
                        values[1] = delegateEncodeProcessor;
                        return values;
                }
                return null;
        }

        @Override
        public void restoreState(FacesContext context, Object state) {
                if (state != null) {
                        Object[] values = (Object[]) state;
                        delegateDecodeProcessor = (MethodExpression) values[0];
                        delegateEncodeProcessor = (MethodExpression) values[1];
                }
        }

        @Override
        public boolean isTransient() {

                return _transient;
        }

        @Override
        public void setTransient(boolean newTransientValue) {
                this._transient = newTransientValue;
        }

        @Override
        public void markInitialState() {
                this.initialStateMarked = true;
        }

        @Override
        public boolean initialStateMarked() {
                return initialStateMarked;
        }
```

```
	@Override
	public void clearInitialState() {
		this.initialStateMarked = false;
	}
}
```

V. CONCLUSION

This paper presents the inner working of MetaRule API and how it works under the hood to provide a decoupled but still a cohesive design to weave the attributes on AttachedObjects and UIComponents.The paper takes through a dynamic type of Converter to show how these API's are wired together to provide the desired functionality. The paper also considers various API's that work together to handler diverse types of AttachedObjects and how these are set in their parent UIComponent.

REFERENCES

[1] JavaServer Faces 2.2 API, website - https://javaserverfaces.github.io/docs/2.2/javadocs/index.html?overview-summary.html

[2] JavaServer Faces Tutorial by Oracle, website - https://docs.oracle.com/javaee/7/tutorial/jsf-intro.htm#BNAPH

[3] MyFaces 2.2 – website - http://myfaces.apache.org/core22/