Java Server Faces – Transformation of Face let to UIViewRoot

Vijay Kumar Pandey

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

Abstract-*This document provides deep dive insight and guidance necessary for application architects, software designers and software programmers who develop enterprise web-based applications utilizing JavaServer Faces (JSF). The guidance is meant to provide an in-depth understanding to the reader, on how XHTML code is processed and converted to Facelet and then to UIViewRoot components, allowing the reader to better design, architect and develop 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 – *JSF, JSF2.2, XHTML, Facelet, UIViewRoot, JSF Component Tree, JSF Lifecycle, MyFaces, PrimeFaces, OmniFaces*

I. INTRODUCTION

This document describes the complex processing that occurs during the conversion of an XHTML file to fill up a JSF component tree *UIViewRoot*. It only considers *Faceletsfor* discussion; it does not consider JSPsbecause they are a deprecated view technology with respect to JSF 2.2. *Facelets* (in this case XHTMLfiles) are converted to *Facelet* objects and then to components that fill up the *UIViewRoot*, following recursive processing; on the other hand, JSPs werecompiled to *Servlets*.The default *Facelet* implementation provided by both *Oracle's Mojarra*&*Apache's MyFaces*(v2.2.12 considered for the purposes of this article) is XHTML. This article will delve into the detailsof the processing that occurs in the implementation engines and how they convert a plain vanilla XHTML to a component tree *UIViewRoot*. The discussion is supported using code samples, that uses*PrimeFaces* 6.1 and *OmniFaces* 2.6 open source projects.

It is common knowledge that the JSF Lifecycle consists of six distinct phases, i.e., *RESTORE_VIEW, APPLY_REQUEST_VALUES, PROCESS_VALIDATIONS, UPDATE_MODEL_VALUES, INVOKE_APPLICATION, and RENDER_RESPONSE*. This discussion focuses on the RESTORE_VIEW and RENDER_RESPONSE phases that are primarily involvedin the conversion of XHTML to *UIViewRoot*.

II. SAMPLE FACELETSCODE

The code samples outlined below are utilized to help the reader follow the discussion:

A. *template.xhtml*

This code provides a mechanism to easily configure common tags for header, footer, and menu objects, for a page among others.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html lang="en"
      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:p=http://primefaces.org/ui"
      xmlns:c="http://xmlns.jcp.org/jsp/jstl/core"
      xmlns:o="http://omnifaces.org/ui">

  <h:head>
    <title>
      <ui:insert name="title">Default Title</ui:insert>
    </title>
  </h:head>
      <h:body>
          <h:panelGroup layout="block">
              <ui:insert name="content" >
                  Default Content
              </ui:insert>
          </h:panelGroup>
      </h:body>
</html>
```

Figure 1

B. *test.xhtml*

This code represents the main XHTML file which will implement a certain use case, in a real-world application.

```
<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"
        template="/WEB-INF/faces/template/template.xhtml">
<ui:define name="title">    Test Title </ui:define>
<f:metadata>
    <f:viewAction action="#{testController.load}"/>
</f:metadata>
<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="#{testController.firstName}"
                id="firstName" />
        <f:facet name="footer">
            <p:commandButton value="Save"
                action="#{testController.save}"
                update="@form" />
        </f:facet>
      </p:panelGrid>
  </h:form>
</ui:define>
```

Figure 2

C. TestController.java

This code represents the main controller java file, which will implement a certain use case, it's basically invoked during Invoke Application phase.This controller class is referred from *test.xhtml*.

```
package test;

import java.io.Serializable;
import javax.faces.application.FacesMessage;
import javax.faces.context.FacesContext;
import javax.faces.view.ViewScoped;
import javax.inject.Named;

@Named
@ViewScoped
@SuppressWarnings("serial")
public class TestController implements Serializable {
        private String firstName;
        public String load() {return null; }
        public String save(){
                FacesContext.getCurrentInstance().addMessage(null,
                        new FacesMessage("Save action executed"));
                return null;
        }
        public String getFirstName() {return firstName; }
        public void setFirstName(String firstName) {
                this.firstName = firstName;
        }
}
```

Figure 3

III. XHTML TO FACELET TO UIVIEWROOTCONVERSION

The lifecycle of a JSF application begins when a user makes an HTTP request for a page and ends when the server responds with the response.The request-response JSF lifecycle handles two kinds of requests: *Initial Request* and *PostBack*. An *initial request* occurs when a user makes a request for a page for the first time. A *PostBack* request occurs when a user submits the form contained on a page that was previously loaded into the browser because of executing an *initial request*.

A *FacesServlet* (provided by JSF implementation) manages therequest-processing lifecycle for web applications and initializes the resources requiredby JSF technology.Before a JSF application can start processing requests, the web containerwill initialize this servlet with required resources.So, it is important to understand how a request is handled by *FacesServlet* and *Lifecycle*before understanding how a physical XHTML file gets converted to a *Facelet*.

A. FacesServlet

At application server start up,*FacesServlet* must have initialized itself. It is important to note that this servlet class is a final class and cannot be extended. The *init* method of this servlet is used to initialize Factory objects such as *FacesContextFactory* and *LifeCycleFactory*.Since *LifeCycle* instance is supposed to be shared across multiple requests, it can be initialized in this method.

The request is handled by the *service* method of the *FacesServlet*. The main object that is prepared here is *Faces Context*. *FacesContextFactory*has a *getFacesContext*method that creates (as needed) a new *FacesContext* object. The first argument of the method *getFacesContext*is of type Object; for the servlet request environment, this is the *ServletContext* object. *FacesContext.getExternalContext()*method returnsExternalContext, which is a wrapper around ServletContext, ServletRequest and ServletResponse. Also, FacesContext is a thread scoped object,which returns the same objecton the same thread until its *release* method has been invoked,which usually happensin the *final*clauseofthis servlet's *service* method.

After the creation of the *FacesContext*, *FacesServlet* delegates further processing of the request, to the *Lifecycle* object. *Lifecycle* class has two main methods*execute* and *render*. The *execute* method processes the first five phases of the *JSF Lifecycle* and the *render* method processes the *RENDER_RESPONSE* phase of the lifecycle. There is also an *attachWindow* method in the Lifecycle, which is related to *ClientWindow (not discussed in this article)* functionality. The below code snapshot describes the same:

```
"init" method – factory object and Lifecycle object initialization
FacesContextFactory facesContextFactory =
(FacesContextFactory)FactoryFinder.getFactory(FactoryFinder.FACES_CONTEXT_F
ACTORY);

LifecycleFactory lifecycleFactory = (LifecycleFactory)
      FactoryFinder.getFactory(FactoryFinder.LIFECYCLE_FACTORY);

Lifecycle lifecycle = lifecycleFactory.getLifecycle(lifecycleId);
   • Here lifecycleId is either application configured Lifecycle id via
     servlet or
     context param (javax.faces.LIFECYCLE_ID)or JSF implementation default

"service" method – request processing – main steps
FacesContext context =
facesContextFactory.getFacesContext(servletConfig.getServletContext(),
request, response, lifecycle);
lifecycle.attachWindow(context);
lifecycle.execute(context);
lifecycle.render(context);
```

Figure 4

B. Lifecycle Execute Method

In the *Lifecycle.execute()* method, the initial five phases get executed – obviously, it can return after any phase due to *responseCompleted* or *renderResponse* marked as true i.e.,*facesContext.getRenderResponse()* or *facesContext.getResponseComplete()* returning true.

It should be noted here that if *facesContext.getResponseComplete()* returns true then *Lifecycle.render()* will also not execute. For e.g., a use case may need to download a dynamic file, possibly in the *INVOKE_APPLICATION* phase, in which case the *inputStream*is written on the *outputStream*(*externalContext.getResponseOutputStream()*).After the stream has been written,it will need to be explicitly markedfor the response as completed on the *facesContext* via *facesContext.responseComplete()*.

Important pre-steps executed before the actual processing of the phases, include

- Setting the current phase id on the FacesContext - *facesContext.setCurrentPhaseId*(<<current phase id>>);
- *Flash.doPrePhaseActions(facesContext)* getting executed
- Any other pre-processing specific to that phase
- All the configured PhaseListener's*beforePhase*method getting executed for that phase
- Checkingfor *facesContext.getRenderResponse()* or *facesContext.getResponseComplete()*, if true (then further processing of this *execute*will stop)
- Actual processing of the phase is dependent on the previous step, i.e., *facesContext.getRenderResponse()* or *facesContext.getResponseComplete()* returning true.

The subsequent section will address the phases where a physical XHTML file is converted to *Facelets* and then to a component tree with*UIViewRoot* as the top level component.

C. Restore View Phase (LifeCycle Execute Method – Initial Request Scenario)

This phase plays a key role in the process of creating a *Facelet*, but involves only the *Facelet* object corresponding to the *f:metadata* tag in the XHTML. The name of this phase suggests that it is supposed to restore the view on the *PostBack* request, but in case of an initial request, it creates the *Facelet* for the *f:metadata* tag.

In the pre-phase action, the *initView* method on the *ViewHandler* is executed. This is basically to set the character encoding on the *ExternalContext*. Refer to the Javadoc of the method *calculateCharacterEncoding* for class *ViewHandler* to understand the algorithm for how encoding is calculated.

```
facesContext.getApplication().getViewHandler().initView(facesContext);
```

At this stage, the main processing in this phase of the lifecycle starts; of course, this occurs after *beforePhase* methods have executed for all *PhaseListeners* configured against this phase.

Up until this point, no *UIViewRoot* has been created i.e.,*facesContext.getViewRoot()* will return null. Before this root object can get created, *viewId* needs to be created. In a straightforward scenario, if the initial request's URLis something like http(s)//<<server>>/<<context-root>>/test/test.xhtml, then from the *servlet api*, the *viewId* for the request is determined as */test/test.xhtml*,which is the servlet path. To have a thorough understanding of algorithm of how a *viewId* is calculated for a non-portlet type of request, refer to the code below:

```
String viewId = (String) externalContext.getRequestMap().
        get("javax.servlet.include.path_info");
If (viewId == null)
viewId = externalContext.getRequestPathInfo();
if (viewId == null)
viewId = (String)externalContext.getRequestMap().
        get("javax.servlet.include.servlet_path");
if (viewId == null){
viewId = externalContext.getRequestServletPath();
```

Based on the *viewId*, *ViewDeclarationLanguage* (VDL) object is determined from the *ViewHandler* (*MyFaces*offers an implementation of the Facelet based VDL). VDL can be determined through *viewHandler.getViewDeclarationLanguage(facesContext, viewId)*. For an XHTML based *Facelet*, once the VDL is determined, ViewMetaData is created via:

```
ViewMetadata metadata = vdl.getViewMetadata(facesContext, viewId);
```

The above object is a *Facelet* based *ViewMetadata*with *viewId*. Once the above object is created, *createMetadataView* method is executed and returns a *UIViewRoot*. This is the method where the actual conversion from an XHTML file to a *Facelet* object occurs, but as stated earlier, this *Facelet* is only related to a *f:metadata* tag in the XHTML file.

```
UIViewRootviewRoot = metadata.createMetadataView(facesContext)
```

The *UIViewRoot* returned, has only *UIViewAction(s)* and *UIViewParameter(s)* as children grouped under a common parent of type *facet*with name of *UIViewRoot.METADATA_FACET_NAME*. This facet is a direct child of *UIViewRoot*.Before the *ViewMetadata*, related components are created through its *facelet*, *UIViewRoot* needs to be created first, via

```
UIViewRootviewRoot = facesContext.getApplication().
getViewHandler().createView(facesContext, "<<viewId>>");
```

```
Main processing inside createView: This method goes through the
ViewDeclarationLanguage.
        createView(facesContext,<<viewId>>) to create the
UIViewRoot.
UIViewRootviewRoot = (UIViewRoot) facesContext.
    getApplication().createComponent.
(facesContext, UIViewRoot.COMPONENT_TYPE, null);
```

There is no *renderer* associated with the *UIViewRoot* component, hence the third argument is null. Sobased on the component type, JSF implementation looks for the implementation class

and creates a new instance of the component (all components, either provided by the JSF implementation or custom component) through its no-arg constructor.

```
Class<? extends UIComponent>componentClass = <<fetch the implementation class based on component type>>
UIComponent component = componentClass.newInstance();
```

At this moment, various annotations tagged on the component such as *ListenerFor*, *ListenersFor*, *ResourceDependency* and *ResourceDependencies*, are handled.

Once the *UIViewRoot* gets created, then *locale* and *renderkitId*are set on the root object through *ViewHandler* class, that has methods such as *calculateLocale(facesContext)* and*calculateRenderKitId(facesContext)*. *MyFaces* also sets the *viewId* on the *UIViewRoot*.

```
ViewHandlerviewHandler =
facesContext.getApplication.getViewHandler();
viewRoot.setLocale(viewHandler.calculateLocale(context));
viewRoot.setRenderKitId(viewHandler.calculateRenderKitId(con
text));
viewRoot.setViewId(<<viewId>>);
```

From this point on,*UIViewRoot*may be used directly to access the current *viewId* of the request. Also, in *ViewMetadata*,a related *Facelet*is created and then based on this *Facelet*,*UIViewRoot*is populated with the components related to *f:metadata*.

ViewMetadataFacelet creation: There are two types of *Facelets* - the normal *View Facelets* and the *View Metadata Facelets*. The*View Metadata Facelets* correspond to *<f:metadata>* tag in the XHTML.

This part of the *Facelet*does not create the full view *Facelets*,because there is no need to handle other kind of tags present in the physical *Facelet* file (XHTML),other than the *f:metadata*. Refer to *JavaDoc*, for class *FaceletCache* for more informationabout this. A physical *Facelet* file corresponds to an XHTML file, which is basically an XML file. Therefore, to get hold of a *Facelet* object, the XML should be first parsed. *MyFaces* internally uses a fast SAX compiler to achieve this parsing. The SAX compiler class is *javax.xml.parsers.SAXParser* and its parse method takes *org.xml.sax.helpers.DefaultHandler* as one of the arguments, which can handle events generated from the parser. *MyFaces* creates these custom handlers to handle the events generated by the parser.

One of the methods from *org.xml.sax.helpers.DefaultHandler* to handle the transformation from an XML to *Facelets*, is *startElement(String uri, String localName, String qName, Attributes attributes)*.In this method,*org.xml.sax.Attributes* are converted to *javax.faces.view.facelets.TagAttribute*. For this type of *Facelet*, anything other than *f:metadata* is not considered.

Using the parameters of the method *startElement*,*javax.faces.view.facelets.Tag* object is created and passed further for processing.

TagDecorator:One of the steps of the *execute* method, that occurs at this point is the transformation of the *Tag* object using *javax.faces.view.facelets.TagDecorator* into a new *Tag* object per the decoration logic(In JSF 2.2, there is a default implementation of *TagDecorator*already available – refer to *TagDecorator*in *JavaDoc*).The sample code used for this article does not call for any tag decoration except processing using the default *TagDecorator* in JSF.

Once the XML parsing is completed, there may be various *TagHandlers*comprising of *Tag* along with the next *TagHandler*. To get the *Facelet* from these *TagHandlers*, typical JSF implementations create a top level *TagHandler*(for e.g., *EncodingHandler*in *MyFaces*) which becomes the starting *TagHandler* inside JSF implementation of *Facelet*. Here is the chain of *TagHandlers* created for this *Facelet*.

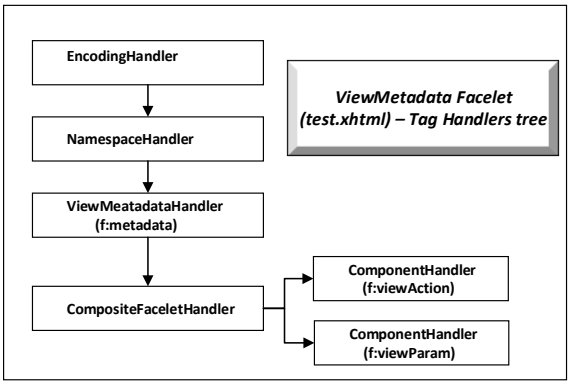

Figure 5

Tag Handler Factories:During the creation of the *Tag Handlers*, *Tag Handler* factories play a key role in setting up the handler objects pertaining to *tag*, *component*, *converter*, *validator*. These factories are specific to JSF implementation, but their purpose is the same, i.e., to set up or provide a mapping from XML markup on the XHTML page to its corresponding handler. For e.g.,*metadata*markup in XML page (which is associated with the namespace http://xmlns.jcp.org/jsf/core) is mapped as a *TagHandler* in one of the libraries (*CoreLibrary*), while *viewAction* and *viewParam* with the same namespace are mapped to component (without any renderer and no specific handler). When a particular component has no specific component handler associated with it, the JSF engine will associate *ComponentHandler* class as the default handler.

Like the core library, there are other types of standard libraries in each JSF implementation, such as *HtmlLibrary*, *JstlCoreLibrary* etc. which help in creating the proper mapping between XML and the namespace to its corresponding handlers. These handlers (if component handler) in turn, help in creating the actual components.

D. Render Response Phase (LifeCycleRender Method)

In response to an Initial Request (as against a *PostBack*request), this phase basically fills up the *UIViewRoot* with the components. However, before this occurs, the *view Facelet* is first created (like*ViewMetadataFacelet* creation in the *RESTORE_VIEW* phase).

View FaceletCreation:As described in the previous section, there are two types of *Facelets* - the normal *View Facelets* and the *View Metadata Facelets*. The*View Metadata Facelet* is created with the help of the method *buildView* in VDL. Before that occurs, if *id* is not present on the *UIViewRoot*, a unique id is created and set. The concept around building *View Facelets* is like building the *View Metadata Facelet*with the exception that the entire XMLmarkup is used to create tag handlers, component handlers, etc (as against using just the *f:metadata* tag in the RESTORE_VIEW Phase). Once the chain of handlers is created and set in the top level *Facelet*, various components get created when *apply* method is executed on the Facelet.

In the next figure 6, *EncodingHandler* is the root of the handler, which starts the building process of the component tree. The entire process starts once the apply method is executed on the *Facelet*(which has this *EncodingHandler*, as its main root handler), which in turn executes recursively the next handler and so on,until the whole chain is executed eventually creating the component tree.

```
package javax.faces.view.facelets;
public abstract class Facelet {
  public abstract void apply(FacesContextfacesContext, UIComponent parent) throws IOException;
}
```

The above method in the *Facelet* passes the *UIViewRoot* as the parent component. Once a *ComponentHandler* is encountered, a component of that type is created and then the next handlers*apply* method is executed.*ComponentHandler* extends from *DelegatingMetaTagHandler*with the following functions, to provide the capability of calling the chained handlers in recursion till the whole tree is built.

```
public void apply(FaceletContextctx, UIComponent parent) throws IOException{
getTagHandlerDelegate().apply(ctx, parent);  //this method internally calls the //applyNextHandler method for chaining the next handler
  }
```

```
public void applyNextHandler (FaceletContextctx, UIComponent c) throws IOException{
nextHandler.apply (ctx, c);
  }
```

The diagram below provides the actual tag handlers tree present in the view *Facelet*.

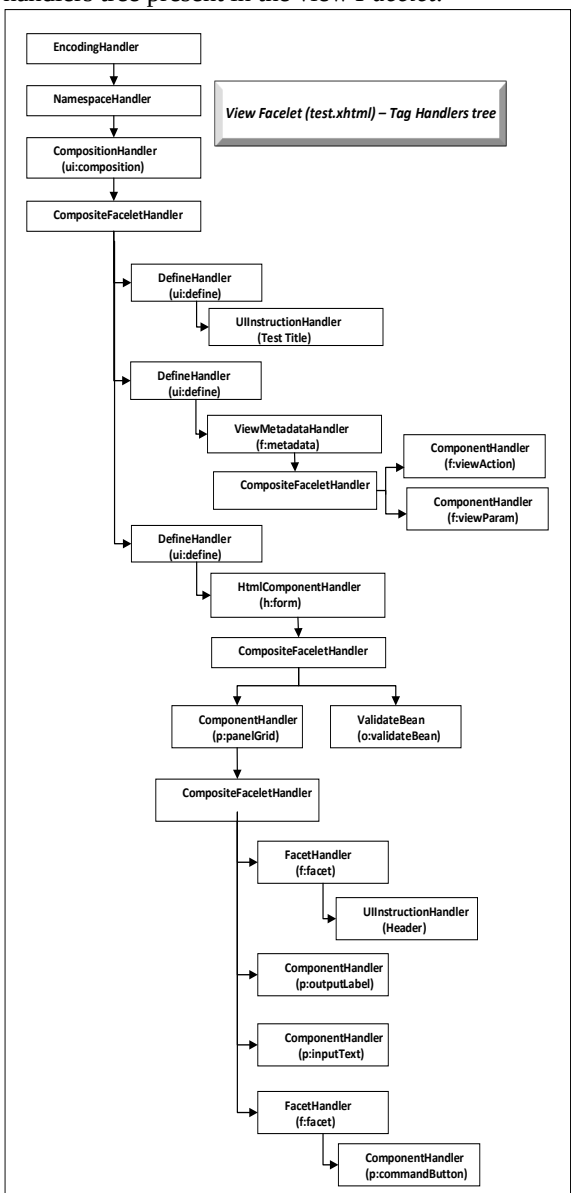

Figure 6

Since the current view has the template specified, *CompositionHandler* includes the actual *template Facelet*, which can also add new handlers – this execution is explained next.

The *apply* method of *CompositionHandler*executes the following method to include the *template Facelet*. This method is present in the *FaceletContext*.

```
public abstract void includeFacelet(UIComponent parent, String relativePath);
```

Of course, in the above method, the *UIComponent* object passed will be *UIViewRoot*. As explained

earlier in this document, the *template Facelet* is built using the same strategy of parsing it with the SAX compiler and building the tree.

InsertHandler: The *apply* methodchecks if the main *View Facelet* has the *DefineHandler* with the same name directly under *UICompositionHandler*. If found,that *DefineHandler* gets applied (i.e.,*DefineHandler's*nextHandler gets applied);if not found, nextHandler of *InsertHandler* gets applied. The diagram below depicts the tag handler tree of the *template.xhtmlview facelet*.

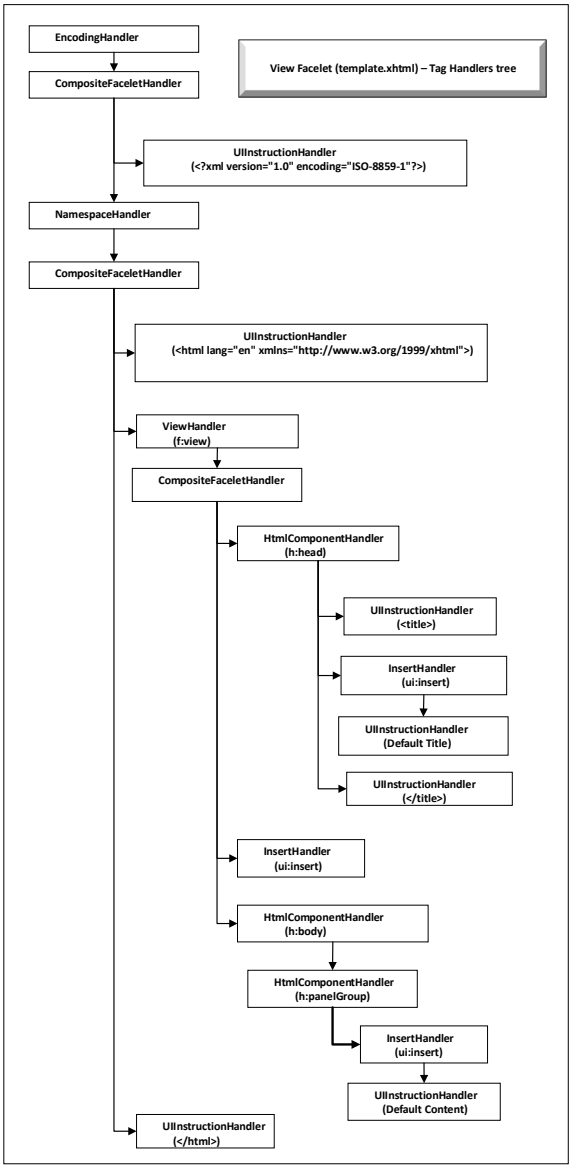

Figure 7

UIViewRoot – Component Tree:The diagram below depicts the component tree built in *UIViewRoot*, with the help of the handlers described previously:

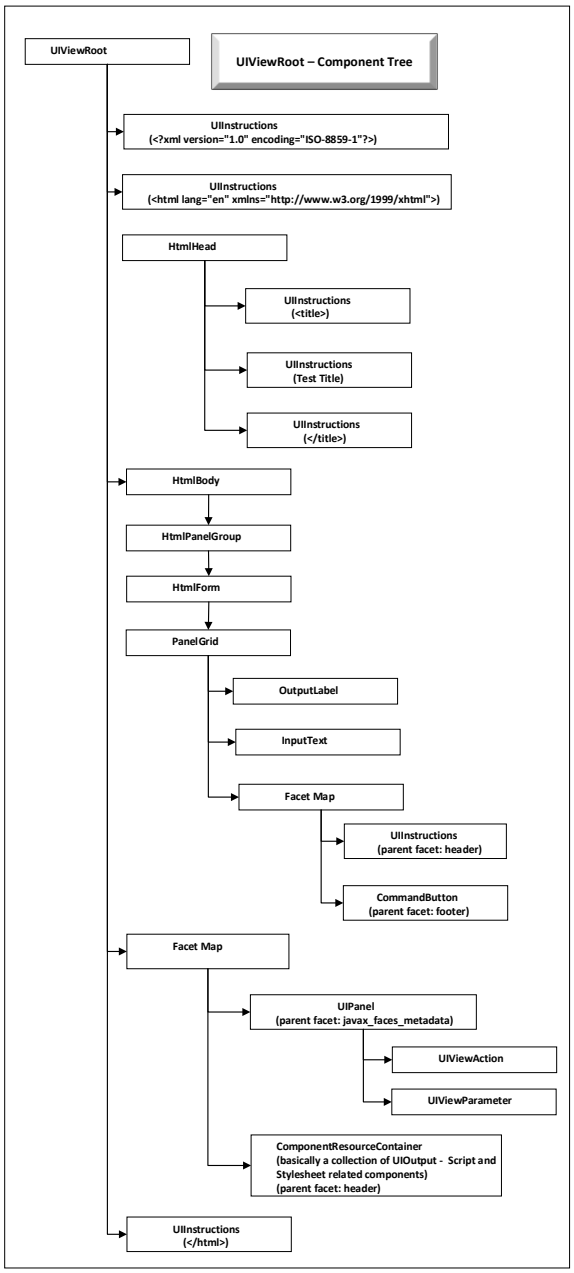

Figure 8

The table below provides a detailed description of the *UIViewRoot* Component Tree.

S. No.	Component (UIComponent)	Parent Component	Tag Handler	Remarks
1.	UIViewRoot	None	None	There are no tag handlers to create this, but as discussed above this component gets created in the RESTORE_VIEW phase.
2.	UIInstructions[1]	UIViewRoot	UIInstructionHandler[1]	This is JSF implementation specific component, and refers to any text or EL text outside of any tag markup element name or attribute name or value. This component contains the text from *template.xhtml* i.e., <?xml version="1.0" encoding="ISO-8859-1"?>
3.	UIInstructions[1]	UIViewRoot	UIInstructionHandler[1]	This component contains the text from *template.xhtml* i.e., basically the html tag "html" <html lang="en" xmlns="http://www.w3.org/1999/xhtml">
4.	HtmlHead	UIViewRoot	HtmlComponentHandler[1] This handler basically extends from ComponentHandler	This component comes from the *template.xhtml* – tag markup*h:head*
5.	HtmlBody	UIViewRoot	HtmlComponentHandler[1]	This component comes from the *template.xhtml* – tag markup*h:body*
6.	UIInstructions[1]	UIViewRoot	UIInstructionHandler[1]	This component contains the text from template.xhtml i.e., basically end html tag for "html"*</html>*
7.	UIInstructions[1]	HtmlHead	UIInstructionHandler[1]	This component contains the text from *template.xhtml* i.e., *<title>*
8.	UIInstructions[1]	HtmlHead	UIInstructionHandler[1]	This component contains the text from *test.xhtml* i.e., *Test Title*
9.	UIInstructions[1]	HtmlHead	UIInstructionHandler[1]	This component contains the text from *template.xhtml* i.e., *</title>*
10.	HtmlPanelGroup	HtmlBody	HtmlComponentHandler[1]	This component comes from the *template.xhtml* – tag markup*h:panelGroup*
11.	HtmlForm	HtmlPanelGroup	HtmlComponentHandler[1]	This component comes from the *test.xhtml* – tag markup*h:form*
12.	PanelGrid	HtmlForm	ComponentHandler	This component comes from the *test.xhtml* – tag markup*p:panelGrid*
13.	OutputLabel	PanelGrid	ComponentHandler	This component comes from the *test.xhtml* – tag markup*p:outputLabel*
14.	InputText	PanelGrid	ComponentHandler	This component comes from the *test.xhtml* – tag markup*p:inputText*
15.	UIInstructions[1]	PanelGrid (contained against the facet "header")	UIInstructionHandler[1]	This component contains the text from *test.xhtml* i.e., *Header*
16.	CommandButton	PanelGrid (contained against the facet "header")	ComponentHandler	This component comes from the *test.xhtml* – tag markup*p:commandButton*
17.	UIPanel	UIViewRoot (contained against the facet "javax_faces_metadata")	None	This component is created because *f:metadata* is present in the *text.xhtml* (per JSF spec)
18.	UIViewAction	UIPanel	ComponentHandler	This component comes from the *test.xhtml* – tag markup*f:viewAction*
19.	UIViewParameter	UIPanel	ComponentHandler	This component comes from the

S. No.	Component (UIComponent)	Parent Component	Tag Handler	Remarks
				test.xhtml – tag markup*f:viewParam*
20.	ComponentResourceContainer[1,2] (collection of several UIOutput components)	UIViewRoot (contained against the facet "header")	None	These UIOutput basically refers to various stylesheets and javascript resources present on PrimeFaces related components.

[1]*These components and handlers are specific to JSF implementation.* [2]*See below for description of Component Resources*

Table1

The following is a mapping of java Classes to fully qualified Class Names (the mapping only provides classes and class names that are part of JSF API, PrimeFaces and not those that are specific to a JSF Implementation):

- *UIViewRoot-javax.faces.component.UIViewRoot*
- *HtmlHead - javax.faces.component.html.HtmlHead*
- *HtmlBody - javax.faces.component.html.HtmlBody*
- *HtmlPanelGroup - javax.faces.component.html.HtmlPanelGroup*
- *HtmlForm - javax.faces.component.html.HtmlForm*
- *PanelGrid – org.primefaces.component.panelgrid.PanelGrid*
- *OuputLabel - org.primefaces.component.outputlabel.OutputLabel*
- *InputText - org.primefaces.component.inputtext.InputText*
- *CommandButton - org.primefaces.component.commandbutton.CommandButton*
- *UIPanel – javax.faces.component.UIPanel*
- *UIViewAction - javax.faces.component.UIViewAction*
- *UIViewParameter - javax.faces.component.UIViewParameter*
- *UIOutput - javax.faces.component.UIOutput*
- *ComponentHandler – javax.faces.view.facelets.ComponentHandler*
- *ResourceDependency – javax.faces.application.ResourceDependency*

Component Resources:Components have *ResourceDependency* annotation to let the JSF engine be aware of resource dependencies. Therefore, when a particular component is created, one of the subsequent steps is to examine these annotations and if present, createsa corresponding *UIOutput* component,and set attributes on that component such as *name, library, target*.This *UIOuput* component is then added to *UIViewRoot* via *UIViewRoot.addComponentResource* method. This method adds these *UIOutput* components inside a *Facet* named *header* or if a *target* attribute on *ResourceDependency* was specified, the method addsthe *UIOutput* components against that *Facet* name.For. e.g.,*InputText* component of *PrimeFaces* has the following *ResourceDependency* annotations.

```
@ResourceDependencies({
        @ResourceDependency(library="primefaces",
        name="components.css"),
        @ResourceDependency(library="primefaces",
        name="jquery/jquery.js"),
        @ResourceDependency(library="primefaces",
        name="core.js"),
        @ResourceDependency(library="primefaces",
        name="components.js")
})
```

PostBack Request:During *PostBack* request, the component tree is built in the RESTORE_VIEW phase of the Lifecycle.

```
UIViewRootviewRoot = viewHandler.restoreView(facesContext,
viewId);
```

Overriding Components:Components provided as part of the JSF API or any other open source JSF component library such as *PrimeFaces*, may be overridden by providing custom components in the *faces-config.xml*. For e.g.,the following code outlines a way to provide a custom implementation for *PrimeFacesInputText*:

```
<component>
        <component-
type>org.primefaces.component.InputText</component-type>
        <component-class>class extending
(org.primefaces.component.inputtext.InputText)</component-
class>
</component>
```

IV. Conclusion

This paper presents a thorough deep dive understanding of how a JSF 2.2 based framework converts a XHTML physical file to multiple *Facelet* java objects and then to *UIViewRoot*. This paper provides sample code to make it easy for the readers to easily follow this complex conversion. This understanding will be helpful for software architects and designers, involved in building complex enterprise web based applications, using JSF 2.2 technology and Java Enterprise Edition (JEE) 7 platform.

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 - http://myfaces.apache.org/core22/, website -

[4] PrimeFaces showcase, website - https://www.primefaces.org/showcase/

[5] PrimeFacesAPI, website - https://www.primefaces.org/docs/api/6.1/
[6] OmniFaces showcase, website - http://showcase.omnifaces.org/
[7] OmniFaces API, website - http://javadoc.io/doc/org.omnifaces/omnifaces/2.6.8