International Journal of Management, IT & Engineering
Vol. 9 Issue 6, June 2019,
ISSN: 2249-0558 Impact Factor: 7.119
Journal Homepage: http://www.ijmra.us, Email: editorijmie@gmail.com
Double-Blind Peer Reviewed Refereed Open Access International Journal - Included in the International Serial Directories Indexed & Listed at: Ulrich's Periodicals Directory ©, U.S.A., Open J-Gage as well as in Cabell's Directories of Publishing Opportunities, U.S.A

XHTMLtransformation to JSF ComponentTree

Vijay Kumar Pandey*

Abstract: -
Java Server Faces (JSF) is a component-oriented framework to manage html-basedrequest and response. The current version of JSF 2.3 included as part of Java Enterprise Edition (JEE 8) has the default view technology of Facelet. Facelet is the java object representation for the XHTML (Extensible HyperText Markup Language) in the JSF context. Enterprise project teams building JSF based system typically write the view layer code in an XHTML format. XHTML are physical files,which are processed by the JSF run time engine to convert it into a runtime java object as Facelet and which are then transformed to component tree based on the various tag handlers associated with the xml tags present in the xhtml file. This paperlays down the ground work for the complex processing behind the transformation of xhtml to component tree which can help software architects and developers to better understand this processand helping them to design and maintain their JSF based system in an effective manner. This document assumes that the reader has a basic understanding of JSF.

Keywords: - XHTML, JSF Component Tree, VDL (View Declaration Language), CDI (Context Dependency Injection), Facelet, UIViewRoot, JSF Lifecycle.

*Director of Technology Solutions, Intueor Consulting, Inc. Irvine CA – USA, pandey@intueor.com

I. INTRODUCTION

This papergoes in depth to describe the complex processing that occurs during the conversion of an XHTML file to fill up a JSF component tree *UIViewRoot*. XHTML files are converted to *Facelet* objects and then JSF component tree and for a http request this component tree resides in *UIViewRoot*.The default *Facelet* implementation provided by both *Oracle's Mojarra*&*Apache's MyFaces* is XHTML. This paper will lay out in detail the processing that occurs in the JSF runtime enginefor converting a physical XHTML file to a JSF component tree. The sample code provided in this paper utilizes open source projects such as *PrimeFaces* and *OmniFaces*. JSF Lifecycle consists of six phases, i.e., *RESTORE_VIEW, APPLY_REQUEST_VALUES, PROCESS_VALIDATIONS, UPDATE_MODEL_VALUES, INVOKE_APPLICATION, and RENDER_RESPONSE*. The transfornation of XHTML to Facelet and component tree mainy happens in the RESTORE_VIEW and RENDER_RESPONSE phases.

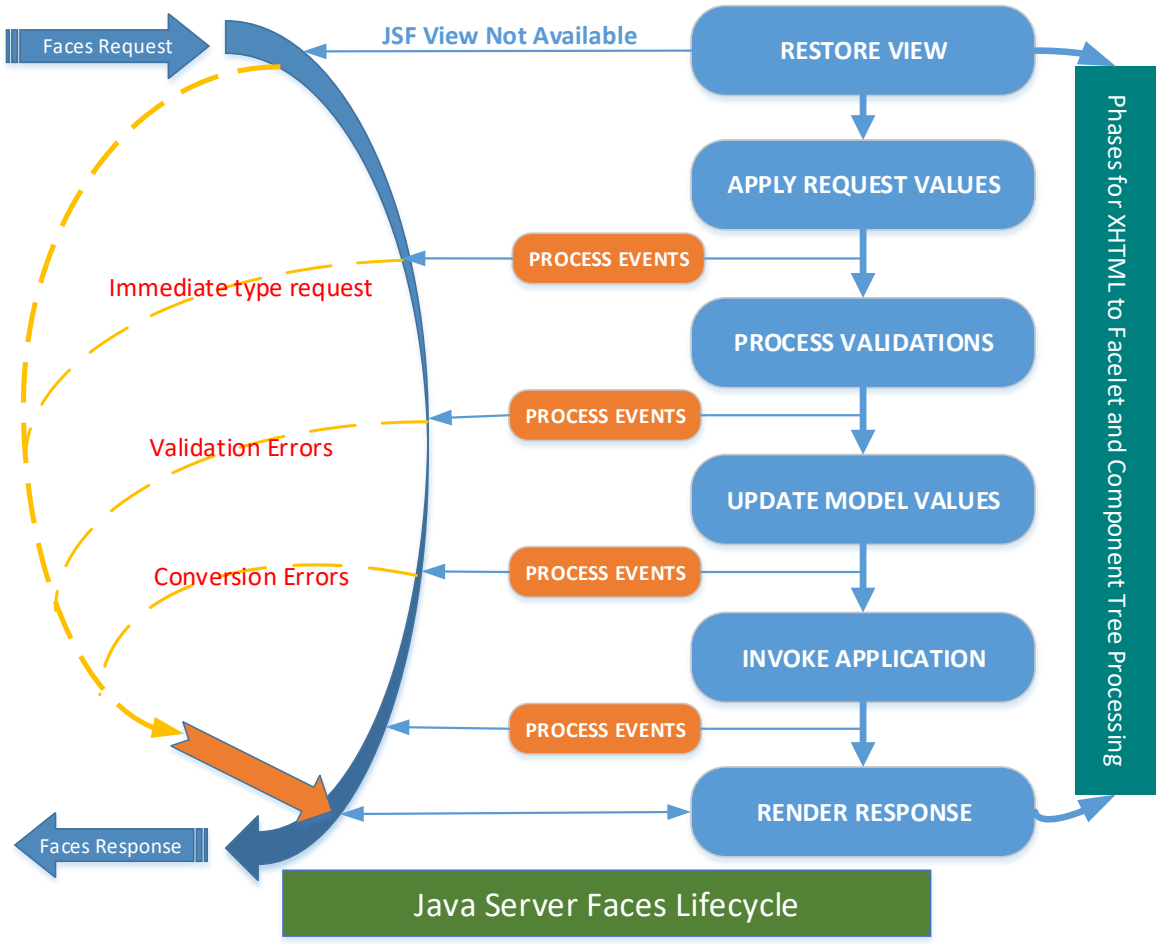

Figure 1

II. FACELETS CODE

The code outlined below are utilized to help the reader follow the processing of converting xhtml to component tree:

A. pagetemplate.xhtml

This template xhtml provides a mechanism to 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">

        <f:view contentType="text/html" >
          <h:head>
                        <title>
                                <ui:insert name="title">Page Title</ui:insert>
                        </title>
          </h:head>
                <ui:insert name="metadata"/>
          <h:body>
                        <h:panelGroup layout="block">
                                <ui:insert name="content" >
                                        Page Content
                                </ui:insert>
                        </h:panelGroup>
          </h:body>
        </f:view>
</html>
```

Figure 2

B. pagetest.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:p="http://primefaces.org/ui"

                                        xmlns:poc="http://test/ui"
                                        template="/WEB-INF/faces/template/templatepaper.xhtml">

 <ui:define name="title">XHTML Page Title</ui:define>
        <f:metadata>
                <f:viewAction action="#{controller.viewAction}"/>
        </f:metadata>
        <ui:define name="content">
          <h:form >
                <p:panelGrid columns="2" >
                  <f:facet name="header">Paper Title</f:facet>
                  <p:outputLabel value="Value" for = "value" />
                  <p:inputText value="#{controller.value}" id="value" />
                   <f:facet name="footer">
                        <p:commandButton value="Save" action="#{controller.execute}" update="@form" />
                   </f:facet>
                </p:panelGrid>
          </h:form>

    </ui:define>
</ui:composition
```

Figure 3

C. Controller.java

Controller is a CDI annotated bean that will process the request submitted by the above defined pagetest.xhtml. The *execute* method is invoked during the *INVOKE_APPLICATION* phase.

```
package article;

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
public class Controller implements Serializable {
	private String value;
	public void viewAction(){}

	public String execute(){
		FacesContext.getCurrentInstance().addMessage(null, new FacesMessage("Execute processed"));
		return null;
	}
	public String getValue() {
		return value;
	}
	public void setValue(String value) {
		this.value = value;
	}
}
```

Figure 4

III.XHTML TO JSF COMPONENT TREE TRANSFORMATION

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*.*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 container will initialize this servlet with required resources.The prerequisite to understand the transformation of XHTML to facelet is to understand how a request is handled by *FacesServlet*.

A. JSF ServletInitialization

Duringweb server start up, *FacesServlet*will get initialized.The *init* method of this servlet is used to initialize Factory objects such as *FacesContextFactory* and *LifeCycleFactory*. The request is handled by the *service* method of the *FacesServlet*. *FacesContext*is the main object that is created in this method which can then later be accessed through *ThreadLocal* mechanism from controller classes. *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 returns ExternalContext, which is a wrapper around ServletContext, ServletRequest and ServletResponse. *FacesContext*object is released from the thread in the *final*clauseof the 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.

B. Lifecycle Execute

In the *Lifecycle.execute()* method, the initial JSF 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 request creates a scenario where the response will be a downloaded file, possibly in the *INVOKE_APPLICATION* phase, in which case the stream will be written to the *outputStream*. After the stream has been written,it will need to be explicitly markedfor the response as completed on the *facesContext*through *facesContext.responseComplete()*.

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* butinvolves only the *Facelet* object corresponding to the *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 *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*.

```
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 URL is something like http(s)//<<server>>/<<context-root>>/article/pagetest.xhtml, then from the *servlet api*, the *viewId* for the request is determined as */article/pagetest.xhtml*, which is the servlet path. The code below provides the algorithm of how a *viewId* is calculated for a non-portlet type of request:

```
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*, VDL object is determined from the *ViewHandler*. 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*. After the creation of the above object,*createMetadataView* method is executed and returns a *UIViewRoot*. This is the where the actual conversion from an XHTML file to a *Facelet* object occurs, but as stated earlier, this *Facelet* is only related to a *metadata* tag in the XHTML file.

```
UIViewRoot viewRoot = metadata.createMetadataView(facesContext)
```

The above created *UIViewRoot* 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

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

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

There is no *renderer* associated with the *UIViewRoot* component, hence the third argument is null. Based on the component type, JSFengine 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();
```

During the component creation time, 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)*.

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

After the above code is executed, *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 *metadata*.

ViewMetadata Facelet initialization: There are two types of *Facelets* - the normal *View Facelets* and the *View Metadata Facelets*. The*View Metadata Facelets* correspond to *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 *metadata*.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 2.3, there is a default implementation of *TagDecorator*already available – refer to *TagDecorator*in *JavaDoc*).The code used for this paper does not needany custom tag decoration except processing using the default *TagDecorator*of 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*.

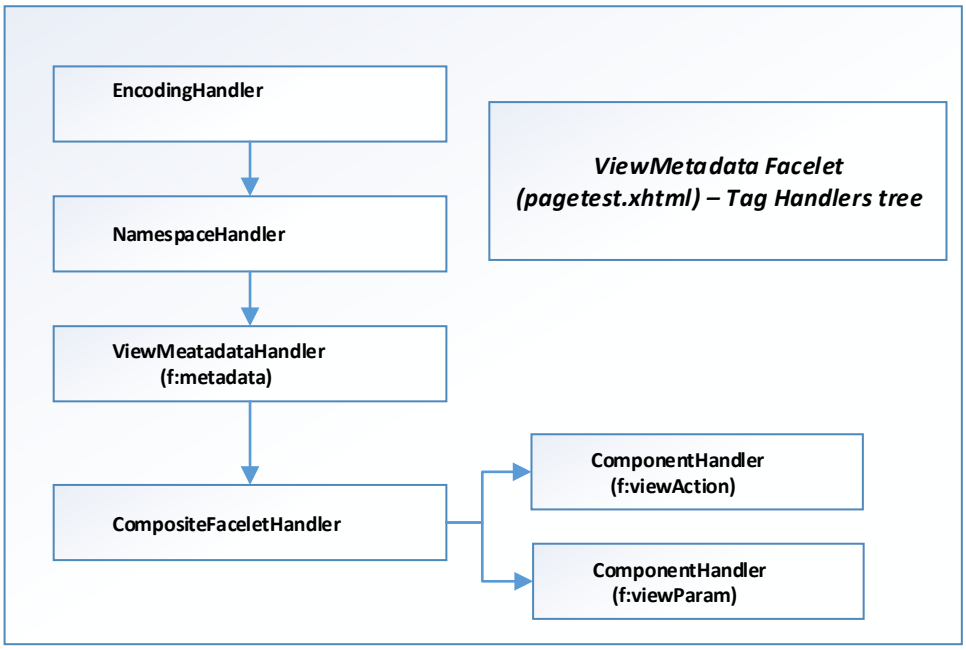

Figure 5

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 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 (LifeCycle Render 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 *ViewMetadata Facelet* creation in the *RESTORE_VIEW* phase). 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.The concept around building *View Facelets* is like building the *View Metadata Facelet*with the exception that the entire XML markup is used to create tag handlers, component handlers, etc (as against using just the *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(FacesContext facesContext, 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(FaceletContext ctx, UIComponent parent) throws IOException{
//this method internally calls the //applyNextHandler method for chaining the next handler
            getTagHandlerDelegate().apply(ctx, parent);
   }
public void applyNextHandler (FaceletContext ctx, UIComponent c) throws IOException{
      nextHandler.apply (ctx, c);
}
```

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

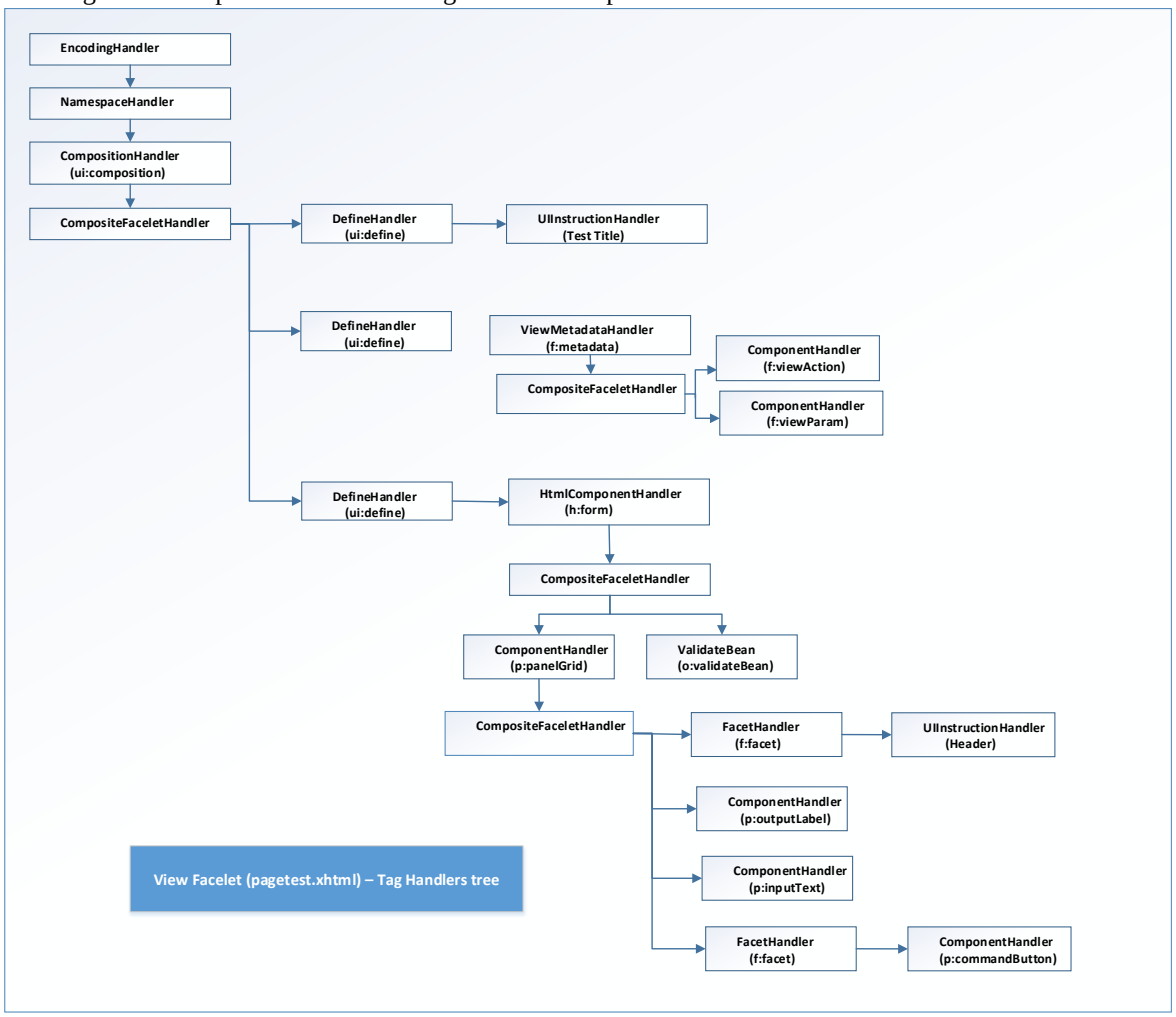

Figure 6

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.

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

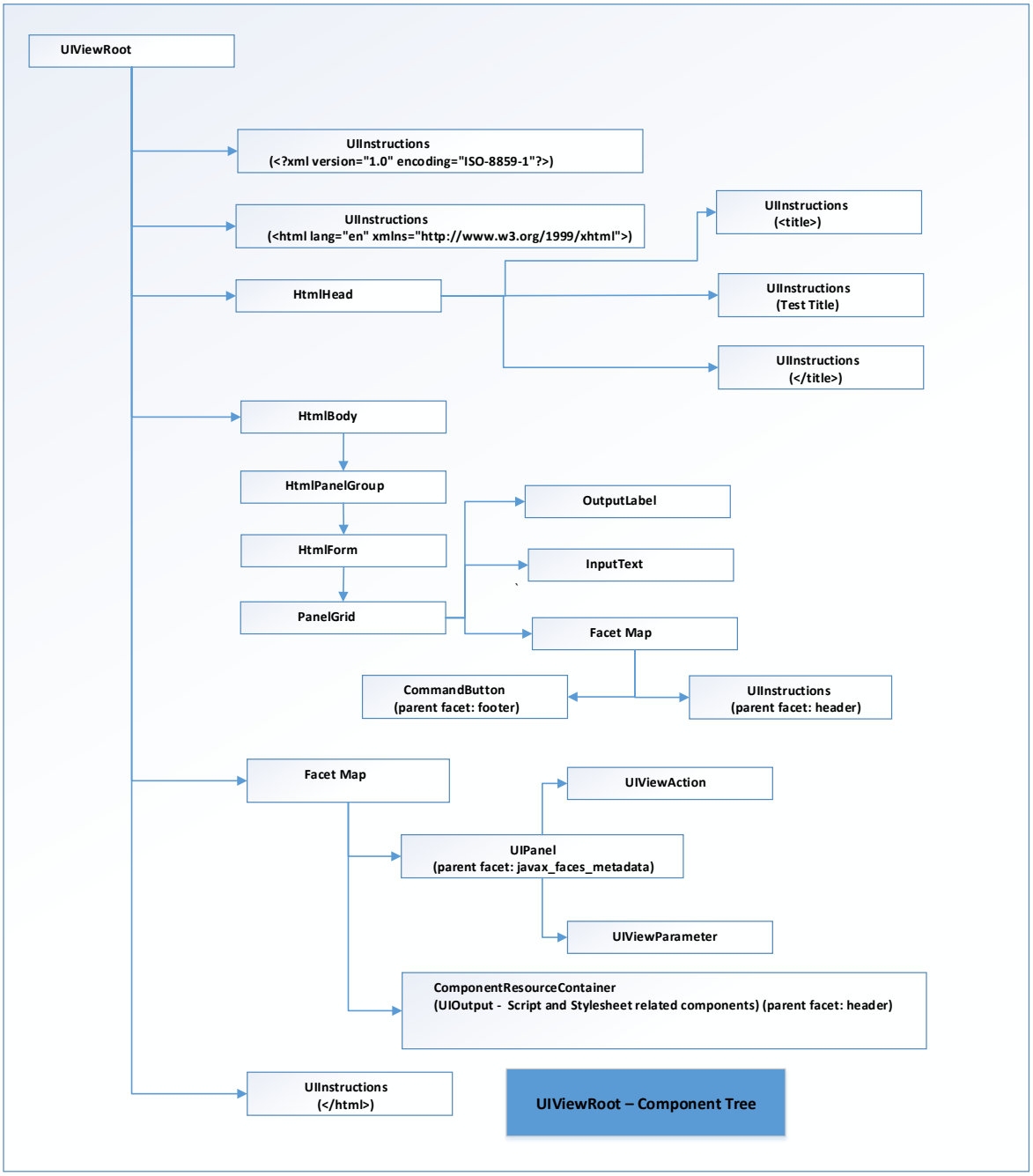

Figure 7

JSF Component Class Mapping:-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*

IV. CONCLUSION

This paper presents a thorough analysis of internals of JSF engine and covered topics such as FacesServlet initialization and how FacesContext is initialized, LifeCycle, ViewMetadata, VDL, TagDecoratorsto help readers understand how a JSF 2.3runtime eginetransforms a physical XHTML file to multiple *Facelet* java objects and and then to JSF component tree as *UIViewRoot*. This paper provides sample code to make it easy for the readers to easily follow this complex transformation of XHTMl file. The thorough understanding of the internals of JSF engine will help software architects and designers to design and maintain complex enterprise systems based on JSF 2.3.

REFERENCES

[1] JavaServer Faces 2.3 API, website - https://javaserverfaces.github.io/docs/2.3/javadocs/index.html
[2] JavaServer Faces 2.3 Tutorial, website - https://javaee.github.io/tutorial/jsf-intro.html#BNAPH
[3] ZEEF JSF, website - https://jsf.zeef.com/arjan.tijms
[4] ZEEF JEE 8, website - https://javaee8.zeef.com/arjan.tijms
[5] PrimeFaces 7API, website - https://www.primefaces.org/docs/api/7.0/
[6] OmniFaces3.3 API, website - http://omnifaces.org/docs/javadoc/3.3/
[7] MyFaces 2.3, website - https://myfaces.apache.org/core23/index.html/