State Management, Partial and Full State Saving Strategy - JavaServer Faces

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 the reader, on the concepts of State Management, Partial and Full State Saving strategy and the under the cover working of the State Management Methods in JSF 2.2. 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 and Facelets.*

Keywords – *State Management, Partial State Saving, State Saving Method,Facelet, JSF, JSF 2.2, MyFaces.*

I. Introduction

This article divesdeep into the JSF 2.2 State Management feature. JSFbeing mainly a stateful framework and HTML being stateless, JSF provides a very comprehensive state management feature with the introduction of Partial State Saving(PSS - via the web context config param javax.faces.Partial_State_Saving) feature. By default, PSS is enabled, meaning JSF does not save the full state of components during the RenderResponse phase of the JSF Life Cycle. The JSF framework saves only the partial state (also referred to as 'delta') which is then used on subsequent postBack requests during the Restore View phase. JSF provides an option through which the full state may also be saved for certain viewids, when PSS is set to true (javax.faces.Full_State_Saving_View_Ids, allowing comma separated viewids). Additionally, JSF provides an option to save the state either on the server or on the client using the web context config param (javax.faces.State_Saving_Method).The default value in MyFacesisserver and the ability for a programmer to change it to client. There are pros and cons that should be considered carefully while setting this parameter. At a high-level, saving the state on the server consumes more memory, may warrant session replication among servers set up in clusters, may also have to consider expired exceptions (most notably in applicationsthat utilize multiplebrowser tabs). On the other hand, saving the state on the client could lead to additional usage of network bandwidth, and may increase vulnerability to Cross-Site Request Forgery (CRSF) attacks. This article provides code samples from Apache MyFaces 2.2.12 JSF Implementation, for a better understanding of the different scenarios.

II. Important JSF API Classes Associated With State Management

A pre-requisite to understanding State Management in JSF 2.2, is the knowledge of the different API classes that are utilized in State Management feature and how they work internally:

A. UIComponent

The base abstract class for all user interface components is javax.faces.component.UIComponent. This classdefines the state information and behavioural contracts for all components through a Java programming language API. UIComponentprovides the mechanism of saving and restoring individual component state.

A component from any of the component libraries or a custom component musthave UIComponent as one of its super classes (in the chain of super classes). JSF implementation provides UIComponentBase that extends from UIComponent,providingdefault implementations of various UIComponentmethods, and relieves application developers from the risk of gettingadversely affected due to an API change in UIComponent. Then there are other types of components i.e.,

- UIOutput extends from UIComponentBase, whichis used for displaying values during rendering of the component.
- UIInput extends from UIOutput,which is used for displaying valuesbut most importantly, during post back requests (likehtml <input> type tags)their values will be submitted. It should be noted that any component that submitsits values, should have UIInput as one of the super classes in its chain. Of course, one can create a totally new type of UIInputextending from UIComponentBase but that basically will lead to reinventing the wheel.

The UIComponent class implements the interface PartialStateHolder, which in turn extends StateHolder interface. This is how UIComponent is associated with state management interfaces to take part in either full or partial state saving and in restoring the state in postBack requests.

B. StateHolder Interface

Methods of this interface include:

```
void restoreState(FacesContext facesContext, Object state);
Object saveState(FacesContext facesContext);
void setTransient(boolean newTransientValue);
boolean isTransient();
```

C. PartialStateHolder Interface

Methods of this interface include:

```
void markInitialState();
boolean initialStateMarked();
void clearInitialState();
```

D. ViewHandler

ViewHandler is class through which JSF implementation abstracts the low-level details of a request and response, and essentially makes it independent of the servlet APIs. A ViewHandler manages the component-tree-creation and component-tree-rendering parts of a request lifecycle (i.e. "create view", "restore view" and "render response"). The main methods of this class that take active part in state management features are:

```
public abstract UIViewRoot restoreView(FacesContext facesContext, String viewId);
public abstract void writeState(FacesContext facesContext) throws IOException;
```

E. StateManager

StateManager is the main class that handles the management of state saving and state restore. It works hand-in-glove with the ViewHandler class. This class is usually invoked by a concrete implementation of ViewHandler. Note that the ViewHandler isolates JSF components from the details of the request format, while StateManager isolates JSF components from the details of the response format. Since request and response are usually tightly coupled, the StateManager and ViewHandler implementations are also tightly coupled (i.e. the ViewHandler and StateManager implementations come as pairs).

F. StateManagementStrategy (“sms”)

The mechanism of saving and restoring the Partial State is implemented differently in Mojarra and MyFaces. Mojarra uses tree visiting with a pluggable mechanism, while MyFaces uses a non-pluggable “facets + children” traversal. To overcome this issue, JSF 2.2 makes it mandatory to use StateManagementStrategy (involving a pluggable mechanism), while the tree visiting mechanism is fully capable of saving the partial state. The methods of this class are:

```
public abstract UIViewRoot restoreView(FacesContext facesContext, String viewId, String renderKitId);
public abstract Object saveView(FacesContext facesContext);
```

StateManager uses StateManagementStrategy, to delegatethe execution of saveView and restoreView. Therefore, if StateManagementStrategy is available, it takes over theresponsibility of saving view and restoring view.

The mechanism to access **sms** is through ViewDeclarationLangauge. It should be noted that, currently **sms** is only available for Facelets and not for JSPs. State Managementfor JSPs will fall back on the implementation of StateManager.

```
ViewDeclarationLanguage vdl = facesContext.getApplication().getViewHandler().getViewDeclarationLanguage(facesContext,viewId);
StateManagementStrategy sms = vdl.getStateManagementStrategy(facesContext, viewId);
```

G. ViewDeclarationLanguage

JSF implementation provides for multiple view types (XHTML as Facelets, JSPs, etc) and therefore, ViewHandler, for the most part, delegates to the implementation of the ViewDeclarationLanguage (aka VDL). Some of the main methods of VDL are:

```
public abstract UIViewRoot createView(FacesContext context, String viewId);
public abstract void buildView(FacesContext context, UIViewRoot view) throws IOException;
public abstract void renderView(FacesContext context, UIViewRoot view) throws IOException;
public abstract StateManagementStrategy getStateManagementStrategy(FacesContext context, String viewId);
public abstract UIViewRoot restoreView(FacesContext context, String viewId);
```

H. ResponseStateManager

ResponseStateManager is the helper class to StateManager, that knows the specific rendering technology being used to generate the response. It is accessed through RenderKit.

III. STATE MANAGEMENT LIFECYCLE

This section describes how the different API classes work together to execute the JSF Lifecycle, from the perspective ofState Management.

A. Initial Request

Once the request reaches the Render Response phase of the JSF Lifecycle, the view is built. What that means is that the XML markup is converted into Facelet (or taken from the facelet cache) and all tag handlers in the chainare applied, creating a full-fledged component tree stored inside UIViewRoot.

buildView: This process of building the view always occurs, either in Render Responsephase (for Initial Request) or in Restore View phase (for subsequent postBack requests). At this point, all the components along with UIViewRoot are at their initial state.

This happens through:

```
viewHandler.buildView(facesContext, uiViewRoot);
```

Of course, as described earlier,ViewHandlerdelegates to the appropriate VDL (ViewDeclarationLanguage) for buildView.

```
vdl.buildView(facesContext, uiViewRoot);
```

At this point UIViewRootis traversed in a recursive manner through all non-transient components and component.markInitialState()method is executed.

It should be noted that the markInitialState method is from PartialStateHolder. This method simply sets a boolean instance variable (in UIComponent) to true for future checks on the initial state. Both Mojarra and MyFaces provide this implementation in the base class UIComponent.

Once the buildView method completes the recursive execution, all non-transient components are marked with their initial state. This essentially means that any changes to components in terms of their attributes (which are part of the State Management, and any new dynamic components being added) are marked as partial or delta changes in Partial State Saving.

renderView: As part of this phase, after buildView is completed, execution will move to rendering of the view via the following code

```
viewHandler.renderView(facesContext, uiViewRoot);
```

Again, ViewHandlerdelegates to the appropriate VDL (ViewDeclarationLanguage) for renderView.

```
vdl.renderView(facesContext, uiViewRoot);
```

During the rendering of view, viewRoot.encodeAll(facesContext) is executed, which internally ends up calling all the components in the tree with their encodeAll. Also,encodeAllcalls encodeBegin(facesContext), encodeChildren(facesContext) or children.encodeAll(facesContext), for all the children and then encodeEnd(facesContext).

HTML form encoding: After encodeBeginis executed and all children have been encoded, a hidden input is added in the form with the name clientId of the form component and with suffix _submit with a value of 1. This input is used as part of the decode of the form component, to determine whether the form was submitted or not (during Apply Request Values phase) in postBack request).

```
<input type="hidden" name="j_id_9_SUBMIT" value="1" />
```

Since State Management is associated with the form, it is important to know which form was submitted.

State Writing: Prior to encodeEndcompletes for the form component, State Saving Methodmust be executed (server or client). This is done by executing viewHandler.writeState(facesContext), which delegates the call to StateManager,which in turn delegates to ResponseStateManagerto write the state. In an HTML type response in servlet environment, this is accomplished through:

```
RenderKit renderKit = facesContext.getRenderKit();
ResponseStateManager responseStateManager = renderKit.getResponseStateManager();
responseStateManager.writeState(facesContext, state);
```

Based upon the State Saving Method(client or server) and partial or full state saving, a state encoded token is written as an input hidden typewith the name javax.faces.ViewState. For e.g., a server state saving will create a response like the following:

```
<input type="hidden" name="javax.faces.ViewState" id="j_id__v_0:javax.faces.ViewState:1"
value="r7HbNKeUvF+sKfI6aHVqMBUD3UfWAfGwCzXUds3N0twkYZrv" autocomplete="off" />
```

In a client mode of state saving, the above value attribute has a much longer data value because it containsnot just the encoded token, but also a full/partial state encoded value. If the viewwastransient, the value of the above field will bestateless. Astateless view can be defined by having the f:view tag in the XHTML with the attribute transient set to true.

The above hidden input does not have the id value as javax.faces.ViewState because in multi HTML forms, id will beduplicated and that will be wrong, since one cannot have a html document with duplicate id, so it’s the name attribute that has the value javax.faces.ViewState.

State Saving in the Session: WhenState Saving Method is set to server, and the encoding of the UIViewRootis complete, an important process after that is the actual saving of the partial or full state in the session. This saved state is used in postBack request,and mapped with the token that was sent in the response. Executing saveView on the StateManagerdelegates to StateManagementStrategy.

```
StateManager stateManager = facesContext.getApplication().getStateManager();
stateManager.saveView(facesContext);
```

In a nutshell, the above saveView translates to:

```
    ViewDeclarationLanguage vdl = facesContext.getApplication().getViewHandler().getViewDeclarationLanguage(facesContext, uiViewRoot.getViewId());

    StateManagementStrategy sms = vdl.getStateManagementStrategy(facesContext, viewId);

    Object savedState = sms.saveView(facesContext);
    //Internal Implementation - save the above serializedView in the //session in a map type implementation, implemented as LIFO //(Last In First Out)
```

JSF provides a feature to save a certain number of views (for server state), so that an application accessed in multiple browser tabs/windows can workforthe same session. MyFacesweb context config parameter that manages this count is org.apache.myfaces.Number_Of_Views_In_Session and is set to a default value of 20. As a result, if 21tabsare opened for the same session, with different non-transient views, a ViewExpiredExceptionwill be encountered when the user returns to the first tab and attempts a postBack requestbecausethe first saved view state would have been removed while saving the state for the 21st view.

Saved State: During saving of the view (ObjectsavedState = sms.saveView(facesContext)),UIViewRoot component tree is visited to reach each component,and component'ssaveState methodis executed.The stateis put in a map keyed by the component's clientId. To ensure that this tree visit does not involve the virtual components inside UIData (e.g. ui:repeat, h:datatable etc.), a hint is set while visiting the components:

```
    facesContext.getAttributes().put(SKIP_ITERATION_HINT, Boolean.TRUE);
```

A code sample (showing only the state saving) from MyFacesis included below for a better understanding:

```
facesContext.getAttributes().put(SKIP_ITERATION_HINT, Boolean.TRUE);

try{

uiViewRoot.visitTree( getVisitContextFactory().getVisitContext( facesContext, null, VISIT_HINTS), new VisitCallback(){
            public VisitResult visit(VisitContext context, UIComponent target){
                FacesContext facesContext = context.getFacesContext();
                Object state;
                if ((target == null) || target.isTransient()){
                    // No need to bother with these components or //their children.
                    return VisitResult.REJECT;
                }
                if (target.getParent() != null){
                    state = target.saveState (facesContext);
                    if (state != null){
                        // Save by client ID into our map.
                        states.put (target.getClientId (facesContext), state);
                    }
                        return VisitResult.ACCEPT;
                }else{
                    //Only UIViewRoot has no parent in a //component tree.
                    return VisitResult.ACCEPT;
                }
            }
        });
    }
    finally{

facesContext.getAttributes().remove(SKIP_ITERATION_HINT);
    }
    if (!uiViewRoot.isTransient()){
        Object state = uiViewRoot.saveState (facesContext);
        if (state != null){
            states.put (uiViewRoot.getClientId (facesContext), state);
        }
    }
```

The returned map is the actual state (full or partial) that is then encoded (and can be serialized,encrypted too) and saved in the session.

Saved View Scope: As part of saving the state, the actual data/bean model that provide values through EL expressions, is not saved as part of component state, but the references are saved as part of their scope. For e.g.,

- RequestScope–is destroyed once request is completed
- ViewScoped (CDI) – This introduced as a part of the Context Dependency Injection (CDI) in JEE7 and differs from managed bean ViewScoped, is put in a scope such that the bean instance is available until and unless the view has navigated to a different view. Therefore, if CDI is available, the bean annotated with this CDI scope is saved in a JSF implementation session scoped CDI bean.If not, it is saved directly in the session where the state is saved.

B. PostBack Request

The best way to determine a request is a postBack request (for e.g., HTML form submission), is to check facesContext.isPostBack(). The first step of the postBack request in an HTML & servlet setting is to evaluate the request parameter javax.faces.ViewState.

```
facesContext.getExternalContext().getRequestParameterMap().containsKey("javax.faces.ViewState");
```

Restore State: The first phase of the JSF Lifecyle i.e.,Restore Viewis invoked for restoring the state. Of course, before a view can be restored, it must be built first (likebuildView, component marking to the Initial State, as described in the Initial Request). At this point, the UIViewRootis the same, as was built in Initial Request. Restoring View happens through the following:

```
UIViewRoot viewRoot = viewHandler.restoreView(facesContext, viewId);
UIViewRoot viewRoot = vdl.restoreView(context, viewId);
```

This internally delegates the restore to the StateManager and StateManagementStratgey.

```
Application application = facesContext.getApplication();
ViewHandler applicationViewHandler =
application.getViewHandler();
String renderKitId =
applicationViewHandler.calculateRenderKitId(facesContext);
UIViewRoot viewRoot =
application.getStateManager().restoreView(facesContext,
viewId, renderKitId);
```

StateManager further delegates the task of restoring the view to StateManagementStrategy.

```
StateManagementStrategy sms =
vdl.getStateManagementStrategy(facesContext, viewId);
UIViewRoot viewRoot = sms.restoreView(facesContext, viewId,
renderKitId);
```

Restore state involves the recursive execution through all components and children, as shown below:

```
component.restoreState(context, componentState);
```

Since initial state is built using Facelet, using partial state saving (PSS) is almost always a better approach than full state save and restore. The component state object is either retrieved from the session (for server state) or decoded from the javax.faces.ViewState request parameter.

StateManagementStrategy will go through the ResponseStateManager for restoring the state.

```
ResponseStateManager manager =
getRenderKitFactory().getRenderKit(context,
renderKitId).getResponseStateManager();
Object[] state = (Object[]) manager.getState(context, viewId);
```

The above method gets the request parameter value javax.faces.ViewState and then decodes it (internally the decoding involves cipher and other mechanisms, to make sure the request is not a hacked one). Based on this token, JSF implementation decodes the actual state from the session or from the request parameter value. MyFacesimplementation returns the state as an array of objects.

The State is essentially a map of clientId with the associated state of the component; states is the Object[1] from the above "state".

```
Object viewRootState =
states.get(viewRoot.getClientId(context));
```

Based on the clientId, the state for a particular component is retrieved and passed to the component's restoreState method to restore it.

Component Types (with Attached Objects): The main types of UIComponent's are UIOutput and UIInput. The following section explains the chain of component types, how they extend from each other, andthe type of objects state managed in these types of components, by default.

- UIComponentBase extends UIComponent:
    - Manages the state for ComponentSystemEventListener, one of the methods used to subscribe to the event (theevents are managed in such a way that when a component state is saved, the event listeners are also saved)

```
public void subscribeToEvent(Class<? extends SystemEvent>
eventClass, ComponentSystemEventListener componentListener)
```

- Manages the state for ClientBehavior, which can be added to component through:

```
public void addClientBehavior(String eventName,
ClientBehavior behavior)
```

- Manages the state for FacesListener, which can be added to component through

```
protected void addFacesListener(FacesListener listener)
```

UIOutput extends UIComponentBase: Manages the state for Converter, which can be added through

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

UIInput extends UIOutput: Manages the states for all the Validators associated with the component; a Validatorcan be added to the component through

```
public void addValidator(Validator validator)
```

StateHelper: The JSF implementation supports State Management of any UIComponent through the following mechanism

- UIComponent's implementation is Serializable
- UIComponent's properties are managed with StateHelper
- UIComponent provides its own implementation of saveState and restoreState

The way to get hold of the StateHelper is through component.getStateHelper(). StateHelper also extends from StateHolder. Along with that,StateHelper provides following methods:

```
void add(Serializable key, Object value);
Object eval(Serializable key);0
Object eval(Serializable key, Object defaultValue);
Object get(Serializable key);
Object put(Serializable key, Object value);
Object put(Serializable key, String mapKey, Object value);
Object remove(Serializable key);
Objectremove(Serializable key, Object valueOrKey);
```

The actual implementation of StateHelper is managedby default, in the class UIComponent (in both Myfaces and Mojarra implementation of JSF). For e.g.,the following sample code outlines how the styleproperty is managed with the help of StateHelper:

```
public String getStyle(){
  return (String) getStateHelper().eval(PropertyKeys.style);
}
public void setStyle(String style){
  getStateHelper().put(PropertyKeys.style, style );
}
```

The implementation of StateHelper manages both the full and partial (delta) state, as well as UIComponentBase'ssaveState and restoreState methods.

```
public Object put(Serializable key, Object value){
            Object returnValue = null;
            if (_component.initialStateMarked()){
                  if (_deltas.containsKey(key)){
                        returnValue = _deltas.put(key,
value);
                        _fullState.put(key, value);
                  }else if (value == null
&& !_fullState.containsKey(key)){
                        returnValue = null;
                  }else{
                        _deltas.put(key, value);
                        returnValue =
_fullState.put(key, value);
                  }
            }else{
                  returnValue = _fullState.put(key, value);
            }
            return returnValue;
}
```

If the above put method is executed when initailStateMarked has already been marked true for the component, the value is also assigned to the _deltas. The primary difference between StateHelper'sget and eval methods is that eval looks for a ValueExpression that might be present for the key, and then evaluatesit, if there were no literal values set.

```
public Object eval(Serializable key, Object defaultValue){
          Object returnValue = _fullState.get(key);
          if (returnValue != null){
                  return returnValue;
          }
          ValueExpression expression =
_component.getValueExpression(key.toString());
          if (expression != null){
                        return
expression.getValue(_component.getFacesContext().getELContex
t());
          }
          return defaultValue;
}
```

To understand the eval method better, refer below to the creation of an input text tag in XHTML using:

```
<h:inputText value="#{testController.firstName}" />
```

During the postBack request, in process validation phase, the value that was submitted for firstNameis converted and then set through setValue method. Ifthe tag has a ValueExpression for the value attribute, in the validate phase a literal value is set against the value attribute. When StateHelper'seval method is executed for any subsequent invocation of getValue, it returns the literal value and fetches the value from the ValueExpression only after it has been reset to null.

IV. Conclusion

This paper presents a thorough deep dive understanding of how state management along with state saving strategy works during JSF Lifecyle and how state is saved duringInitial Requestand state restore during postBack requests. It also discusses the concept of how ViewScoped beans are saved in session for stateful views. It helps to clear any misunderstanding that UIComponents instances themselves are not saved, but it's their attributes that are saved and then reused during restore of the state. Proper understanding of state management, partial and full state saving can help architects to better design complex enterprise system based on JSF 2.2.

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/