Orientation in Objects

JSF Best Practices

Scope management for clean sessions

) Schulung)

) Beratung)

) Entwicklung)

) Artikel)

AUTHOR

Thomas Asel
Orientation in Objects GmbH

Published: 11.3.2011

ABSTRACT

When starting with JSF one of the common pitfalls is how to pass values or parameters efficiently. In many cases developers end up putting Managed Beans in Session Scope to share Bean attributes though more appropriate solutions are available. This article lists some routine tasks for JSF developers and gives tips and tricks on how to pass values efficiently without having to pollute the session-object. The intended audience for this article are novice JSF developers as well as developers that are interested in alternative solutions to common problems.

Orientation in Objects GmbH

Weinheimer Str. 68
D-68309 Mannheim

Tel. +49 (0) 6 21 - 7 18 39 - 0
Fax +49 (0) 6 21 - 7 18 39 - 50

www.oio.de info@oio.de

SESSION POLLUTION AND WHY IT IS HAZARDOUS

Admittedly, putting all your Managed Beans in Session Scope may be a working solution for some problems encountered while developing web applications with JSF. But it bears a high potential for undesired behavior, which is mostly not noticed until real problems arise. Problems caused by wrong Bean Scopes are often hard to trace and require profound knowledge of the JSF framework (and the fundamental principles of web applications in general). This is why particularly developers new to JSF get easily in trouble by excessively using Session Scoped Beans.

There are a few simple questions to consider when dealing with Session Scope:

1. **Is the instantiated object required available throughout the entire user session?**

   The session spans over all requests and may involve traversing over your whole application.

2. **Is your code thread-safe?**

   The Session object is shared among all threads initiated by the same user. This may cause problems when objects are accessed concurrently e.g. when the user opens a second browser tab of the application. Concurrency problems are hard to discover, particullarly in the presentation tier where unit testing may be complicated, due to technical difficulties or, in some cases, simply is not possible.

3. **How does a larger session object affect your systems performance?**

   This question can often only be answered by observing the system under load. Unfortunately, performance problems are often perceived very late: When the software is already deployed and performs poor in the production environment. At least, keep in mind that heavier session objects require more physical memory.

4. **How does session-wide accessibility of Managed Beans affect your software architecture?**

   Consider architectural principles like loose coupling, information hiding, etc. Is your intended architecture breached by exposing a certain object throughout a session? Even when you are frequently evaluating your implementation against your intended architecture (which is highly recommended to do!), scopes are a rather technical aspect. This stipulates for your analysis tools to understand these technical aspects by evaluating proprietary annotations, metadata and configuration artifacts.

With these considerations about session pollution and the resulting risks let's take a look at common scenarios in JSF development. The following sections discuss three scenarios commonly encountered in almost every web application.

PASSING INFORMATION TO HANDLERS

Consider a simple view related to a single Managed Bean. Further, think about a Command Button triggering an action method. How can the action method know about the command button it was triggered from? And even more interesting: How can the context in which the button was pressed also be passed on to the method? A simple example:

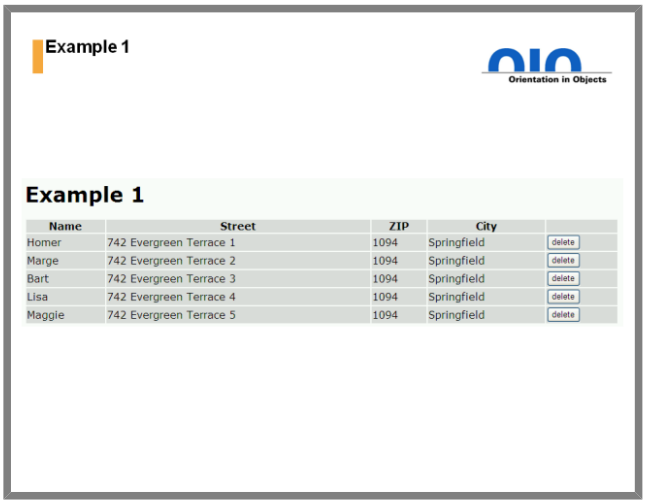

Figure 1: Example 1

Regardless which one of the h:commandButtons is pressed, they are all triggering the same action method. The method contains some logic to process the respective set of data (displayed in the tables rows). But how exactly does the method know which set of data it is to process? Lets take a look at the view definition for this example:

```
...
<h:form>
 <h1><h:outputText value="Example 1"/></h1>
 <h:dataTable value="#{adressTableBeanExample1.addresses}"
  var="address" >
 <h:column>
  <f:facet name="header" >
     <h:outputText value="Name"/>
  </f:facet>
  <h:outputText value="#{address.name}" />
 </h:column>
 <h:column>
  <f:facet name="header" >
     <h:outputText value="Street"/>
  </f:facet>
  <h:outputText value="#{address.street}" />
 </h:column>
 <h:column>
  <f:facet name="header" >
     <h:outputText value="ZIP"/>
  </f:facet>
  <h:outputText value="#{address.zipCode}" />
 </h:column>
 <h:column>
  <f:facet name="header" >
     <h:outputText value="City"/>
  </f:facet>
  <h:outputText value="#{address.city}" />
 </h:column>
 <h:column>
  <h:commandButton value="delete"
   action="#{adressTableBeanExample1.delete}"/>
 </h:column>
</h:dataTable>
</h:form>
...
```

Example 1: View Definition for Example 1

This is a standard task and encountered in almost every web-application. The following sections discuss some approaches to solve the problem described above.

SETTING A PROPERTY ON THE MANAGED BEAN

An easy solution is to simply fill a certain property in the managed bean. Your Managed Bean may contain a property named "selected" to hold the set of data that was selected by the user when clicking the h:commandButton. Since JSF 1.2 a tag is offered to conveniently set a property in a corresponding Managed Bean called f:setPropertyActionListener. Don't get confused by the bulky tag name as it describes the tags purpose pretty well. When using the f:setPropertyActionListener within a component, an Action Listener is created by the framework. This Action Listener fills the targeted value of the Managed Bean with a given value. The above example using an f:setPropertyActionListener looks like this:

```
...
<h:column>
 <h:commandButton value="delete"
  action="#{adressTableBeanExample1.delete}" >
  <f:setPropertyActionListener
   target="#{adressTableBeanExample1.selected}"
   value="#{address}" />
 </h:commandButton>
</h:column>
...
```

Example 2: View Definition for Example 1 using f:setPropertyActionListener

With the implicitly created Action Listeners filling the property in the Managed Bean, all you need to do to provide availability of the selected data in the Action Method is to simply access the beans "selected" property:

```
...
public String delete(){
 addresses.remove(selected);
 return "";
}
...
```

Example 3: Action Method in Backing Bean for Example 1

Although very simple and handy this solution has some drawbacks:

- Selected values need to be all of the same type
- With lots of different selections on the same page, your Managed Bean may grow quickly, resulting in reduced cohesion and thus, more important, poor maintainability.

However, unless your view isn't very complex and therefore a single selection is all you need, this may be the method of choice for you.

ADDING PARAMETERS TO THE COMMAND COMPONENT

The following approach eliminates the drawbacks of the prior one, by introducing parameters to the Command Component. While f:setPropertyActionListener requires to be placed (at least somewhere) within a component derived from ActionSource, every component derived from UIComponent (which is true for every component represented by a tag in views) is capable of carrying parameters. By adding the selected dataset as parameter to the h:commandButton, this parameter is available whenever you are dealing with this component. Compared to Action Methods, Action Listeners are aware of the ActionEvent that triggered the action. Action Events refer to the component triggering the event, in this case the h:commandButton. Having Access to the component, all parameters are accessible, so all you need to do is to call the components getChildren() method to access the nested components – including the nested parameters.

```
...
<h:column>
 <h:commandButton value="delete"
  actionListener="#{addressTableBeanExample2.delete}">
  <f:param name="selected" value="#{address}" />
 </h:commandButton>
</h:column>
...
```

Example 4: View Definition for Example 2 using f:param

```
...
public void delete(ActionEvent event){
 for(UIComponent component :
 event.getComponent().getChildren()){
  if( component instanceof UIParameter ){
   UIParameter param = (UIParameter) component;
   if(param.getName().equals("selected")){
    addresses.remove(param.getValue());
   }
  }
 }
}
...
```

Example 5: ActionListener in Backing Bean for Example 2

As you can see, the Action Method from the previous example has changed to an Action Listener. Please note that since the h:commandButton may contain multiple parameters, the Listener is responsible of checking for the parameter name to avoid evaluation of wrong parameters.

The major drawback of this solution is that it is code intensive. You have to access the component and need to write some logic to traverse its children. Keep in mind that components may also have further child components besides f:param, therefore you also need to check for the correct type of the nested child components. When dealing with multiple parameters you also need to distinguish them by name, which requires additional code. Since Action Listeners are used quite frequently, this approach can often be found, especially in Ajax-intensive applications.

ADDING ATTRIBUTES TO THE COMMAND COMPONENT

A more convenient approach, due to reduction of required code, is to add the desired dataset as an attribute to the component instead of nesting it as a child parameter component. When adding objects as attributes to components, these objects are available through the components attribute map. The following code shows the above example using f:attribute instead of f:param

```
...
<h:column>
 <h:commandButton value="delete"
  actionListener="#{addressTableBeanExample3.delete}" >
  <f:attribute name="selected" value="#{address}" />
 </h:commandButton>
</h:column>
...
```

Example 6: View Definition for Example 3 using f:attribute

```
...
 public void delete(ActionEvent event){
  Address selected = (Address) event.getComponent().
   getAttributes().get("selected");
  addresses.remove(selected);
 }
...
```

Example 7: Action Listener in Backing Bean for Example 3

The main differences between usage of f:attribute and f:param is that parameters are added to the closest UIComponent associated with a custom action, requiring f:param to be placed as child component somewhere underneath a component implementing the ActionSource interface. While this is valid only for CommandComponents, attributes may be added to every kind of UI Component, e.g. input components like input text fields, implementing the ValueHolder interface instead of ActionSource.

Compared to the previous example, the Action Listener became pretty lean this time. The Listener in this example simply assumes that the parameter named "selected" is an instance of the Address class. Please note that this may lead to errors during runtime when changing the attributes value in the view definition to another EL expression referencing a different type . Until today IDE's lack on type checking mechanisms for attributes defined in JSF views.

PASSING VALUES TO CONVERTERS AND VALIDATORS

The benefit of Validators or Converters is often increased by enabling them to deal with parameters. It is worth thinking about a dedicated Validator class to keep your Managed Beans slim and maintainable whenever a Validator contains a lot of logic. But even if you decide to keep validation in the Managed Bean, passing parameters as described in the previous two sections is an option to keep your code a bit smarter by simply attaching an f:attribute to the component. You can access the attribute within your validation method via the component since it is passed as an argument to the validation method. Validation methods give developers access to all the information needed, as their signature is like

```
public void validate(FacesContext ctx,
 UIComponent component,
 Object toValidate)
```

The same principle can be applied to Converters since the component is passed as argument in both methods, `getAsString(…)` and `getAsObject(…)`.

PASSING VALUES IN TABLES

So far the concepts for passing values to handlers were applied within h:dataTables. Passing values to handlers is easily achieved by placing a command component in a table column. Clicks on the command component may trigger an Action Method or Action Listener, the information about which data row to process may be passed as attribute or parameter. The following code shows an example with an ActionListener using an Attribute to describe the selected table row.

```
...
<h:dataTable value="#{addressTableBeanExample4.data}"
var="data">
 <h:column id="firstname">
  <f:facet name="header">
   <h:outputText value="Firstname"/>
  </f:facet>
  <h:outputText value="#{data.firstname}" />
 </h:column>
 <h:column id="lastname">
  <f:facet name="header">
   <h:outputText value="Lastname" />
  </f:facet>
  <h:outputText value="#{data.lastname}" />
 </h:column>
 <h:column id="customerId">
  <f:facet name="header">
   <h:outputText value="Customer ID" />
  </f:facet>
  <h:outputText value="#{data.customerId}" />
 </h:column>
 <h:column id="action">
  <h:commandButton value="Select" actionListener=
   "#{addressTableBeanExample4.selectionListener}">
  <f:attribute name="selection" value="#{data}"/>
  </h:commandButton>
 </h:column>
</h:dataTable>
...
```

Example 8: View Definition for Example 4 using an ActionListener with f:attribute

```
...
@ManagedBean(name="addressTableBeanExample4")
@ViewScoped
public class ExampleBean4 implements Serializable{
 private static final long serialVersionUID = 1L;
 private transient List<Customer> data = new
 ArrayList<Customer>() ;
 private Customer selected;
 public ExampleBean4(){
  // create example data model
  data.add(new Customer("Homer","Simpson",80085));
  data.add(new Customer("Barney","Gumble",83321));
  data.add(new Customer("Ned","Flanders",81813));
 }
 public void selectionListener(ActionEvent event){
  Customer customer = (Customer) event.getComponent().
   getAttributes().get("selection");
  this.selected = customer;
 }
 public Customer getSelected() {
  return selected;
 }
 public void setSelected(Customer selected) {
  this.selected = selected;
 }
 public List<Customer> getData() {
  return data;
 }
 public void setData(List<Customer> data) {
  this.data = data;
 }
}
```

Example 9: Backing Bean Definition for Example 4

While the previous example requires explicit definition of an f:actionListener, JSF offers a more data centric approach using a distinct data model for DataTables. The preceding example used a value binding to a collection containing the data to display. Using a reference to a DataModel instance instead of a collection offers a more convenient way to access the selected data set:

```
...
<h:dataTable value="#{addressTableBeanExample5.data}"
var="data">
 <h:column id="firstname">
  <f:facet name="header">
   <h:outputText value="Firstname"/>
  </f:facet>
  <h:outputText value="#{data.firstname}" />
 </h:column>
 <h:column id="lastname">
  <f:facet name="header">
   <h:outputText value="Lastname" />
  </f:facet>
  <h:outputText value="#{data.lastname}" />
 </h:column>
 <h:column id="customerId">
  <f:facet name="header">
   <h:outputText value="Customer ID" />
  </f:facet>
  <h:outputText value="#{data.customerId}" />
 </h:column>
 <h:column id="action">
  <h:commandButton value="Select"
   action="#{addressTableBeanExample5.select}"/>
 </h:column>
</h:dataTable>
...
```

Example 10: View Definition for Example 5

```
...
@ManagedBean(name="addressTableBeanExample5")
@ViewScoped
public class ExampleBean5 implements Serializable{
 private static final long serialVersionUID = 1L;
 private transient ListDataModel<Customer> data =
  new ListDataModel<Customer>() ;
 private Customer selected;
 public ExampleBean5(){
  // create example data model
  List<Customer> customers = new ArrayList<Customer>();
  customers.add(new Customer("Homer","Simpson",80085));
  customers.add(new Customer("Barney","Gumble",83321));
  customers.add(new Customer("Ned","Flanders",81813));
  this.data.setWrappedData(customers);
 }
 public Customer getSelected() {
  return selected;
 }
 public void setSelected(Customer selected) {
  this.selected = selected;
 }
 public ListDataModel<Customer> getData() {
  return data;
 }
 public void setData(ListDataModel<Customer> data) {
  this.data = data;
 }
 public String select(){
  this.selected = data.getRowData();
  return "";
 }
}
...
```

Example 11: Backing Bean using a DataModel in Example 5

As you can see in the code example above, JSF takes care of informing the data model which distinct data set was selected . When an ActionSource is triggered, JSF takes notice about the related element of the wrapped data and updates the table model. Accessing the selected data set is made easy using the getRowData() Method of the TableModel. Please note the difference between the ActionListener in Example 4 and the ActionMethod in this example.

SHARING INFORMATION BETWEEN VIEWS

USING F:SETPROPERTYACTIONLISTENER

The usage of the f:setPropertyActionListener tag has been discussed earlier. The example in section one described how the tag can be used to set a managed bean property. The same approach can be used to set a property on a managed bean that belongs to the next view to display. Simply add the f:setPropertyActionListener tag as child component to the command component triggering an action method (or using JSF 2's implicit navigation feature) and associate the target attribute to a property on the next views managed bean:

```
...
<h:form id="form">
 ...
 <h:inputText value="#{exampleBean6a.input}" />
 <h:commandButton value="submit" action="example6b.xhtml">
  <f:setPropertyActionListener value="#{exampleBean6a.input}"
   target="#{exampleBean6b.value}"/>
 </h:commandButton>
</h:form>
...
```

Example 12: View Defintion of first view in Example 6

```
...
<h:form id="form">
 ...
 <h:outputText value="Passed Value: " />
 <h:outputText value="#{exampleBean6b.value}" />
</h:form>
...
```

Example 13: View Defintion of 2nd view in Example 6

The Backing Beans of both views are nothing more but simple POJO's containing accessor methods for a single attribute (and of course, both could easily be implemented as instances of the same class):

```
...
@ManagedBean
@RequestScoped
public class ExampleBean6a implements Serializable{
 private static final long serialVersionUID = 1L;
 private String input;
 public String getInput() {
  return input;
 }
 public void setInput(String input) {
  this.input = input;
 }
}
@ManagedBean
@RequestScoped
public class ExampleBean6b implements Serializable{
 private static final long serialVersionUID = 1L;
 public ExampleBean6b(){
 }
 private String value;
 public String getValue() {
  return value;
 }
 public void setValue(String value) {
  this.value = value;
 }
}
...
```

Example 14: Backing Beans of Example 6

The question here is: Why is this working? A good question, since the instance of the managed bean for the following view is not expected to be created before the "Render Response" Phase of the lifecycle (unless the managed beans scope is anything else then request). When f:setPropertyActionListener is used, instantiation happens by the time of executing the implicitly generated Action Listener, that is before the "Invoke Application" phase of the lifecycle. Thus, the Managed Bean for the following view is already available when reaching the "Render Response" phase.

ACTION LISTENER

The usage of f:setPropertyActionListener is a convenient way to store values in Managed Beans of subsequent views. However, though more code intensive, the same effect can be achieved by manually designed ActionListeners. This approach gives you the ability to process values before storing them in another Managed Bean. However, this may also tempt to put logic in the Presentation Layer that belongs elsewhere. Keep in mind, that JSF offers concepts for conversation and validation whenever you think about data transformation when passing values to managed beans using an Action Listener.

FLASHSCOPE

With JSF2 a new feature called "Flash" was introduced. It is inspired by the Ruby on Rails Framework and offers a convenient way to pass information between views. Though often mistakenly referred to as "Flash Scope", the Flash is not a scope like the request or session scope. It is rather a map managed by the framework. It is capable of holding a value until the next view is processed, so you would not want to put an entire managed bean in the "Flash Scope".

The following example shows how the Flash can be used to pass an input from one view to another. There is a request scoped backing bean for the second view only. Note how the Flash is injected in the backing bean using the @ManagedProperty annotation. You could also access the Flash programmatically by calling `FacesContext.getCurrentInstance().getExternalContext().getFlash()` but having the Flash injected into the bean is more convenient.

```
...
<h:form id="form">
 ...
 <h:outputText value="Enter a value into the Flash"/>
 <h:inputText value="#{flash.inputText}" />
 <h:commandButton value="submit" action="example7b.xhtml"
 />
</h:form>
...
```

Example 15: View Definition for first view in Example 7

```
...
<h:form id="form">
 ...
 <h:outputText value="Value From Flash:"/>
 <h:outputText value="#{flashExampleBean.inputFromFlash}"
 />
 <h:commandButton value="back" action="example7a.xhtml" />
</h:form>
...
```

Example 16: View Definition for 2nd view in Example 7

```
...
@ManagedBean
@RequestScoped
public class FlashExampleBean implements Serializable{
 @ManagedProperty("#{flash}")
 private Flash flash;
 public String getInputFromFlash(){
  String inputText = (String) flash.get("inputText");
  flash.keep("inputText");
  return inputText;
 }
 public void setFlash(Flash flash) {
  this.flash = flash;
 }
 public Flash getFlash() {
  return flash;
 }
}
...
```

Example 17: Backing Bean Definition for 2nd view in Example 7

You may have noticed the call to flash.keep() in the backing beans getter method. This tells the flash to keep the value for another subsequent request as values stored in the Flash during request N are only available throughout the request N+1 unless the Flash is told to keep it for another request. By calling Flash.keep() we asure that the input is still available when returning to the first page by pressing the back button.

WHEN TO USE SESSION-SCOPE

The previous sections discussed several methods to pass values to handlers and between views without bloating the session. Finally we should take a look at scenarios where session scope should be used. As mentioned in the beginning of this article, objects stored in session scope remain until the end of the user session or until they are removed programmatically. A common scenario where an object is used throughout the entire lifetime of the user session is authentication. Consider a login screen where the user has to enter its credentials. If authentication is successful the user session is associated with an object representing the authenticated user. It may contain the users name, its customer id etc. This object may be used throughout the entire session to decide for example the look and feel of the application, the options given to the user and the general behavior of the application for this particular user.

It is totally vital to have a session scoped managed bean in your JSF application that stores information needed throughout the user session, but it is good practice to have only one session bean. If you feel you need more, think about the basic design of your application.

CONCLUSION

To review the usage of beans with session scope is one of the easy winners for every reviewer of most real world JSF applications. The main goal of this article is to discuss common scenarios of improperly session scoped beans and to give advice on how to prevent this. Of course all this does not mean that session scope is a bad thing. The usage of session scoped beans may be totally valid. However, developers need to be sure that their intended solution is not only working for a single use case but also free from side effects. This article hopefully helps to shed some light on the side effects of improper usage of session scopes. And where not – if you remind one thing: Always have some reasons for using state in a session.

The complete source code of the examples is available from [1]. Plese read the license file license.txt

The sources are contained in a ZIP archive as Eclipse project. The sources were developed on Mojarra 2.0.3, though any JSF 2.0 compliant implementation should be fine. Be sure to include the JSF libraries jsf-api.jar and jsf-impl.jar in WebContent/WEB-INF/lib. The libraries can be obtained from ORACLE's Mojarra project [2] or MyFaces [3], the open source implementation of JSF.

REFERENCES:

- [1] Source code of the examples
  JSF-Best-Practices.zip (/public/java/jsf/JSF-Best-Practices.zip)
- [2] JSF Implementation Mojarra
  http://javaserverfaces.java.net/ (http://javaserverfaces.java.net/)
- [3] JSF Implementation MyFaces
  http://myfaces.apache.org/ (http://myfaces.apache.org/)