SaveState

UISaveState

Description

Where possible, server-side objects used to process a JSF request should be "request-scoped". This allows memory on the server to be freed as soon as the request is complete. It also allows an application to be scaled up; distributing processing across multiple servers is simple and efficient when only request-scoped objects are used.

However there are also drawbacks to using request scope. In particular, when the user is performing an operation that is spread across multiple pages then some kind of state needs to be stored. The standard solution is to use "session" scope, ie to use the standard servlet HttpSession object to store state. This is an easy to use but not always satisfying approach. The main issues are:

- If the application does not explicitly remove something from the session, then it hangs around in memory until the user's session times out. But keeping track of data and explicitly removing it is a very tricky task.
- State can be kept too long; returning to a page visited some time ago may show stale data if the associated session-state data has not been discarded.

The Tomahawk t:saveState tag provides an alternative "view" scope. It allows objects to be associated with the current JSF view, so that they are kept alive while the user stays on the same view (page), but are automatically discarded when the user moves to another view. And it is possible to configure saveState tags so that objects associated with the previous view are "passed on" to the new view when a navigation occurs, so that objects can be kept alive across a specific set of pages.

JSF has always had a "view scope" functionality built in (see UIViewRoot.getAttributes). However there has been no convenient way to use this. JSF2.0 will have the ability to declare managed beans as having "view" scope; therefore when using JSF2.0 you might want to investigate the standard facilities first. But for JSF1.0 and JSF1.1 applications, t:saveState provides this now (via a page tag rather than a managed-bean declaration).

Note that the t:saveState tag does require that all the objects it stores are Serializable (or implement the JSF StateHolder interface).

When JSF is configured to use "client-side-state-saving", then objects in "view" scope are automatically sent to the browser along with the rest of the JSF view state. It is then possible to have a scalable distributed system without needing to configure session clustering. However of course the network bandwidth needed does increase as the view-scoped objects must be transferred back and forth on each request.

When JSF is configured to use "server-side-state-saving" then objects in "view" scope are automatically stored in the HttpSession along with the rest of the JSF view state.

Example (see "sample1.jsp" of the "examples" web application):

```
<t:saveState id="save1" value="#{calcForm.number1}" />
<t:saveState id="save2" value="#{calcForm.number2}" />
<t:saveState id="save3" value="#{ucaseForm.text}" />
```

The current values of the three properties number1, number2 and text are automatically saved within the client response and get restored at the next client request.

You can also save a whole bean. Example:

<t:saveState id="saveCalcForm" value="#{calcForm}"/>

The whole bean automatically is saved and restored. To be able to save and restore the value of a bean property or the bean itself, it must implement one of the following:

- the java.io.Serializable interface
- the javax.faces.component.StateHolder interface and a default constructor

Scoping and Traversing with t:saveState

General Description

By using t:saveState you can ease object passing and traversing between pages. Thus enabling scopes on the frontend side of things which are smaller than session and longer than request, without having to put an extra burden on the session object.

Usage:
you simple set a

```
<t:saveState id="save1" value="#{scopebean}" />
```

on page 1

then you do something which pushes you to page2

```
<t:saveState id="save1" value="#{scopebean}" />
```

on page 2 should restore the entered data from page1 at the beginning of the phase cycles. thus once you reference scopebean on page2 you will get the values from page1, before the value of the submit are applied to the bean (if at all).

You can do this for as many pages as you want. The scopebean is dropped from the cycle the moment the user runs into a page without a t:saveState component.

Thus get a session like behavior on the scopebean can be achieved, without having to put the affected bean into the session.

The usage example for such a mechanism is a wizard or some kind of dialog system. The mechanism is comparable to such systems, but easier to use and limited to the component level.

Alternate systems which provide similar functionality

- Orchestra http://myfaces.apache.org/orchestra
- struts Shale Framework dialog
- spring dialog

and probably a bunch of other systems.

Note: dialog systems go far beyound the t:saveState component, as such as they introduce another layer of pageflow into the system, while t:saveState can be seen as quickly to implement solution to the problem of having to keep data over multiple pages.

Advantages and disadvantages of t:saveState over session scope

Advantages: It is safer, because objects do not have to be explicitely dropped. And depending on the saveState mechanism, there is the possibility of not having to put extra burden on the session.

Disadvantages: Every object dropped into the t:saveState component has to be serializable, thus extra burden is put onto the server and/or the client, and the implementation effort is increased for complex objects.

Useful Links

Article about the scope problems and how to solve them with t:saveState