ConversationTag

Conversation Tag

The tomahawk sandbox component ConversationTag will introduce a new scope to JSF called: conversation scope
Other keywords for this scope are: dialog scope, pageflow scope, application transaction, unit of work.

The features of this tag are:

- Introduce a new conversation scope
- Multiple window aware (you can start the same conversation from multiple differen windows)
- Named (parallel) conversations
- Syncronize with lifetime of your database session
- Timeout to get rid of orphaned conversations
- Do not use serialization - thus it can be used with transparent persistence frameworks

The conversation tag do NOT require a persistence framework. Virtually every managed bean and its content can be put into the conversation scope. There are just helpers to synchronize the lifetime of your database sesssion with a conversation.

Glossary

- conversation

Same as dialog, application transaction, pageflow, etc.

Mainly this is a scope which spans mutliple requests and has a defined start and end point

- multiple window

Using a session bean often lead to the problem that you cant deal with multiple open windows. The conversation tag tries to avoid this by adding a parameter to the url named "conversationContext". You can have multiple named conversations within an conversationContext and multiple conversationContexts within an session.

Within an conversationContext you can forcibly start a new conversationContext.

I am not aware of a solution which is 100% save against the multiple window problem. For example: If the user clicks on an link which already has a conversationContext parameter there is NO WAY on the server to prevent this nor to know this.

However, using the conversation tag is a good try to avoid the session scope at all.

Setup

Before you can start using the conversation tag you have to add additional stuff to your web.xml configuration.

```
        <filter>
                <filter-name>conversationFilter</filter-name>
                <filter-class>org.apache.myfaces.custom.conversation.ConversationServletFilter</filter-class>
        </filter>

        <filter>
                <filter-name>requestParameterProvider</filter-name>
                <filter-class>org.apache.myfaces.custom.requestParameterProvider.RequestParameterServletFilter<
/filter-class>
        </filter>
...

        <filter-mapping>
                <filter-name>requestParameterProvider</filter-name>
                <url-pattern>*.jsp</url-pattern>
        </filter-mapping>

        <filter-mapping>
                <filter-name>conversationFilter</filter-name>
                <url-pattern>*.jsp</url-pattern>
        </filter-mapping>
```

You have to adjust the url-pattern according your setup. It should be the same mapping than your MyFacesExtensionsFilter.

Context parameters:

- org.apache.myfaces.conversation.MESSAGER - FQN of your ConversationMessager

Sometimes the conversation stuff has to send some message to the user, to allow to configure this stuff you can configure your own org.apache.myfaces.custom.conversation.ConversationMessager. The best will be to derive your own class from DefaultConversationMessager.

- org.apache.myfaces.conversation.PERSISTENCE_MANAGER_FACTORY - FQN of your PersistenceManagerFactory

To allow to synchronize the database session lifetime with the conversation lieftime you have to provide your own implementation of org.apache.myfaces.custom.conversation.PersistenceManagerFactory.
A sample how it should look like will be shown below.

Unverified problem

imario@apache.org: I have no problems using the RequestParameterServletFilter with the ExtensionsFilter and the DefaultAddResource. Ben, could we discuss on myfaces-dev whats the problem you found?

Finally, if your application makes use of the MyFaces ExtensionFilter, you must use the StreamingAddResource and <t:document*> tags. Some components (e.g. InputDate, InputCalendar...) will not work when using the RequestParameterServletFilter with DefaultAddResource.

- org.apache.myfaces.ADD_RESOURCE_CLASS - org.apache.myfaces.component.html.util.StreamingAddResource

s:startConversation

```
<s:startConversation
   name="myConversation"
   [persistence="true|false"] />
```

This will start a conversation. You can embed this tag within an commandLink or commandButton to start the conversation on a specific action, though, I think the most common use case will be to put this tag right after the h:view tag to automatically start a conversation if the user navigate to the page.

If you set the persistence to true a new PersistenceManager will be created through your PersistenceManagerFactory.

The name is required and has to be unique.

A already started conversation will not be restarted, instead startConversation do nothing.

s:conversation

```
 <s:conversation
   name="myConversation"
   value="#{myConversationBean}" />
```

The data required within a conversation should be put into its own request scoped managed bean.

The above tag will get the bean out of its current scope and put it into the conversation scope.

You can have multiple s:conversation tags for the same named conversation.

s:endConversation

```
 <s:endConversation
   name="myConversation"
   [onOutcome="success[,yes,jo-o]"]
   [errorOutcome="startOver"]
   [restart="true|false|#{valueBinding}"]
   [restartAction="#{restartAction}"]/>
```

Like s:startConversation you can simnply put the endConversation on your page, but in this case I think the most common use case is to put the endConversation tag as child of one of the action tags (e.g. commandLink, commandButton). So the conversation will be ended as soon as the user e.g. press a "save" button.

You can bind the end of the conversation to the outcome of your action method. So only if the action method returns of the given outcomes (onOutcome) the conversation will be ended.

If you setup a errorOutcome and a exception happens during your action method this special outcome (errorOutcome) will be returned insted of simply showing the exception to the user.
In your ConversationMessager you can configure what to do with the exception. The default is to pass the stacktrace as FacesMessage to the JSF framework.

All beans within this scope are deleted afterwards. If your bean implements the ConversationListener interface it will be informed about it, this can be used for cleanup stuff.

With restart/restartAction you can immediately restart a conversation, see in "Advanced Topics" what this is good for.

The signatur of "restartAction" can be

```
public void restartAction()
```

s:seperateConversationContext

```
    <s:separateConversationContext>
        <h:form>
.... commandLinks, commandButtons, navigationMenu, etc ....
        </h:form>
    </s:separateConversationContext>
```

To make the conversation multiple window aware a new parameter will be added to the url called conversationContext.

Now if you require to forcibly start a new conversation context you can use the s:separateConversationContext tag to render links without this conversationContext parameter. That way, if the page requests a conversation (s:startConversation) a new conversationContext will be started to store your named conversations.

s:ensureConversation

```
    <s:ensureConversation
        name="myConversation"
        redirectTo="view-id" />
```

Think about a wizard style page flow with e.g. 4 pages. The first page contains the s:startConversation tag and the last page the s:endConversation tag (within a commandButton for sure).

Now, for some reasons, it might be possible that a user navigates directly to page 2, 3 or 4 and NOT started with page 1 - another reason might be, that the conversation timeout take place and when the user came back the conversation is no longer existent.

This is where the s:ensureConversation tag take place. Put this tag on all pages but the one with s:startConversation (in our case these are the pages 2, 3 and 4).

Now it will be checked if the named conversation myConversation is active, else a redirect to the given view-id will be issued - say to page 1. That way you have a clean restart of the conversation.

Again the ConversationMessager will be used to send a message to the user.

ConversationManager

ConversationManager allows you to easily retrieve your beans from the converation scope.

```
    MyBean myBean = (MyBean) ConversationManager
                .getInstance()
                .getConversation("myConversation")
                .getBean("myConversationBean");
```

PersistenceManagerFactory

The PersistenceManagerFactory is responsible to create (or synchronize) a database session.

TODO: Example of it for e.g. hibernate

Advanced Topics

Parallel/Named conversations

You can have multiple running conversations per conversationContext. I think the most common use case is to have maximum 2 conversation per conversation. One conversation with a larger lifetime and one with a shorter one which deals with persistence.

```
request              = = = = = = = = = = = = = = = = = = = = = = = =
addItem conversation       ======= ======= ======= ======= =======
order conversation         =======================================
session              ======================================================================
```

Think about a order processing system where you have a handful of pages to create an order and add items to it.
Such an application might have these two entities:

- order head
- order item

In this scenario the addItem conversation will be the one which deals with persistence and thus is also the demarcation of your database session. IF the addItem conversation is the demarcation of your database session you have a problem with the order head. Again, after the addItem conversation ends this entity will be detached.

To workaround this problem its best to store the entity id of the order head in your outer converstion (order conversation) and refetch it from the database on request.

Notice: This will NOT hit the database each time. You have the running database session from the addItem conversation and thus the persistence framework will get the entitiy from its cache.

On the end of the addItem conversation the session will be closed and on restart for the next addItem conversation a new database session will be created. This means that NOW a fresh copy of the order head will be read from the database, but only ONCE as then the persistence framework again will get it from the cache.

If you will avoid hitting the database even in this case, you can configure the second level cache of your persistence framework.

If you do it that way, there is no need for a session scope bean and the user can start multiple "order conversations" within the same session.

GREAT!

Suggestion how to design your beans

For a successful work with the conversation tag I propose to separate the conversation scoped data from your normal "view controller" backing bean.

Say you have a backing bean called "EnterBlog" you should also have a bean "EnterBlogData" which will hold the entities only, a simple bean with getter /setters. The "EnterBlog" should be request scoped and can hold stuff like the component bindings and values. "EnterBlogData" will hold your entities and maybe component values too.

While you can use the managed-property feature (faces-config.xml) in this special case I propose to use the value-binding strategy to get access to the "EnterBlogData".

You can do this with a utility function like:

```
  FacesContext fc = FacesContext.getCurrentInstance();
  return fc.getApplication().createValueBinding("#{" + beanName + "}").getValue(fc);
```

The reason for this is, that a conversation can die - and restarted - in the mid of a request, after the restart JSF wont reset the managed-property beans, this is why you have to use the solution above.

Review Data after endConversation

As you have learned, the conversation bean will be removed from any scopy after endConversation so you can start over your persistent session. Depending on your use case it is required to review the currently changed data.

Ok, lets describe the flow:

- you start with a empty page - a new conversation will be started
- the user enter some data
- and press save - which is also the end of the conversation
- the user should be able to change the already entered data

Now, there is a problem, the conversation has been ended and so all entities are detached, you have to refetch the data from the database using the new conversation.

But there is a timing problem.

The conversation will be ended AFTER the save action, so you cant reload the data from within - and much more bad, a new conversation will only be started when the view will be rendered, so way to late to reload the data.

One might argue that you always can do a "layz reload", means, to load the data in any getter requesting data from the entity. But trust me, this is bad, very bad. The reason is, that it can happen, that the bean hasnt fully initialized (values set from the component) at this point.

The solution is to restart the conversation. This can be done by configuring restart/restartAction on your endConversation.

```
<s:endConversation name="myConv" restart="true" restartAction="#{myBean.loadDataAction}" />
```

In your loadDataAction the conversation has been started and you have access to new, fresh conversation beans. Now, you can reload the data which will be associated to the new conversation then.

Why not just NOT end the conversation?

The reason is simple: The user might have navigated many different pages, opened some "browse/search" pages where you loaded a huge number of entities into the session which still fills your memory. Ending the conversation will allow them to be garbage collected (which is what we want), after the review the session will hold the data required to display all the stuff only (which is what we want too).

Additional Informations

- The sandbox examples: http://svn.apache.org/repos/asf/myfaces/tomahawk/trunk/sandbox/examples/src/main/webapp/conversation/

Without persistence, but sufficient to get a clue about how it works.