Shale Dialog Manager

Introduction

One of the frustrating aspects of organizing the flow of control in a web based application is that fact that it is composed of completely disconnected interactions with the client (via the HTTP protocol). The popularity of application frameworks based on model-view-controller (MVC) principles, and particularly the emergence of the *front controller* design pattern, have become the de facto standard architectural approach.

Like other frameworks, JavaServer Faces supports a mechanism to define navigation rules for transitions between views. The actual processing is performed by an implementation of the `javax.faces.application.NavigationHandler`. The standard implementation provided by the framework (which can be customized via a pluggable API) performs transitions from one view to another based on three inputs:

- What view is currently processing this form submit?
- Which of the potentially several actions were invoked? (This allows you to support different "submit" buttons with different functionality, or share actions between, say, a "Save" button at the top and bottom of a table.)
- What "logical outcome" was returned by the action that was invoked?

Basing navigation on outcomes, by the way, assists in reducing the coupling between pages, because the developer that writes the action method is only focused on reporting "what happened" rather than worrying about "where do I go next". This concept is also found in the way Struts has `Action.execute()` methods that return a logical `ActionForward` describing the outcome of performing the action.

However, it is still difficult to reuse individual views in more than one "conversation" or "dialog" with the user, nor to treat one dialog as a "black box" subroutine that can be called by more than one calling dialog. To address these needs, Shale offers Dialog Manager support.

The functionality of this feature was **heavily** inspired by the implementation of Spring Webflow (Preview 2), whose home page is:

http://opensource.atlassian.com/confluence/spring/display/WEBFLOW/Home

API and Implementations

The Shale *Dialog Manager* defines an API that supports access to an abstract "execution engine" that manages the processing flow through a dialog. In addition, multiple implementations of this API are provided that offer different sets of unique features:

- Basic Implementation A relatively simple implementation that models a dialog as a state diagram with four types of states:
  - *Action* - Execute an arbitrary method
  - *Exit* - Terminate execution of this dialog
  - *Subdialog* - Execute another dialog as a subroutine
  - *View* - Display a JSF view (page) and wait for the following submit to execute an application action method

This implementation supports a superset of the functionality that was present in versions of Shale up through 1.0.3.

- State Chart XML Implementation A more sophisticated implementation based on state charts modelled with State Chart XML, which is currently a Working Draft published by the W3C. This technology has grown out of the use of similar techniques in the telephony industry, and Shale uses the Jakarta Commons SCXML library to provide the required execution engine.

The remainder of this document describes services that are available no matter which implementation you choose. See the module descriptions for the implementation modules for details of configuration, as well as the unique features provided by that implementation.

Services Provided

The fundamental APIs that an application interacts with are simple and straightforward:

- DialogContext - Represents the state of an active dialog with the user. There will be one such instance for each window or frame running a dialog, stored in session scope.
- DialogContextManager - Factory for creating new `DialogContext` instances. The Shale Dialog implementation that you select will provide a suitable factory as a session scope managed bean under a well-known key. At most one active `DialogContext` instance can be associated with each window or frame that the user is operating, in association with the same session.
- DialogListener.html - An active `DialogContext` fires events that document interesting changes in the state of the dialog. Interested objects can ask to be notified of such events by implementing this interface, and registering themselves with the `DialogContext` using standard JavaBeans event listener design patterns.

DialogContextManager provides public methods that support the following functionality:

- Create and return a new `DialogContext` instance, optionally associated with a parent `DialogContext` (useful for popup windows that need to coordinate their behavior with the underlying page).
- Retrieve an active `DialogContext` for the current user, based on a specified dialog identifier.
- Remove an active `DialogContext` instance, denoting that this instance is no longer active. As a side effect, the content of the `data` property of this `DialogContext` will be made available for garbage collection, as long as the application does not maintain any other references to the data object.

DialogContext provides the following public properties:

- `active` - Flag indicating that this `DialogContext` instance has been started but not yet stopped.
- `data` - General purpose object made available for storing state information related to a particular active dialog instance. Details of how this property is implemented are specific to the implementation you choose, but will generally default to being an instance of `java.util.Map`. You can also replace this object at runtime with an object that contains state properties specific to a particular use case.
- `id` - An identifier, unique within the scope of the current user, for this particular instance. The Dialog Manager framework promises to transport this identifier along with the JSF component tree, and will use it to regain access to the corresponding `DialogContext` instance on a postback.

- `name` - The logical name of the dialog definition being executed by this instance. This will typically map to configuration information that is specific to the implementation you select.
- `parent` - Optional reference to a parent `DialogContext` instance that we were associated with when this instance was created.

DialogContext provides public methods to support the following functionality:

- Start the active execution represented by this instance, advancing until the instance has displayed a JSF view and needs to wait for the user to fill out a form and submit it.
- Advance the state of the computation represented by this instance, passing in the logical outcome that was returned by the application's action method.
- Stop the execution of the computation represented by this instance, which will cause it to be passed to the `remove()` method on the `DialogContextManager` instance for this user.

DialogListener is an event listener interface that follows standard JavaBean design patterns. Interested objects can register themselves as a listener on a `DialogContext` instance, and be notified of the occurrence of the following events:

- This `DialogContext` instance was started.
- This `DialogContext` instance was stopped.
- An exception was thrown by this `DialogContext` instance.
- A named "state" was entered (details are specific to the selected implementation).
- A named "state" was exited (details are specific to the selected implementation).
- A transition from one named "state" to another was performed (details are specific to the selected implemenation).

Applications that wish to implement listeners are encouraged to subclass AbstractDialogListener instead of implementing the interface described above. In addition to only needing to implement event handling methods you are interested in, this protects the ability to compile your application against future versions of the listener interface, if more event handling methods are added in the future.

Using Dialog Manager

The following paragraphs describe functionality that works no matter which specific Dialog Manager implementation you have selected. Be sure to consult the page for your selected implementation for additional features and capabilities.

Starting A New DialogContext Instance

At most one `DialogContext` instance can be active, in a particular window or frame, at one time. There are several ways in which an application can cause such a `DialogContext` instance can be activated and associated with the current window or frame. Each technique is described below.

Starting A DialogContext Via Navigation

This technique is very useful if your application contains a mixture of pages managed by standard JavaServer Faces navigation, and pages managed by the Shale Dialog Manager. To enter a dialog whose logical name is **foo**, simply have one of your application actions return a

logical outcome string of `dialog:foo`, and a new `DialogContext` instance will be started for you.

In the example above, we used the default `dialog:` prefix value to trigger starting a dialog. You can also specify your own prefix by setting a context init parameter whose name is defined by the symbolic constant `Constants.DIALOG_PREFIX_PARAM` (i.e. `org.apache.shale.dialog.DIALOG_PREFIX`). By convention this value should end with a ':' character to look like a namespace, but this is **not** required.

Starting A DialogContext Programmatically

Under some circumstances, it may be preferable for your application's event handler to decide programmatically which dialog to use, and start it programmatically. To start a dialog named **foo** programmatically, code something like this in your action method:

```
FacesContext context = FacesContext.getCurrentInstance();
DialogContextManager manager =
  DialogHelper.getDialogContextManager(context);
DialogContext dcontext = manager.create(context, "foo");
dcontext.start();
```

(If you are using a version of Shale before 1.1, you have to work slightly harder to achieve the same effect:)

```
FacesContext context = FacesContext.getCurrentInstance();
DialogContextManager manager = (DialogContextManager)
  context.getApplication().getVariableResolver().
  resolveVariable(context, Constants.MANAGER_BEAN);
DialogContext dcontext = manager.create(context, "foo");
dcontext.start(context);
return null;
```

Starting A DialogContext Via URL Parameters

In a use case like a pop-up window, the first request served by the application will be to a new window that is not currently associated with a current dialog. In order for such a window to immediately become associated with a `DialogContext` instance, the Dialog Manager also recognizes the following request parameters, if they are present, and if there is no active DialogContext already:

- **org.apache.shale.dialog.DIALOG_NAME** - The logical name of the dialog to be created and started.
- **org.apache.shale.dialog.PARENT_ID** - (Optional) the logical dialog identifier of a parent `DialogContext` instance that should become the parent of the newly created one. This allows, for example, a popup window to be associated with the `data` element for the `DialogContext` instance associated with the parent window.

Request Processing for an Active DialogContext Instance

Once a `DialogContext` instance has been associated with the current window, the Dialog Manager performs the following tasks automatically, with no need for application interaction:

- Cause a *dialog identifier* for this `DialogContext` instance to be saved and restored as part of the JSF compoennt tree and associated state.
- When a POST request for this window is processed, the active `DialogContext` instance for the logical dialog identifier for this window will be retrieved from the `DialogContextManager` for this user, and stored as a request scope attribute under key `dialog` for easy reference in expressions.
- During *Invoke Application* phase of the request processing lifecycle, the logical outcome returned by the action method will be intercepted by the Dialog Manager, rather than being fed into the standard JSF navigation handler. Instead, the outcome will be passed in as a parameter to the `advance()` method on the current `DialogContext` instance, which will advance the state of the computation until a further interaction with the user is required. Then, the Dialog Manager will create the requested JSF view, and forward to Render Response phase so that this view may be rendered.

Accessing Per-DialogContext State Information

The Dialog Manager provides a convenient place for an active `DialogContext` instance to maintain state information that lasts only for the lifetime of the instance. This is the `data` property of the `DialogContext` instance for the currently active dialog. Each Dialog Manager implementation will provide a default data structure (typically an instance of `java.util.Map`) for this purpose, but you may also provide a JavaBean class that is application specific if you wish.

Note that, due to the combination of the current `DialogContext` instance being exposed as a request scoped attribute under key `dialog`, and the fact that `data` is a standard Java Bean property on this instance, you can conveniently use JSF value binding expressions to bind component values to state information. For example, assume you have provided an application specific Java Bean class for the state information, and it has a `name` property to contain a customer name. You can easily bind an input component to that name, like this:

```
<h:inputText id="name" ... value="#{dialogScope.name}"/>
```

(Prior to version 1.1, use this approach instead:)

```
<h:inputText id="name" ... value="#{dialog.data.name}"/>
```

As an extra value-added feature, if the object you store as the `data` property is of a class that implements the `DialogContextListener` interface, your data object will also be automatically registered to receive the corresponding events. This can be useful (for example), when your application needs to know when a particular "state" has been entered, or whether a transition to a particular "state" came from some other particular "state".

If you intend to leverage this feature, you can optionally make your data class extend `AbstractDialogContextListener` instead of implementing the interface directly. If you do this, you only need to implement the event handling methods that you are interested in, rather than all of them.

Stopping An Active DialogContext Instance

There are two general approaches to stopping the processing of an active `DialogContext` instance:

- Each Dialog Manager implementation will typically define a particular state as an "exit" or "end" state. When a transition to such a state occurs, the Dialog Manager implementation will call `stop()` on the active `DialogContext` instance, which will take it out of service.
- You can also cause the current `DialogContext` instance to be aborted by calling `stop()` on it yourself.

Shale Dialog Manager 2 (SCXML Implementation)

Introduction

The Shale Dialog Manager defines a generic API by which an application may utilize a Dialog Manager implementation to manage conversations with the user of that application. A user may have (at most) one active conversation in each window or frame that he or she is using.

This module contains the *SCXML (State Chart XML) Implementation* of the Shale Dialog Manager facilities. It uses the Commons SCXML library for the dialog state machine execution under the covers, and the dialogs are described using SCXML documents.

Benefits

- SCXML is a W3C Working Draft which may translate to better support in tooling, number of implementations and various runtime environments. It is the candidate controller notation coming out of the W3C.
- SCXML is more closely aligned to state chart theory and UML2, which helps those using model driven development methodologies.
- SCXML semantics provides for much more than the basic Shale dialogs implementation, such as histories, per state contexts, arbitrary expression evaluation, parallelism and the possibility (*currently not available in the shale-dialog-scxml module*) to add domain-specific XML vocabularies via action namespaces. See the Commons SCXML site for details.
- Those developing multi-channel applications, or using frameworks that use SCXML for the controller bits in other contexts (e.g. RDC framework), may be inclined towards SCXML-based authoring for Shale dialogs.

Describing Shale dialogs via SCXML documents

A Shale dialog is modeled as a state machine. The various "state types" that commonly constitute the dialog state machine are described in the Shale dialogs basic implementation documentation.

This section maps these types to the corresponding SCXML snippets appropriate for the Shale dialogs SCXML implementation. The example dialog from the Shale usecases sample application is captured here as a UML state machine diagram and forms the basis of the snippets below.

- Action state instances may be mapped to executable content in UML `<onentry>` (and may be chained similarly).

```
  <!-- An "action" state -->
  <state id="checkCookie">

    <!-- Execute the method binding expression in the onentry block,
        method must take no arguments and return a String. These
        method binding expressions must use the #{...} syntax -->

    <onentry>
      <var name="cookieOutcome" expr="#{profile$logon.check}" />
    </onentry>

    <!-- Check the return value, and conditionally transition
        to the appropriate state. Arbitrary EL expressions must use
        the ${...} syntax. Since transitions are not guarded by
        events, the transitions are "immediate" -->

    <transition cond="${cookieOutcome eq 'authenticated'}"
        target="exit"/>
    <transition cond="${cookieOutcome eq 'unauthenticated'}"
        target="logon"/>

  </state>
```

- View state instances use event guards to wait for postback. The mapping between the <state> `id` and the JavaServer Faces `view identifier` is pluggable. The default mapping is an identity transform i.e. the state identifier is reused as the view identifier. See the DialogStateMapper Javadocs for details. This mapping may be overridden by using the <shale:view> custom Commons SCXML action. See the Shale dialogs custom Commons SCXML actions section for details. Also note the associated best practices when authoring view <state>s.

```
  <!-- A "view" state, the default convention maps this state to
      to the JSF view identifier "/logon" -->
  <state id="logon">

    <!-- Wait for postback event, which is named "faces.outcome"
        The reserved variable "outcome" contains the logical
        outcome, which is used to conditionally transition
        to the next state -->

    <transition event="faces.outcome"
        cond="${outcome eq 'authenticated'}"
        target="exit"/>
    <transition event="faces.outcome"
        cond="${outcome eq 'create'}"
        target="createProfile"/>

  </state>
```

- Subdialog state instances may be mapped to external SCXML documents (describing the subdialog) via the "src" attribute of the SCXML <state> element.

```
<!-- A "subdialog" state, the "src" attribute points to the SCXML
    document describing the subdialog. -->
```

```
<state id="createProfile" src="edit-profile.xml">

  <!-- Wait for <state_id>.done event, which lets us know
      the subdialog has run to completion. This subdialog uses the
      the "outcome" variable to convey its logical outcome to the
      parent dialog (the SCXML <assign> element can be used
      to assign values to existing variables) -->

  <transition event="createProfile.done"
      cond="${outcome eq 'success' or outcome eq 'cancel'}"
      target="exit"/>
  <transition event="createProfile.done"
      cond="${outcome eq 'failure'}"
      target="fail"/>

</state>
```

- End state instances may be mapped to SCXML final states.

```
  <!-- An "end" state, signifies that the dialog has run to
      completion, the default convention maps this state to
      to the JSF view identifier "/exit". -->
  <state id="exit" final="true"/>
```

Once the dialog reaches an end state, the dialog manager cleans up the current instance of the executing dialog.

Using Dialog Manager (SCXML implementation)

To use the SCXML Dialog Manager facilities in Shale, take the following steps:

- Model your dialog as a series of *State*s with transitions between them labelled with the logical outcome that selects that particular transition. A UML State Diagram is a very useful mechanism for visualizing such a model. Then create a SCXML document for each of the dialogs (dialog state machine diagrams can be easily mapped to SCXML documents, see above section).
- Build the views (and corresponding `ViewController` beans, if you are also using the Shale View Controller Support functionality) that comprise your dialog, using standard JavaServer Faces and (optional) Shale `ViewController` facilities.
- Declare your dialogs via an XML document, conventionally named `/WEB-INF/dialog-config.xml`, that conforms to the required DTD:

```
  <!DOCTYPE dialogs PUBLIC
    "-//Apache Software Foundation//DTD Shale SCXML Dialog
Configuration 1.0//EN"
    "http://shale.apache.org/dtds/dialog-scxml-config_1_0.dtd">

  <dialogs>

    <dialog name="FirstDialogName"
         scxmlconfig="firstdialog.xml"
         dataclassname="org.apache.shale.examples.FirstDialogData"
         />

    <dialog name="SecondDialogName"
         scxmlconfig="seconddialog.xml"
```

```
          dataclassname="org.apache.shale.examples.SecondDialogData"
          />

    ...

  </dialogs>
```

- If you have more than one dialog configuration file, or you have defined your only dialog configuration file as a web application resource with a name different than the one described above, use a context initiaization parameter to define a comma-delimited list of context-relative paths to configuration resources to be loaded:

```
  <context-param>
    <param-name>org.apache.shale.dialog.scxml.CONFIGURATION</param-name>
    <param-value>/WEB-INF/foo.xml,/WEB-INF/bar.xml</param-value>
  </context-param>
```

- In addition to the dialog configuration resources defined by this context initialization parameter, a resource named `/WEB-INF/dialog-config.xml` will be automatically processed, if it exists, and has not already been loaded.
- Alternatively, or in addition to the above, any JAR file in `/WEB-INF/lib` will be scanned for configuration documents at `META-INF/dialog-config.xml`. Such resources will be automatically processed, making it easy to define JAR files with dialog configurations and corresponding Java classes and resources, which are recognized simply by including this JAR file in the application.
- To initiate a dialog named "xxxxx", use one of the techniques defined by the Shale Dialog Manager.

Custom Commons SCXML actions

The Shale dialogs Commons SCXML implementation provides a couple of custom Commons SCXML actions out of the box (background reading on custom actions). The first one allows the use of redirects while navigating to a view, and the second allows overriding the DialogStateMapper mapping between a "view" state and the associated JSF view identifier.

- **<shale:redirect>** - Typically used in the <onentry> section of the "view" <state> that should be visited by issuing a redirect.

```
  <onentry>
    <shale:redirect/>
  </onentry>
```

- **<shale:view>** - Typically used in the <onentry> section of the "view" <state>, such that the "viewId" attribute contains the JSF view identifier that should be rendered when in this dialog state.

```
  <onentry>
    <shale:view viewId="/faces/wizardpage3" />
  </onentry>
```

The *shale* prefix used above is arbitrary. The association is made using the namespace URI associated with the prefix (the above custom actions belong to the *http://shale.apache.org/dialog-scxml* URI), so the SCXML document describing the above dialog would need to establish that prefix to namespace URI association, for example:

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
       xmlns:shale="http://shale.apache.org/dialog-scxml"
       initialstate="...">
```

It is possible for application developers to define additional custom actions per dialog definition. For example, a developer may define a custom Commons SCXML action via a class *my.actions.Foo* (which must extend *org.apache.commons.scxml.model.Action*, see background reading link above) and make it available in the namespace URI *http://foo.bar/actions* to the dialog named "wizard" by defining it in the *dialog-config.xml* like so:

```
<dialog name="wizard" scxmlconfig="wizard.xml"
           dataclassname="wizard.Data">

    <scxmlaction name="foo" uri="http://foo.bar/actions"
                   actionclassname="my.actions.Foo" />

</dialog>
```

and further using it in the *wizard.xml* SCXML document like so:

```
<scxml xmlns="http://www.w3.org/2005/07/scxml" version="1.0"
       xmlns:shale="http://shale.apache.org/dialog-scxml"
       xmlns:my="http://foo.bar/actions"
       initialstate="...">

    ...

    <state id="state1">
        <onentry>
            <my:foo .../>
        </onentry>

        ...

    </state>
```

Best practices

The particular usecase of SCXML within Shale dialogs implies certain restrictions on the SCXML document used to describe the dialog. In particular, best practices for SCXML documents used to describe Shale dialogs include:

- A "view" <state> must be a simple leaf state (should not contain other <state> elements and should not have a <parallel> ancestor).
- A "view" <state> must not rely on <onexit> or <onentry> executable content. Such executable content can be moved to a preceeding or following "action" state. This is due to the possibility of browser navigation buttons (back/forward) being used during the dialog execution. The exception to this is the two custom actions described in the previous section, when used as mentioned above.
- All views that participate in a dialog should provide for checks to guard against double submits (see <token> tag in shale-core) and provide "immediate" actions such as a cancel button to exit out of the dialog.