OrchestraDialogPageFlowDesign2

An alternate design for an Orchestra "page flow" system to that presented in OrchestraDialogPageFlowDesign1.

This design is more complex to configure than the original proposal. One specific benefit of the original was that there was "no new xml configuration format to learn". Instead, there was just a naming convention for view ids.

This proposal does require either components embedded in caller and called pages, or xml configuration files for them. However it gives the benefits of making the data flow between caller and callee much clearer, and allows the dataflow to be checked for correctness at runtime (or even at startup time for the xml configuration file approach). It also is less intrusive; it does not require the user's flow pages to be in a specific directory (which is then visible to the user via the url).

Design principles

Starting a flow

Deciding whether to start a new flow is driven by the navigation outcome returned by action methods (or literal outcomes embedded in the command component). In other words, starting a new flow happens only when a command component is activated by the user. Starting a flow with a GET operation makes no sense, as there is no "caller" page to specify parameters for the called flow, and nowhere obvious for the called flow to place its results.

Flow Transparency

The called flow pages are inserted "transparently" into the caller's sequence of pages. A postback of the caller occurs, and it is just "suspended" at the end of the postback phase. The flow runs, and then eventually a flow postback occurs which ends the flow and "unsuspends" the original postback. The rendering phase for that "original" postback then runs as if the flow had never happened - except that backing bean properties have been changed by the called flow.

Java method call analogy

The mechanism for calling a flow should look as much as possible like calling a java static method. Actually, it should look more like calling a java "service" via a mechanism such as java.util.ServiceLoader or JNDI, where the caller looks up a service by name, then casts it to an expected interface type and invokes a method on the interface using a set of parameter values. Note that the analogy is close but not exact because:

- a callable flow only ever provides one "method" that can be invoked, so "interface type" and "method name" are combined.
- a callable flow can return multiple values, not just one like Java can.

Note however that a called flow should be like a static method in that it has its own set of variables (its parameters and local variables) and only interacts with the caller via the parameters and return values. This makes caller/callee interactions understandable, which would not be the case if a called flow could access any data of the caller. This approach is similar to Trinidad pageflow, where imported/exported data is explicitly declared. Spring WebFlow does something similar.

Therefore:

- Every callable flow declares an "interface type" (what logical function it provides) and what parameters it expects.
- A flow caller uses a navigation outcome to specify which flow it wants to invoke (like looking up a java object by service name). It declares what logical function it expects that flow to implement (like casting a service object to a java interface). And it declares what parameters it is passing to the flow.
- The standard JSF navigation rules are used to find a concrete implementation for the abstract service that the caller wants to invoke; the outcome is the service name and the to-view-id is the concrete flow to invoke. Note that because navigation rules support "from-view-id", this allows flexible control over which outcomes (service names) map to which concrete flow implementations (views).
- The "signature" of the concrete flow that is returned by the navigation-case lookup is compared against what the caller expected. It is an error if the called flow does not provide the expected function, or expects incompatible parameters. This is equivalent to the ClassCastException that would occur in java if a service lookup returned an object that did not implement the expected interface.

Configuration Principles

While the information needed to match caller and callee is specific, there are several possible ways that this data can be defined. One is for the caller and callee pages to embed special JSF components that provide this information. Another is for there to be a configuration file "next to" the caller and callee pages which defines this. It is also possible to put at least the caller part of this information into the navigation-case. The first approach (embedded components) will be implemented first. Care is taken in the design to make sure that per-page config files can be implemented at a later date. Other options (navigation-handler etc) are not a priority.

Configuration Examples

Example caller page using embedded components

```
<f:view>

<o:onFlow outcome="doCustSearch" type="com.ops.foo.CustSearch">
  <param name="x" src="#{caller.x}"/>
  <param name="y" src="#{caller.y}"/>
  <return name="z" src="#{caller.z}"/>
  <onReturn action="#{....}"/>
  <modifies componentId="id"/>
</o:onFlow>

<o:onFlow outcome="doAddressSearch" type="com.ops.foo.AddressSearch" mode="modal">
  <param name="a" src="#{caller.a}"/>
  <return name="b" src="#{caller.b}"/>
  <onReturn action="#{....}"/>
  <modifies componentId="foo"/>
  <onFlowBegin>..</onFlowBegin>
  <onFlowEnd>..</onFlowEnd>
</o:onFlow>

<h:commandButton action="xyz"/>
```

Example callee page using embedded components

```
<f:view>

<o:flow type="com.ops.foo.CustSearch" onepage="true">
  <param name="x" src="#{callee.x}"/>
  <param name="y" src="#{callee.y}"/>
  <return name="z" src="#{callee.z}"/>
  <commitWhen outcome="commit"/>
  <cancelWhen outcome="cancel"/>
</o:flow>

<h:commandButton action="commit"/>
```

Example caller with separate per-page config file

Page file "foo/bar.jsp" has matching file "foo/bar.flow.xml"

```
<flowConfig>
 <onFlow outcome="doCustSearch" type="com.ops.foo.CustSearch">
  <param name="x" src="#{caller.x}"/>
  <param name="y" src="#{caller.y}"/>
  <return name="z" src="#{caller.z}"/>
  <onReturn action="#{....}"/>
  <modifies componentId="id"/>
 </onFlow>

 <onFlow outcome="doAddressSearch" type="com.ops.foo.AddressSearch" mode="modal">
  <param name="a" src="#{caller.a}"/>
  <return name="b" src="#{caller.b}"/>
  <onReturn action="#{....}"/>
  <modifies componentId="foo"/>
  <onFlowBegin>..</onFlowBegin>
  <onFlowEnd>..</onFlowEnd>
 </onFlow>
</flowConfig>
```

Example callee with separate per-page config file

Page file "custSearch/search.jsp" has matching file "custSearch/search.flow.xml"

```
<flowConfig>
 <flow type="com.ops.foo.CustSearch" onepage="true">
  <param name="x" src="#{callee.x}"/>
  <param name="y" src="#{callee.y}"/>
  <return name="z" src="#{callee.z}"/>
  <commitWhen outcome="commit"/>
  <cancelWhen outcome="cancel"/>
 </flow>
</flowConfig>
```

Navigation Case Configuration

The JSF navigation rules defined in faces-config.xml files work just like normal. There is no special syntax there at all, which means that IDE tools will continue to work.

To start a flow, there simply should be a navigation rule from the calling page to the flow entry page's viewId using outcome string (as normal). The only difference is that when the calling page has an <onflow> entry matching that same outcome, then the to-view-id page is expected to have a matching <flow> configuration. When that is not true, an error is reported. When it is true, the to-view-id is executed in a new conversation context.

This does mean that looking at the navigation rules gives no hint as to whether the navigation starts a flow or not. But if a user cares about that, they can use a naming convention for their pages that are meant to be flows, eg always start them with "/flow" or add the word "Flow" as a suffix (custSearchFlow.jsp).

The navigation rules for returning from a flow are of course not defined; the exit page for a flow will specify a navigation outcome that matches the "commitWhen" or "cancelWhen" clause of the current flow. Normal navigation then does not occur; instead navigation occurs back to the calling flow. So IDEs that show navigation rules will not be able to show the correct return flow - but that is obviously simply impossible to do as the return address is effectively dynamically determined at runtime.

Tradeoffs of Configuration via In-Page Components

Advantages:

- IDE support when adding the tag (autocompletion).
- Only one file to look at. Simply browsing a page source shows whether it calls flows or expects to be called as a flow, what parameters are passed, etc.
- No custom xml syntax to learn
- Refactoring-friendly. When the page is moved, the info goes with it.

Disadvantages:

- Does require modifying pages
- Cannot do checking of flow data at system startup, as inspecting pages to extract this data is too complex. So errors like calling a non-existent flow or passing incorrect parameters can only be checked when the call is actually made. The xml-config-file version can do this check much earlier..

Notes:

- We cannot support accessing the parameter data until the embedded component of a called flow has executed. But that really only means that this data is not available in the bindings phase and I cannot imagine why parameter data would be needed when configuring bindings.

Tradeoffs of Configuration via per-page Config File

Advantages:

- No change to pages; adding or removing orchestra flow management can be done without touching the page.
- Allows static testing of correct flow declarations; can scan the webapp to find all flow declarations, and detect inconsistent duplicates. Then can scan all flow calls, and detect mismatched types or params.
- Moderately refactoring-friendly. When the page is moved, the flow.xml file just needs to be moved with it.
- The flow.xml files look almost exactly like the in-page-component alternative.

Disadvantages:

- Another syntax to learn.
- The viewhandler logic will be a bit different. First it will need to look for a flow.xml file next to the viewid. When found, it needs to read it and fetch the type. Then it needs to go back and find the onflow.xml file. And the import/export process is then run by ViewHandler before the view exists rather than by o:flow during render. Not too bad though.

Logic Flow

Server-side logic to handle flows goes like this: (should make this an interaction diagram)

A trivial custom NavigationHandler is needed, plus a moderately complex custom ViewHandler.

- Navigation occurs to caller page; view is created. Nothing special happens.
- Postback occurs, the onflow restore places all its data in request scope.
- An action method returns a navigation outcome
- A simple custom NavigationHandler caches that outcome in request scope, for later use.
- viewhandler first discards child contexts:
    - if the current context has children, discard them all. This means that when the "back" button is used to go back to a page that has a different contextId then we cancel the nested contexts. It also means that when we have modal flows, and the user forcibly closes the modal flow somehow then uses the parent window we clean up ok.
      Note that this makes it safe to create a child context then allow a GET to execute it; the GET has the contextId in it which allows us to find the relevant info. If a different request is received for some reason, then we automatically clean up later.
      Issue: if the flow really is in a non-modal popup window then using the parent window for any purpose will kill the flow for the child. Killing the flow isn't too bad; it cannot return anything to the parent so must die. But what happens when a postback specifies a contextId that has been deleted? Do we error, or just start a new context? If just start a new context then this will be very confusing for the user as the page will be in the middle of a flow, but the backing beans will be new (context deleted). But erroring on invalid contextId screws bookmarks; bookmarks will have old contextIds in them that we should just ignore.
      We could keep a list of "deleted" ids in the session, and error on those - sounds good. But for later.
- viewhandler looks at the outcome:
    - if there is no outcome (ie this is a GET):
        - if the url is a "flow url", report an error
        - if we are in a flow then do DISCARD
        - do normal navigation
    - if there is an <onflow> entry for the current view which matches the outcome then do START
    - if there is a <flow> entry for the current context and the commit-on attribute matches the current outcome then do COMMIT
    - if there is a <flow> entry for the current context and the cancel-on attribute matches the current outcome then do CANCEL
    - if starts with current flowPath, do nothing (stay in current flow)
    - if not a flow url, do DISCARD (navigation goes elsewhere)
    - report error (nav to flow, but with no onflow definition).
- viewhandler then creates view as normal
- o:flow render method does this on *first run* only:
    - verify that caller onflow definition's type matches the callee type. Report error if mismatch found.
    - validate parameters match (exact same param and return names)
    - update the FlowInfo with the return flow info declared here
    - run the caller's param expressions using the parent context and store the results in a map
    - run the callee's param expressions reading from the map

START:

- create new context
- look in the request for the caller's onflow info, and select one using the outcome string. If none match (or more than one) report an error.
- create a FlowInfo object that holds the "flow path", and caller info including the matched onflow clause.
- serialize the previous view and save it in the new context

DISCARD:

- optionally, if the destination is a "flow path" url, then report an error
- invalidate and remove all contexts except the root of the current context.
- make the root context the active one.

COMMIT:

- fetch saved flow info from current context
- run the callee's return expressions and save into map
- discard current context, select parent
- run the caller's return expressions reading from map
- if onreturn action defined, then run it.
- if null navigation is returned from onreturn action then
  + set currentViewId to caller,
  + restore the saved caller view tree
  + clear the submitted value of any component specified in <modifies> (if no modifies clauses, clear all).
- if a non-null navigation is returned from onreturn action, then execute the navigation-handler again.

CANCEL:

- discard current context, select parent
- set currentViewId to caller
- restore the saved caller view tree

Process "is a flow" can be implemented in several ways. The easiest are just looking for substring "/flow/" or page name "*Flow.*" in the viewId.

Computing the "flow path": by default, this is the parent directory of the viewId. However when onepage is true, this is the complete viewId. This means that multipage flows must live in their own directory, but that trivial lookup logic which is just one page doesn't need that.

Notes

1. This approach effectively mimics a normal static java method call; the callee declares a type and a method prototype. The caller must then pass the right set of parameters. Like a java classpath, the callee can be anywhere as long as it is retains the same fully-qualified class name. Here, the JSP/XHTML file can be anywhere as long as the "type" parameter remains unaltered. The type value can be any user-selected string, but using the name of the flow backing bean or similar seems appropriate.
2. This should work well with existing navigation-aware IDEs. The navigation file is totally unaltered, and the viewids are also completely normal - except for special mappings needed for "flow:cancel" and "flow:commit" outcomes, but that is easy enough to map to some dummy page.
3. Recursive entry to a flow works naturally. We can just test the outcome against the
4. The START process is effectively enhancing navigation, as if we had added extra information to a navigation case. But the NavigationHandler API sucks so badly, it is easier to do it in this way.
5. The modifies section allows the calling page to control what fields get fresh data after the flow has finished. The whole point of a flow is to change something that the current view is displaying; if the change is just to read-only data then nothing needs to be done as the new data is automatically displayed. And when the flow was started by a non-immediate command then there is no problem as all data was pushed to the model so everything is read fresh. But if the flow was triggered by an immediate command then we have a mix of user data in components that has not been pushed into the model (which we want to keep) and user data in components that is now "stale". We really want to clear out data where the value referenced from the input component has been modified - but that is almost impossible to compute. Instead we need help from the user to tell us which components the "return" and "onreturn" tags have caused modifications to. After the close of the flow, we restore the caller's view tree then render it. Just after restoring it we can clear out any submitted data. It would possibly be even nicer to clear out modified data before invoking the flow, but that means that when a flow is cancelled that data is reset to the model state. It is not normal for a cancelled operation to reset data in the caller.

Issues

1. h:commandButton (unlike h:commandLink) has no "target" property. So we would need a custom or:popupButton which sets the form target, submits the form, then resets the form target. There is still the issue however that on commit/cancel, we want to close the popup and trigger the parent window to refresh rather than loading the caller page into the popup. We need some mechanism for the post-flow-end page to not be the caller but instead a page that just contains "closing...." and some javascript to poke the parent window. We would also need to hack the postback to be a "refresh" type postback where just the <modifies> sections are processed [unless we are really hacky, and temporarily restore the tree, do modifies processing then reserialize it]. Tricky, as the caller-view was stored in the child context which we have already destroyed. Dang.
   2. the "onflow" info needs to be non-transient within the tree because we need it again at the end of a postback phase. It is a moderate amount of data which is a nuisance when client-side state is enabled; it is carried on each request just in case the postback navigates to a flow that needs it.
   3. if data being exported to the called flow is associated with an input component on the current page, then we need that component to push its data into the model in order for it to be available to the called flow. That means that when the button triggering the flow is immediate, then the input component must be immediate too. That is no big deal. However it would be nice if JSF had a way to say "this input component is immediate only when this command component is triggered".
   4. Is the persistence context of the caller passed to the called flow? If not, then the called flow will not be able to navigate any uninitialised references on persistent objects that the caller passes. And it will not be able to pass back persistent objects to the caller. But the called flow can have many conversations, and it is conversations that have persistence contexts. Maybe there can be yet another tag in the onflow entry, <persistence conversation="..."/>

which causes the specified conversation to be created immediately in the invoked context, and for the current persistence context to be attached to that conversation. That can be added later though; in the initial implementation, data passed can be restricted to keys, and the onreturn actions can use the keys to load the object using their own context, or merge it in.

5. We do not support a GET operation anywhere in a flow. This seems ok.

6. What about "ping" type operations, or resource-fetching. We need to ignore these when determining whether to end a flow or not.

7. What about back-button usage? There is code in the view-state-saving to handle fetching an old view tree. But things will get really screwy if we don't have the right context active. The contextId is stored in the url, so if someone goes back across the start of a flow, then the child context will just be ignored, and will eventually time out. That is probably ok. Possibly when a context is activated and it has children we could immediately discard those children? Forward buttons would then not work, but that's ok.

8. Access to global data. We encourage users to create a "global conversation" and load what used to be session-scoped data into that. But child contexts will not have that information available - unless explicitly passed as parameters. Maybe a global "params" property needs to exist that is always exported to flows? Otherwise people will fall back to putting "global" data into the http session.

Optional extensions

per-page navigation

Possibly we could put navigation rules in the onflow.xml file. The custom navigation handler could look in there and if a mapping is present then use that data instead of the normal navigation case. This gives "decentralised" navigation definitions that are nicely coupled with flows:
<onflow outcome="..." type="...">
...params...
<view>newViewId</view>
</onflow>
The syntax is identical to the normal non-navigation case, except that there is a view section.

Rejected approaches

- having the onflow info nested within the commandButton. This implies clearly that the command button is *known* to invoke a flow which has a matching flowType. This isn't too bad, but doesn't elegantly fit with JSF's abstracted navigation concept; it should be possible to configure navrules to go anywhere. Still, we could just say that a nested o:onflow indicates that the navigation must go to a flow *of that type*. But an action can return any of N outcome strings, so we would need to then select by outcome.
- Using custom elements in a navigation-case. In JSF1.2 and later, the xml schema for navigation-case allows extra elements to be added (as long as they are in a "foreign" namespace). But the JSF spec provides no easy way to get at this data. So best to avoid. And anyway, navigation-case entries are ok for annotating the "from" part of a call, but provide no way of annotating the "to" part of a call.
- Using annotations to specify flow parameters. It would be possible to use the Orchestra ViewController to look for the controller class for the called view, and then look for annotations on that class to see if it is a flow and what parameters it expects. However this approach does not make sense for the caller, and a mixed system where only one half can sensibly use annotations is not nice. And anyway, this data really belongs at the "page" level rather than backing-bean level. Anyone writing a page that will call a flow needs to be able to easily see what parameters they need to pass; that info is much more easily available via info in the page or info in an associated flow.xml file than when it is attached to the view controller class for the page.

Popup modal window handling

- postback occurs
- navigation to new flow with "modal" flag set detected
- create a new child context but do not activate it
-

A custom component
<o:modalFlowButton target="foo"/>
renders the necessary html to present a submit button that sets form.target and submits, plus script to handle the "closing window" callback.

When the new child context is created, the modal flag is set in the FlowInfo.

When a "cancel" outcome occurs, the ViewHandler directly renders a simple page with javascript that closes the page and does nothing else. When a "commit" outcome occurs, the ViewHandler directly renders a simple page with "closing window..", plus javascript that locates the original o:modalFlowButton DOM object by clientId in the parent window, and calls the "commit" function on it before closing itself.

Refreshing the parent window is still tricky. We need to cause a postback which

Issues:
(1) not all browsers allow real modal popup windows to be created. But if parent window is closed or navigates somewhere else then that "close flow" javascript cannot run. That's ok, just ignore failure and close self. (2) if user does not have javascript enabled, then modal flows are not possible. We could default to a non-modal flow in that case, or just do nothing. There really isn't much that can be done about this as the modal flow *needs* to close its window at the end, and cause the other window to refresh. And that cannot be done without javascript. Possibly the "closing window" message could just be left there and the user has to close the popup manually. But requiring the user to refresh the parent page manually to see the changed data is real ugly so better not to allow it at all. (3) how do we handle the "modifies" section? We cannot mess with the data before serving the "close" operation, because we have no way of saving back the view tree in a way that guarantees it will be picked up by the postback. And in the case of client-side state it's really tricky to mess with the client state. But if we delay messing with it until the next postback we no longer have access to the child context, which is where the selected onflow data is held. Maybe the closing window can tell the custom o:modalFlowButton what the selected outcome was, and then it can include that data in the postback so that we can re-select the onflow block and then execute just its modifies section.

Alternative: we could hook into the tomahawk AJAX-style stuff, and send an "update" message that updates fields on the client. But again tricky as we have to mess with the view state. Better not to.

Modal flow handling

- postback occurs
- navigation to new flow with "modal" flag set detected
  + the onFlow clause needs to define "onFlowBegin" and "onFlowEnd" scripts. These scripts must be defined in the calling page, and take a url as a parameter. The onFlowBegin script is responsible for showing the "popup" window and loading the specified url into it. The onFlowEnd script is responsible for hiding the "popup" window and doing a GET or POST of the entire page to the specified url.
- we immediately create a new child context, but do not activate it (as the render phase needs the parent context active)
  Store the curent view tree in the child context.
  Store the selected o:onflow into the child context
  execute the param expressions in the parent context, and store the resulting map into the child context.
- store the url of the flow entry page into a request-scoped var. This url includes the contextId of the child context.
- re-render the current view (like null navigation). The rendered o:modalFlow tag this time detects the new-flow url in the request and renders javascript to open the iframe and load that url
- When the page is processed on the browser, a GET is executed to populate the iframe. The new contextId embedded in the url will automatically cause the child context to be activated. We simply allow normal processing to happen; the new view is created and rendered. The o:flow tag in the page will execute, causing the param expressions to import data from the map previously pushed into the child context.
- When the flow eventually navigates to an outcome that is "cancel" then we render an empty page with just enough javascript to close itself.

- When the flow eventually navigates to an outcome that is "commit", we execute return params, then deactivate *but not discard* the child context and execute parent return stuff to update parent context. We then render an empty page with just enough javascript to tell the parent to do a GET or a POST, setting a magic commitContextId=?" or "cancelContextId" field which specifies what child context completed.
- On restore-view, when commitContextId is set then we deserialize the view from the specified child context, clear the modifies fields, execute the onflow return statements, etc. Then remove all child contexts and set renderResponse=true.

The only issue is knowing whether the GET statement belongs to the current context, or whether:

- this should trigger a subflow
- this should trigger discard of the current flow

I don't think we ever need to support creating a subflow with a GET; such a flow could never have any import or export expressions or return address, so is useless.

We can simply look to see whether the url matches the flowPath in the current context to know if a discard is needed. Recursive flows are fine; a flow just navigates to an outcome that it defines itself as an o:onflow. At that point we create the child context.

Note: this is no dangerous timing issue here; if the iframe fails to pop up for any reason, then the next request will be for some parent of the current context

Note: storing the caller view state in the context is mandatory for server-side-state-saving because we do not know whether the flow will push the saved state out of the cache. That would be bad. And for non-popup client-side state it is also mandatory, as the hidden field is long gone. It could be skipped in the case of client-side-state in a model flow, but that would be rather inconsistent.