An Oracle White Paper
April 2011

Task Flow Design Fundamentals

Overview

Controller task flows are one of the key elements of any Oracle Application Development Framework (ADF) application. If you are from a JSF or Struts background you may want to equate task flows with the page flow in those controller frameworks but this would be vastly under exploiting the capabilities of the ADF Controller. This document discusses some of the considerations and best practice for designing task flows that take full advantage of the controller's functionality. This document assumes that you are already familiar with the key concepts and components used by the ADF Controller. If you are not, then take the time to read up on this subject in the *Fusion Developers Guide for Oracle ADF* book at http://www.oracle.com/technology/documentation/jdev.html, before you continue.

What type of Task flow?

The first question that is often asked by developers as they first start to work with the ADF Controller is the distinction between bounded and unbounded task flows. Any projects that use the ADF Controller will have an unbounded task flow definition (adfc-config.xml) created by default. From a functional perspective there are four key considerations relating to unbounded task flows to take into account here.

1. There is only one unbounded task flow within an application. Even if you use multiple definition files for unbounded task flows within the application, they will be combined into a single flat namespace at runtime with the entire flow being loaded into memory at startup.
2. Any view activity in the unbounded flow is URL accessible by the user.
3. Unbounded task flows are restricted to view activities that are implemented as whole pages.
4. Bookmarking is only supported by views in an unbounded task flow and is not available in bounded task flows.

Given these considerations we can see that, in reality, the unbounded task flow should only be used to define entry points to your application, for example the home page, pages that will be deep linked from emails and so on. Within a typical application that makes full use of fragment based page assembly this often means that there is in fact only a single view activity in the unbounded task flow and nothing else.

Bounded task flows should be used for every other situation. In most circumstances these bounded task flows will be hosted within the core page on the unbounded task flow and, as such, will be made up of JSF fragments rather than whole pages.

General Principles of Task Flow Design

State Management

As with any Java EE application developers should pay close attention to the handling of any objects that they create. In general session scope managed beans should only be defined at the adfc-config level (unbounded task flow) and should only be used for objects that truly need to exist for the whole session. Objects that need a scope longer than request or viewScope should generally use pageFlowScope in preference to session scope, as this will enable state isolation and a more component oriented development approach for the application. Storing flow specific information in pageFlowScope will also to limit the long term potential memory requirement of the application as the relevant pageFlowScope for a particular task flow instance will be automatically cleaned up on exit.

For page fragments needing to pass transient state consider understand the lifetime differences between request and viewScope when deciding where to handle this state. You may find that, in the context of fragment based task flows, using viewScope in preference to request scope for a more predictable behavior within the context of the sub-flow that contains the fragments.

Backing bean scope should be used for declarative components which be utilized several times in the same UI fragment, for example, stamped out within a table or other iterating component. This will ensure that each instance of the component will maintain its own independent state. Backing bean scope can also be used for beans managing Regions, which likewise may be utilized several times on the same page.

To help with understanding each scope refer to the diagrams in appendix A.

Exception Handling

Your task flows should always contain an activity marked as an exception handler. This activity does not have to be a view activity it can be a method or router (our preferred approach) with control flow rules to subsequent activities. When using task flows which are wired together with task flow call activities, having an exception handler is not absolutely required. In that case any exception will bubble up to the parent. However, this does constitute a logical information leak in the task flow, and we would always recommend that you provide an explicit handler. In the case of task flows that will be embedded within regions on the page, you must provide an exception handling activity.

To ensure a consistent approach to your exception handling we recommend the following practice:

1. Create an exception handling activity in the form of a router activity in a task flow template (see below). Reference this template from the bounded task flows that you create
2. The task flow exception handler (router) should have a default navigation case defined as “unhandled”. Unhandled exceptions are those exceptions that are not handled by the exception handler in the template (so anything specific for a bounded task flow that cannot be generalized).
3. A wildcard navigation case “unhandled” is created in bounded task flows that inherit the task flow template. This “unhandled” flow then goes to a router activity that can access the Controller context for the exception message so you can do task flow specific operations

Task Flow Templates

Task flow templates allow the developer to define common functionality to be inherited by or copied into flows created subsequently. If you are using an inheritance approach, the template should be used to define common task flow settings such as common managed beans, exception handling flows as mentioned above, logging, audit and transaction behavior.

Task flow activities that are defined within an inherited template task flow are not visible in the design time view of the dependent flows (although the namespaces will be combined at runtime). This being the case, you should not define activities that you will want to create explicit control flow cases to and from within the template as you will not be able to draw this relationship to these activities in the consuming task flow's diagram.

If you elect to copy rather than reference a template task flow, keep in mind that subsequent changes to the template will not be intertie by the newly created flow. However, the copy approach for templates does have its place, for example:

A CRUD task flow that operates on an entity such as Person, Order or Store will need to create, update, delete and show the entity. For this you would use similar process steps such as a router as the default activity, a view for edit, a view for read only display as well as method call activities to query an entity object and to delete it. A blue-print like this, which can also contain property settings regarding the isolation and transaction level of the task flow, can be defined in a task flow template and then used as a copy source when implementing a flow. In such a case you don't need task flow template inheritance. When using this approach, be aware that you need to ensure uniqueness in the task flow metadata id values, which are used or MDS customization.

You should also standardize on a naming scheme for any activity, parameter or managed bean defined by a task flow template to minimize the risk of name collision and inadvertent overriding of the template definition.

Control Flow Rules

With respect to the use of explicit verses wildcard control flow rules within a flow there is no hard and fast rule. However, we do advise that you use the wildcard capability of control flow rules to help to keep your task flow diagrams clear and understandable. For example, commonly used actions such as "cancel" or "return" within a task flow may be accessed from multiple activities, so a single wild card rule can keep the diagram much cleaner.

For wild card rules, make sure to use a consistent naming convention so that a developer can quickly identify the fact that a particular outcome is mapped to a wild card rule. Using a prefix to the rule name such as "global" can be useful here.

Flow Notes, Annotations and Other Documentation

Take advantage of the capabilities provided by diagram annotations and attachments to add value to by documenting the task flows.

For more complex documentation requirements, consider embedding an additional HTML document within the task flow. You may want to establish some standard templates for describing each of your task flows and a standardized location for this additional documentation.

Task Flows as Components

Overview

If you are using task flows as a re-use mechanism where they will be handed between developers as components you will need to pay special attention to the interfaces between the task flow and its consuming page (or flow). Although we tend to think of bounded task flows as being fairly simple components with parameters for input and return values for output, it is better to look at them as atomic services that describe units of work. Experience shows us that even apparently atomic services are much more complex than you might anticipate. The developer needs to understand in which ways a task flow can "leak." or otherwise depend on external information.

We cover the following interface points in more detail next:

- Naming considerations
- Parameters
- Return values
- Navigation side effects
- Contextual events
- Data control scope
- Session scope variables
- Request scope variables
- Security roles
- File Upload
- Other contextual information
- Resource Bundles

Naming Considerations

A key consideration when creating a taskflow is its namespace. When bounded taskflows are used as components in regions, they are identified uniquely by a combination of the following components:

I. Path to the taskflow definition file relative to the HTML root

II. Name of the taskflow definition file

III. ID of the taskflow within the definition file

Thus a typical region binding might look something like the following:

```
<taskFlow id="addressTF1"
          taskFlowId="/WEB-INF/addr/addressTF.xml#addressTF"
          ...
```

However, as well as the name and location of the taskflow definition being important, you also need to pay attention to the path (relative to the HTML root) of page fragments used within the bounded taskflow. This is especially important with taskflows that are deployed as ADF Libraries. Even though ADF libraries are deployed as separate JAR files, the contents of those archives are all combined into the namespace of the running web application. Thus, if a taskflow inside of an ADF library references a view such as /home.jsff then this could end up overriding a similarly named /home.jsff in the web application consuming the library, or one of its other libraries. To ensure that there is little chance of such namespace clashes occurring, identify, and use, a suitable path structure to uniquely define the views that correspond to a particular bounded task flow's view activities.

For example, in the case of the taskflow illustrated above. The taskflow definition is:

```
/WEB-INF/addr/addressTF.xml
```

Correspondingly we would recommend that the page fragments consumed by the taskflow are stored in the following directory (or subdirectories of this).

```
/addr/addressTF/
```

As well as paying attention to the physical location of taskflow definitions and view activities you should of course define a suitable Java package structure for any classes used uniquely within the flow.

Parameters

Not surprisingly parameters play an important part in the API of a task flow and guarantee loose coupling between a flow and its caller. When defining parameters, the following rules should be followed

1. Use the Required attribute correctly for any parameter to ensure that the design time and runtime environments can identify that a particular parameter is optional or required.
2. When defining a parameter do not rely on the implicit parameter storage in the pageFlowScope that will take place if do not explicitly map a "value" (destination) for the parameter. You should always explicitly define the matching pageFlowScope managed bean in the task flow definition and map this as the value parameter of the parameter. This destination bean definition can contain the type of the parameter being passed and will ensure that it is type checked at runtime, raising an exception if the parameter is of the wrong type. Having a managed bean definition within the task flow will also ensure that the design time will correctly support and audit usages of that data in the subsequent activities within the flow.
3. When mapping expressions within the flow which need to access input parameters, be sure to map these to these receiver managed beans (marked with a bean icon in the expression editor), rather than implicit parameter name variables (marked with a green cube icon in the expression editor).

Return Values

Return parameters, like input parameters to the flow are a key part of the API, however, the results are not type-checked, so again you should define the receiving bean explicitly within the calling flow to ensure a runtime type check.

Navigation Side Effects

The act of navigating within an embedded task flow can elicit action in the parent task flow in two separate ways:

1. The Parent Action activity provides a way for the developer of a bounded task flow to trigger an action in the parent. As such any parent outcomes that a flow might raise are as much a part of the API of the flow as parameters or return values. You should limit your use of such activities to a small number of well defined control flow rules and document to the consumers of the task flow exactly what the event points are.
2. Region Navigation Listeners are specific to the case of fragment based bounded task flows embedded within a page. In this case, the containing page has, via the regionNavigationListener, access to the navigation that takes place within the flow. In this case, the wiring is passive from the point of view of the embedded flow. As such although the region listener provides a way for some information to leak out of the task flow, it is not really part of the formal task flow API. Authors of reusable task flows should consider more explicit mechanisms for notification of the parent if this is needed, for example contextual events or parent actions.

Because of these possible side effects we recommend that the best use for the Parent Action activity type is for when a bounded task flow is closely coupled with its parent. As such it is reasonable to use a Parent Activity to elicit navigation between nested task flows that are bundled within a single reusable component. It is not a good idea to use such an approach on the top level task flow of a component, however, as this will require assumptions about the structure of the consuming task flow. In such cases the looser coupling provided by contextual events should be used.

Contextual Events

Contextual events provide a mechanism on the ADF binding layer for creating a loosely coupled publish and subscribe eventing model between task flows. Events can be both emitted, and subscribed to, from an embedded task flow and can carry a data payload. They are often used for synchronizing data display within a system.

It goes without saying that contextual events provide an extremely powerful tool in the task flow designers armory and should be fully documented, for both the inbound and outbound cases.

When considering a design that will use contextual events be sure to note the following:

- Contextual events use the ADF bindings mechanism to handle both registration and event propagation even if you are not using ADFm for the rest your application.
- Contextual events will only apply when fragment based task flows are embedded within regions on a page.

- When running a task flow that inherits a shared Data Control scope there is probably no need to use contextual events for data synchronization, it should be sufficient to refresh the containing region with PPR to pick up the latest row currency and refresh the data viewed.
- The payload passed with the event does not have to be a single simple scalar type. A receiving method should take advantage of this and provide as many typed arguments as are required to effectively implement the API required

Data Control Scope

A key consideration in any task flow which utilizes ADF binding and data controls is the data-control-scope (set through the Overview > Behavior tab in the task flow editor). By default, this value defaults to true and as such, it implies that you, the developer, expect to be inheriting some implicit state from the data control context. This information would include:

- Anticipation of existing data structures and attributes such as row currency within a collection (e.g. View Object result set).
- A shared transactional context, where commit and rollback within the task flow will have the side effect of committing the parent data control context and vice-versa.

If data control scope is not shared then be aware that although this will maintain a stricter encapsulation of functionality within the task flow it comes at the expense of more database connections and memory usage for caches (assuming that ADF BC is being used).

In your documentation for a task flow be extremely clear about if data-control-scope is shared or not, and if it is shared what dependencies there are, if any.

Session Scope Variables

As a general principle Session scoped attributes within an application need to be managed with care. They are difficult to document and could create many hidden dependencies within your application. As a general rule in any ADF application, Session scoped variables should only ever be created as an exception.

Session scoped (and indeed application scope) variables are, of course, always available from all task flows within the application. It is recommended that you never use these in anything other than a read-only context and even then, it is better to map the data into a pageflow scope variable so that the task flow has a private copy.

If you do access session scoped variables from within the task flow you should be aware that multiple concurrent task flows could write to that state. Therefore from the perspective of a particular flow the value may mutate over time.

Session scope and application scope attributes are difficult to document and it is hard to identify where they are used and when they need to be cleaned up. In addition, because session and application scope variables belong to the parent application they create a dependency in the task flow to the execution environment, reducing re-usability.

If you need to pass data asynchronously to an outer flow use contextual events as the mechanism rather than session variable.

Request Scope Variables

The accessibility of session scoped data within a called task flow is self evident. However you should also be aware that requestScope may slightly overlap with the creation of a flow and allow information to leak into the task flow without passing through the parameter mechanism. With the possible exception of the special case mentioned for setting parameter attributes you should not rely on requestScope as a way of passing information. If data is coming off of the request, map it into the flow as a formal parameter.

Security Roles

If a task flow carries out explicit permissions checks; either on called task flows, view activities or using EL to evaluate permissions then some additional considerations come into play. When task flows are packaged into an ADF library there is no way currently to package up permission metadata as part of the component. This mandates one of two approaches to handling security:

- Document the requirements and package the policy file fragment containing the permissions as part of the jar. Have consuming developers manually merge the permissions into their jazn-data.xml file.
- Define parameters for the flow that will be used to define permissions at the flow level - for example you might define a task flow parameter “isAdministrator” which is populated by the consuming developer with EL which will make that decision. This localized permission proxy can then be used within the flow.

The Special Case of File-Upload

For a fragment based bounded task flow that uses an af:inputFile component the consumer of that flow will have to ensure that the enclosing page af:form component has the *usesUpload* property set to "true."

This of course is an external dependency that you must document for the consumer of the task flow.

Configuration Parameters and Connections

In most cases you should pass any information that is required into the task flow as formal parameters. However, be aware that task flows may depend on other external resources such as connections, queue names or context parameters in the web.xml or other configuration files such as weblogic-application.xml. Any such dependencies should be documented.

In the case of configuration parameters that are critical to the operation of a task flow you should raise a runtime exception if the parameter or resource is not correctly configured.

Utility Classes, Singletons and Other Shared Code

In most cases your applications will share a small set of convenience classes for activities such as logging or expression language evaluation. These classes should not of course be maintaining state in most cases (see the notes on Session State above), however, you do need to identify the dependency on these classes, and their versions, if they are packaged externally to the ADF library that is encapsulating the taskflow.

Resources

Generally, any resources such as strings, JavaScript files or images should be packaged up within the task flow jar making it self contained in this respect. However, inevitably, there is a chance of some reliance on common resources such as common error messages or icons which you have no desire to duplicate. Such shared resources should be packaged independently from all of the consumers rather than being embedded in the parent. As such both the parent and child task flows can inherit and document a dependency on this shared resource library.

Other Design Considerations

Unbounded ADF Taskflows and ADF Libraries

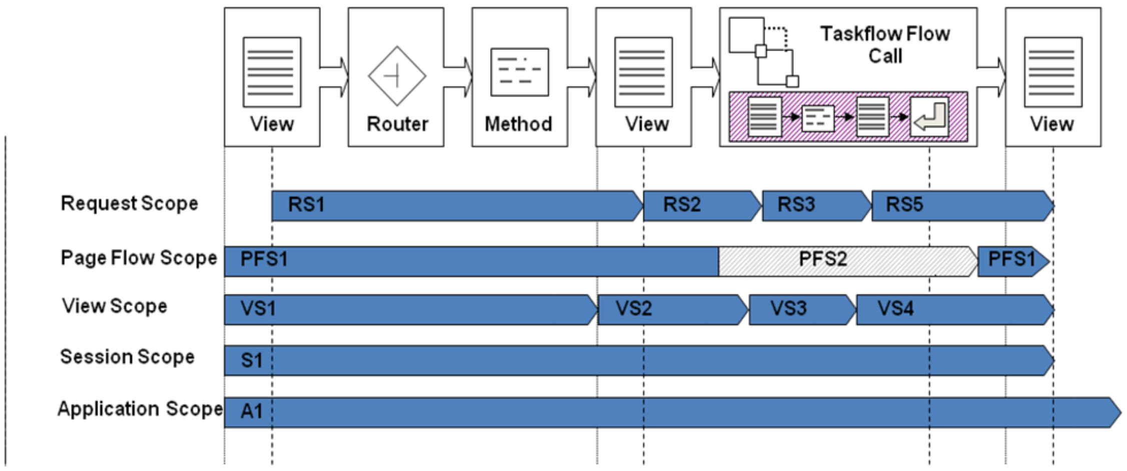
Because all unbounded taskflow definition XML files are combined into a common namespace at runtime, you should avoid including such definitions into ADF libraries unless you are extremely careful with naming and documentation of naming. Certain features of unbounded taskflows such as the XML menu model and the design time support for taskflows do not expect the unbounded taskflow definitions to be distributed across read-only libraries in this way and may not function as anticipated.

Conclusion

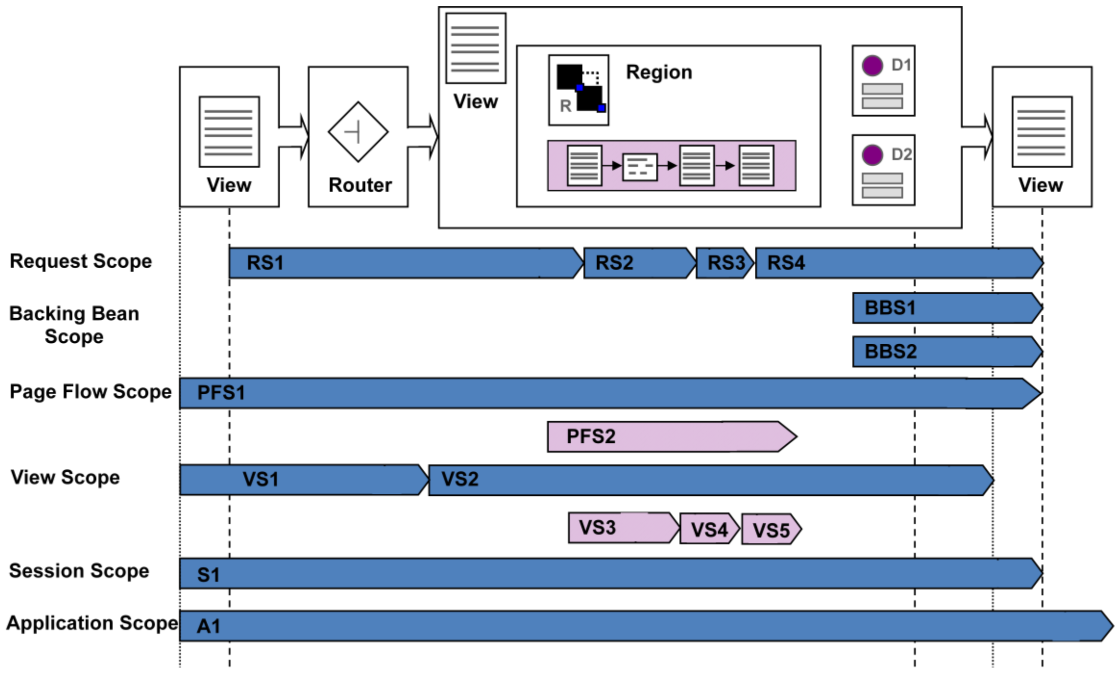
This paper illustrates some of the key practices and design considerations for ADF Controller task flows. When creating task flows for shared use you should pay close attention to the various interface points detailed here and carefully harden and document your flows so as to minimize the risk of misuse or surprise at runtime

Appendix A - ADF Controller and JSF Memory Scopes

Scopes in the Context of Full Page Flows



Scopes in the Context of Fragment Based Flows



Oracle is committed to developing practices and products that help protect the environment

Task Flow Design Fundamentals
Version 1.1, April 2011
Author: Duncan Mills
Contributing Authors: Frank Nimphius

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com