

The logo features the letters 'ADF' in a bold, red, sans-serif font. The 'A' and 'D' are connected, and the 'F' is separate. To the right of 'ADF', the words 'Architecture Square' are written in a blue, sans-serif font. The entire logo is enclosed within a circular graphic composed of several overlapping, semi-transparent white rings that create a sense of depth and motion.

# ADF Architecture Square

## Oracle ADF Task Flow Transaction Fundamentals



[twitter.com/adfArchSquare](https://twitter.com/adfArchSquare)

### Abstract

Task flows are an integral part of Oracle Application Development Framework (ADF) applications built with Oracle JDeveloper 11g. A task flow is a modular and reusable unit of business navigation between views and non-visual activities like routers and methods. Through their design task flows provide opportunities such as reuse, the ability to map to business processes, and compose the overall application architecture.

Within the context of task flows they also support the concept of the *transaction* allowing a collection of work to be committed or undone in its entirety. This whitepaper discusses the concepts and features round the transaction and data control scope features provided by task flows to allow ADF developers to choose the correct combination of options to meet the user requirements.

Author:

Chris Muir

Date:

07/AUG/2012

## Introduction

Oracle JDeveloper 11g introduces the powerful concept of task flows to the Application Development Framework (ADF) that goes beyond the limited page flow facilities provided in JavaServer Faces. Task flows enable the design of a modular set of views and flows that can be reused and coupled with other task flows to create a larger application. Task flows allow developers to align web applications closely to the concept of business processes rather than a disparate set of web pages strung loosely together by URLs. Overall ADF task flows are a key concept in defining the architecture of an ADF application.

The transaction and data control scope behavioral options available to bounded task flows provide a sophisticated set of functionality for spawning and managing one or more transactions during an ADF user's session as well as sharing state between disparate parts of the application. Traditionally applications didn't require such features but with the increasing demands from users and the improved capabilities of technology such functionality is becoming a requirement for contemporary applications.

To explain the requirements around transactions let's discuss an example. Imagine an application to support call center operators who are receiving orders from customers. First the operator must take the customer's delivery address, and order one or more items at a time. Obviously any application we deliver to support the staff must aid taking orders, orders are our business's bottom line.

Taking orders from customers presents many challenges though. Customers give the wrong address, add new items to existing orders, remove other items, change their mind, call center staff don't have an easy job. It would be made even more difficult if the system we provided to support them didn't assist with this chaotic process. If staff record an order's items and the customer decides to change their delivery address, we don't want to undo all the work to start again and reenter the address. The entry of the address should reside in a separate transaction to recording the order items. It's this sort of problem the transactional capabilities of ADF task flows are designed to solve. And the benefit is to the bottom line of the business in processing customers' orders efficiently.

This paper is designed to assist you in understanding the task flow transaction and data control scope options available to you in order to build a sophisticated and contemporary application.

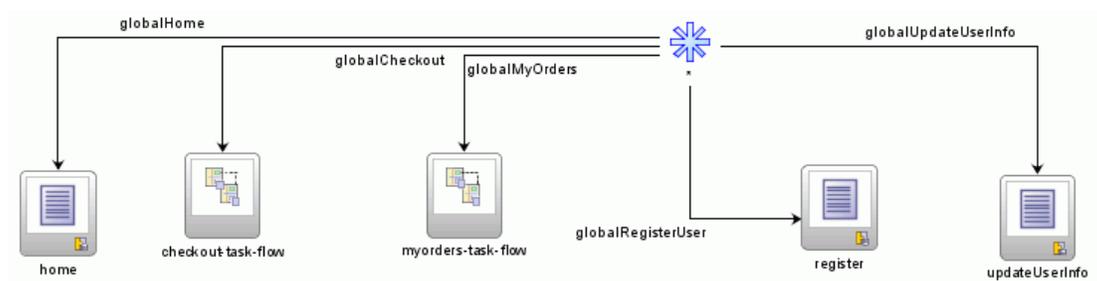
## What is an ADF Task Flow?

Traditional web page development involves a series of pages connected by `<a href>` tags. Trying to discover how a user flows through such an application is difficult as it requires the inspection of the `<a href>` tags in every page to know the navigation paths.

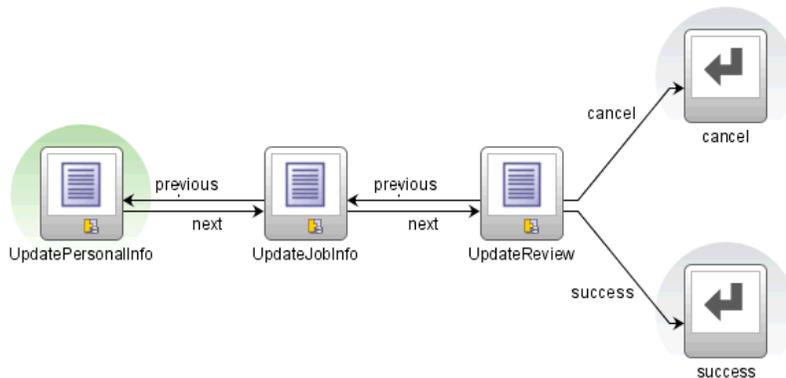
JavaServer Faces overcame this difficulty by prescribing the page and navigation rules between them in a `faces-config.xml` file. This makes it much easier to define and determine the path the user will take in our web applications, particularly important to enterprise applications where there is the need to take users through a prescribed set of steps.

Oracle ADF 11g takes the JSF implementation one step further and introduces the concepts of task flows. ADF task flows compared to JSF allow not only pages to be specified in the application's flow, but also router logic, method calls, transaction support, save points and more.

An ADF unbounded task flow mimics the `faces-config.xml` implementation of JSF, where essentially there is no entry or exit point to the application. With a bookmark containing a URL of any of the pages within the unbounded task flow, the user can leap into the application and start navigating the defined paths from there.



**Figure 1 - Unbounded task flow**



**Figure 2 - Bounded task flow**

It is however the bounded task flow that brings true power to the ADF ecosphere. A bounded task flow is analogous to a Java method, with a defined name, parameters, entry and exit points. Each bounded task flow can be designed to provide a discrete function. Bounded

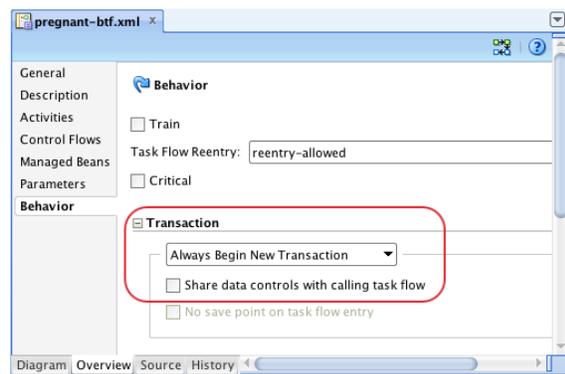
task flows can be assembled together through task flow calls to create logically larger functions or composite applications. Overall bounded task flows allow a concept rarely seen in web application, that of modularization and reuse.

Readers should have a good knowledge of ADF task flows before continuing with this document. At the conclusion of this document you will find a number of references can be found which will assist in learning task flow features.

## Who Defines a Transaction in ADF?

In ADF transactions are defined and owned by the underlying business service implementation used within the Model layer of the application. As example ADF Business Components (through the definition of root Application Modules) take out connections and transactions with the database. Such services are then exposed to the ViewController layer through an abstraction known as a *Data Control* for the ViewController to work with.

As our application can utilize multiple data controls, the ADF Controller (ADFc) through the facilities provided by the task flow transaction and data control scope options, allow one or more data controls *to be grouped together and committed or rolled back as a group*. The underlying business services still own the transaction and ultimately execute the



**Figure 3 - A bounded task flow's transaction and data control scope options**

commit and rollback operations, but the ADF Controller defines the boundaries of the task flow transaction, namely where the transaction starts and stops and which data controls are involved.

These task flow transaction and data control scope options are available by opening a bounded task flow in the IDE, selecting the Overview tab at the bottom of the document window, then the Behavior node as shown in Figure 3.

The associated *Transaction* drop down, and the checkbox entitled *Share data controls with calling task flow* control the task flow transaction settings. In manipulating these options the relating XML file for task flow is populated as follows:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<adfc-config xmlns="http://xmlns.oracle.com/adf/controller" version="1.2">
  <task-flow-definition id="task-flow-definition">
    <transaction>
      <new-transaction/>
    </transaction>
    <data-control-scope>
      <isolated/>
    </data-control-scope>
    <use-page-fragments/>
  </task-flow-definition>
</adfc-config>
```

It's not necessary to be familiar with the XML structure nor definitions. However for the purposes of this paper we'll use the XML element names for describing the options rather than the somewhat convoluted labels in the Behavior node of the bounded task flows Overview tab.

Within this document the drop down options will be referred to as the transaction options, and the checkbox as the data control scope options.

## Task Flow Transaction Vocabulary

Before discussing in detail what the actual task flow options do it is useful to have an understanding of the vocabulary used by other programmers. Learning the vocabulary will assist you when reading documentation, blogs and OTN forum posts, as well as speaking to your ADF peers and logging requests with Oracle Support.

Firstly, as we saw in Figure 1, the transaction options and the separate data control scope options are edited through the bounded task flow's Behavior node on the Overview page. Note these options are defined on the task flow itself. The caller does not define them. So whenever these options are discussed it is from the context of what was defined in the task flow that has been called. Remember this when another programmer describes any transaction options.

For the bounded task flow transaction option itself there are only 4 options:

- "<No Controller Transaction>"
- "Always Begin New Transaction"
- "Always Use Existing Transaction"
- "Use Existing Transaction if Possible"

Sometimes you might hear somebody use terms like New Transaction, Sharing Transactions and Chaining Transactions but these terms don't align well to the above four options. As an example, both the "Always Begin New Transaction" and "Use Existing Transaction if Possible" options can create a new transaction so the term is loose in its definition. Be careful not to make an incorrect assumption based on any generic terminology you hear.

For the data control scope checkbox while it is either "Isolated" when unchecked or "Shared" when checked. There also appears to be a third option where the checkbox is a blue box or a dash. This isn't a third distinct option; it simply means the "default" which is the "Shared" data control scope.

Finally you might hear both options in combination discussed in an abbreviated way such as an "Isolated New Transaction BTF" or a "Shared No Transaction BTF". Again be careful as these terms have ambiguity, ask questions to get the exact options used if it isn't obvious.

## Working with the Data Control Scope and Data Control Frames

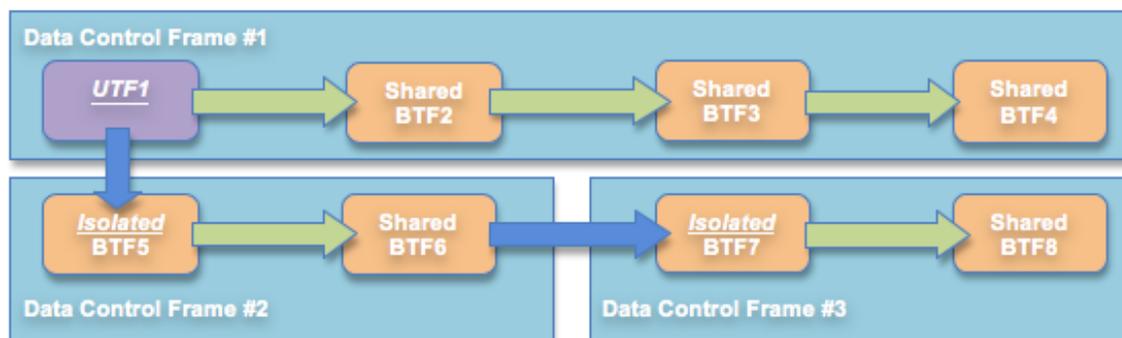
ADF supports several different types of data controls. Ultimately data controls are designed to provide a standard set of APIs to call the underlying business services. For example ADF Business Components as a business service provides operations to query, insert, update and delete data. However depending on the underlying business services implemented in the data control, not all functions consistently have an implementation. As an example, ADF Business Components provide functionality to query, insert, update and delete data from a database, keep track of current row indicators on records sets, and even issue commits and rollbacks on the underlying database transaction. Separately a web service data control may provide functionality to query, insert, update and delete data, but the concept of database transactions will be a foreign one as once the web service is called, it's assumed the transaction is complete.

Moving from the concepts of the data control to that of the data control scope, this refers to the visibility or life of a data control across parts of our application. When a task flow defines a *shared* data control scope, this implies the task flow will attempt to share any instance of a data control (and by implication it's state) with the task flow's caller if the data controls have the same definition, rather than creating a new instance. Alternatively if a task flow defines an *isolated* data

control scope, even if both task flows use the same design time data control definition, at runtime each task flow will have their own instance of the data control. We'll look at some examples of this soon.

The *Data Control Frame* is the magic behind how this works. Essentially each task flow has the *potential* to have it's own data control frame containing a list of the used data controls. A data control frame is created at runtime for your application's unbounded task flow and any isolated data control scoped bounded task flow. However when a bounded task flow specifies a shared data control scope the current task flow uses the data control frame of the caller rather than creating its own, giving the called task flow the chance to share data control instances attached to the frame. Alternatively if the bounded task flow specifies an isolated data control scope, a new frame will be created and a new instance of any data controls used by the bounded task flow will be attached to this new frame.

Figure 4 gives an example of the logical data control frame boundaries that exist when different combinations of data control scope are defined within your application. As seen the unbounded task flow (UTF1) starts a new data control frame that it can then share with future bounded task flows (BTF2, BTF3 and BTF4) as long as they specify a shared data control scope. Alternatively, separate data control frames are created whenever a bounded task flow specifies an isolated data control scope. As demonstrated in Figure 2 BTF5 and BTF7 start new data control frames. However note using an isolated data control scope doesn't preclude any subsequent BTFs sharing the data control frame, so BTF6 with a shared data control scope can join BTF5 in the second data control frame, and BTF8 can also share with BTF7 in the third data control frame:



**Figure 4 - When data control frames are created**

It is worth reminding readers, though it may seem the point of a data control frame is to couple with a single data control, as stated earlier the data control frame is designed to track multiple different data controls and commit and rollback them as a group. In Figure 4's example UTF1 can introduce both an ADF Business Component data control and BTF2 alternatively can introduce a POJO Data Control. However as the two data controls are of different types and

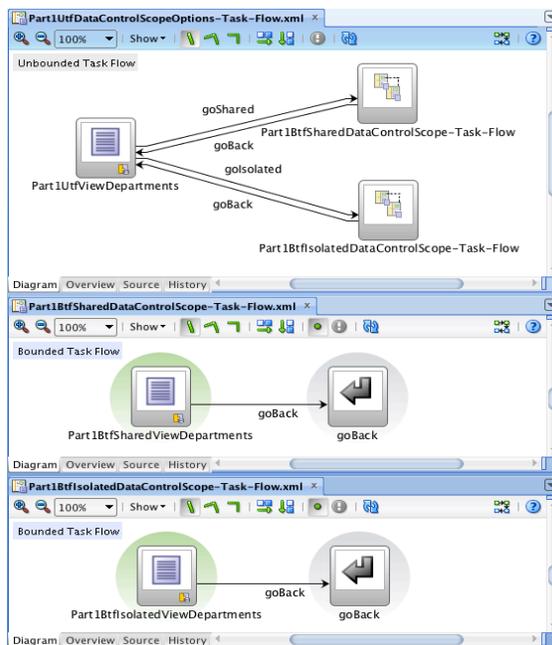
implementations it's not possible for them to share state, as they don't represent the same type of objects. But for purposes of task flows essentially the data control frame is tracking both data controls and will commit and rollback them as a group.

### Data Control Scope Examples

In Figure 5 you can see an unbounded task flow that calls two separate bounded task flows. The first bounded task flow uses the shared data control scope with "<No Controller Transaction>". The second task flow uses isolated data control scope with the same "<No Controller Transaction>" option.

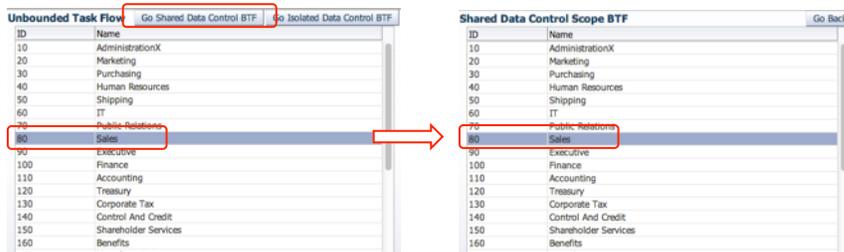
Across the three task flows, each has a view activity that shows a read only ADF table displaying Departments data from a view object in an ADF Business Component data control.

Let's investigate the different options at runtime.



**Figure 5 - An unbounded task flow that calls two separate bounded task flows, one shared, one isolated**

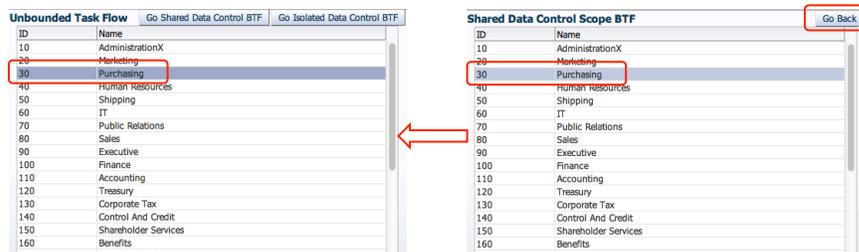
In Figure 6 below on entering the unbounded task flow we select department ID 80. This updates the current row indicator of the view object attached to an ADF Business Component application module data control and is part of the state stored within the data control. We then navigate to the shared data control scope bounded task flow which is re-using the same ADF Business Component data control defined at design time:



**Figure 6 - An unbounded task flow calling a shared data control scope bounded task flow**

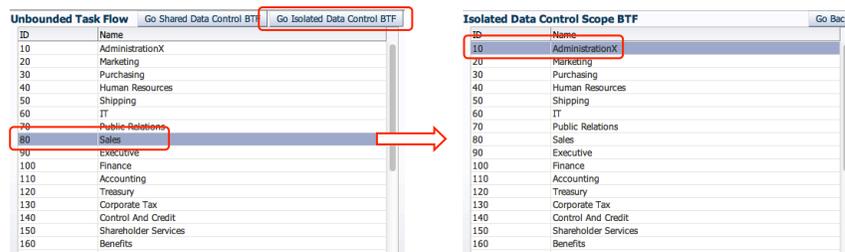
As seen in Figure 6 the current row indicator state is maintained across the task flow calls thanks to the shared data control scope and this occurs as the instance of the data control is shared. This sharing of state for the data control can be shown to go both ways. In Figure 7 while still

using the shared data control scope bounded task flow if we select department ID 30, we can see the selection is reflected in the unbounded task flow page on return:



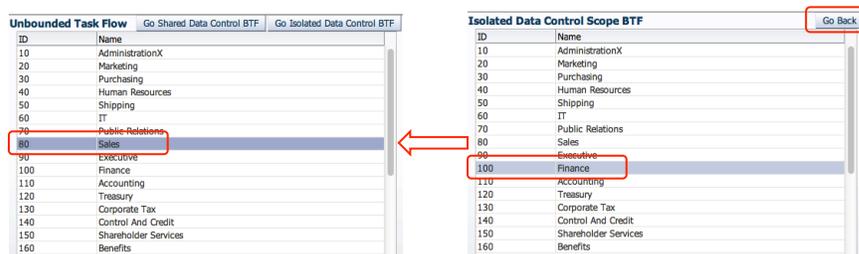
**Figure 7 - Returning to the unbounded task flow from the shared bounded task flow**

In Figure 8 we have the near identical options to the first except when the unbounded task flows calls the bounded task flow, the bounded task flow has an isolated data control scope:



**Figure 8 - An unbounded task flow calling an isolated bounded task flow**

In the previous Figure 8 you can see even though the user selected department ID 80 in the unbounded task flow, the current row indicator state is not shared by the page in the bounded task flow as controlled by the isolated data control scope option because there are now 2 instances of the data control with independent states. And indeed selecting the different department ID 100 in the bounded task flow will not influence the result on returning to the unbounded task flow, it's at department ID 80 as shown in Figure 9:



**Figure 9 - Returning to the unbounded task flow from an isolated task flow**

As explained the data control scope option is pivotal in allowing disparate task flows to share an instance of a data control and its underlying state, or alternatively spawn two separate instances of the data control which will not share state. Each option is viable, but which you select for your

application is dependent on your requirements, do you need to share the data control state or maintain them separately?

In a standalone single application that consists of an unbounded task flow and several bounded task flows, it's an easy case for ADF to share or isolate data controls especially if at design time there's only one data control that exists in your design. Yet ADF allows applications to have multiple data controls, and indeed data controls can be split across separate ADF Libraries comprised of separate task flows that are later compiled into a master application. In this case how does ADF know that the separately defined data controls can be shared, and don't represent two disparate data controls implementing different business services instead?

The problem is solved through the fact that data controls are exposed in the DataBindings.cpx file for each ViewController workspace in each ADF Library. As a developer you need to ensure the same name and type is used for each data control across the two DataBindings.cpx files of the separate workspaces.

For an ADF Business Components derived application this typically requires you to use the same Application Module definition for both the master application and the associated ADF Libraries. In turn as the Application Module name is then used to generate a default data control name (e.g. AppModuleDataControl) it's not recommended to change the default data control name as this makes it not possible to share the data control. If you do decide to use different names the data controls cannot be shared and will have their own instances. In our previous limited examples this would mean *the current row indicators would always be out of sync*.

However as we will see not everything is lost as the task flow transaction options allow disparate task flows to participate in the same transaction.

As a final note when bringing different DataBindings.cpx files together into a composite application via ADF Libraries, the individual DataBindings.cpx files must reside in different Java packages/namespaces otherwise a naming conflict will occur.

## Investigating the Task Flow Transaction Options

Having understood that the data control scope options allow us to share data controls and their state, let's now discuss the transaction options in detail.

As a user interacts with an application, a transaction can encompass the actions of viewing, editing, committing and rolling back data. In the context of ADF the term transaction is aligned with the same concept from the Oracle RDBMS. The work undertaken in a transaction is visible only to the current user session until either a commit or rollback occurs. A commit will result in the transaction results being available to other users, while a rollback undoes all the work and is no longer available to even the original user.

The power of the transaction is that a set of operations can be committed or rolled back together. As an example, imagine a banking application where we're transferring funds between two accounts. First we must subtract money from the first account and then add it to the second account. Without transactions we must commit at the end of each operation. If we successfully subtract money from the first account and commit, but encounter an error on adding the money to the second account, the money is in danger of being lost. A transaction would solve this problem as we only commit at the end of both operations, and if an error occurs, we rollback both operations so no work is done.

Such a simple but powerful concept is extended to ADF task flows through task flow transaction options. Data controls and the underlying business services own and implement the transactional behavior, and ADF task flow transaction options allow you define where the transactions start and finish in terms of application pages and commit and rollback the associated data controls as a group.

For bounded task flows there are essential four values for the transaction options with the following meaning:

- **"Always Begin New Transaction"** - the "Always Begin New Transaction" option configures a bounded task flow to start a new transaction in your application. Typically this option is coupled with the isolated data control scope and a new separate data control frame.

If the "Always Begin New Transaction" option is coupled with a shared data control scope, this means the previous task flow's data controls are shared with the current task flow. As a result the pre-existing data controls may already have an open transaction (i.e. `isTransactionDirty() == true`). If this is true where an existing transaction is detected at runtime the framework will throw "ADFC-00020 + Task flow '<name>' requires a new transaction, but a transaction is already open on the frame".

If a bounded task flow does start with a new transaction, when it wishes to complete it must call a task flow return activity commit or rollback. These call the underlying commit() or rollback() operations on the associated data control frame, essentially committing or rolling back all data controls attached to the frame.

Alternatively programmatically you can call the following code which has the same result:

```
BindingContext bc = BindingContext.getCurrent();
String dcfName = bindingContext.getCurrentDataControlFrame();
DataControlFrame dcf = bc.findDataControlFrame(dcfName);
dcf.commit(); // or dcf.rollback();
```

This is in contrast to using a commit or rollback operation from the Data Control Palette . These only commit or rollback the associated data control, not all data controls for the frame. As a result your task flow would break the intended task flow transaction behaviour and as such should not be used in conjunction with the task flow transaction options (besides <No Controller Transaction> explained later) unless you specifically have a reason to commit a single data control within the frame.

- **"Always Use Existing Transaction"** - a bounded task flow that uses the "Always Use Existing Transaction" option is designed to share the transaction of the previous task flow in your application, it will not start a new transaction. When opened it enforces that it is joining an existing transaction of the previous task flow otherwise it throws "ADFC-00006: Existing transaction is required when calling task flow <task flow name>" at runtime.

How this is enforced is through the fact this option is only available with a shared data control scope implying it always shares the previous task flow's data control frame. You cannot select an isolated data control scope at design time for this option. At runtime using the shared data control frame ADF immediately calls the `DataControlFrame.getOpenTransactionName()` method and expects it to return a none null result as the previous task flow should have created a transaction.

For a bounded task flow using the "Always Use Existing Transaction", on closing it cannot make use of a commit or rollback task flow return activity, at design time such a task flow return activity will be flagged in error. Only a task flow that starts a new transaction can call these. In this case the "Always Use Existing Transaction" option will depend on its caller to finalize the transaction. Instead the "Always Use Existing Transaction" task flow simply calls a task flow return activity with its *End Transaction* property set to "<Default> None". If at runtime an "Always Use Existing Transaction" bounded task flow does use specify a commit or rollback they're just ignored and default to none.

- **"Use Existing Transaction if Possible"** - this option is designed to be the most flexible of the transaction options and it is a combination of the "Always Begin New Transaction" and "Always Use Existing Transaction" options.

If "Use Existing Transaction if Possible" is used with an isolated data control scope, it operates in the same manner as the "Always Begin New Transaction" with an isolated data control scope creating a new data control frame.

If "Use Existing Transaction if Possible" is used with a shared data control scope it's behavior is dependent on if a transaction is open on the shared data control frame of the caller (again determined by calling `getOpenTransactionName()`).

If a transaction is open it has the same behavior of "Always Use Existing Transaction" with a shared data control scope. If a transaction is not open, it behaves the same as "Always Begin New Transaction" and a shared data controls scope.

If you're not sure how the task flow is going to be used, using the "Use Existing Transaction if Possible" option along with the shared data control scope is a good option, making the task flow rather promiscuous in its nature.

In terms of finalizing a "Use Existing Transaction if Possible" bounded task flow, as it's unclear at design time if the task flow will really start a transaction or join one, you should make use of commit and rollback task flow return activities regardless. If the task flow does result in a new transaction the task flow return activities will commit and rollback the data control frame, and if the task flow instead joins a transaction, the return activity commit and rollbacks will simply be ignored.

- **"<No Controller Transaction>"** - of all the task flow transaction options the "<No Controller Transaction>" option is potentially the hardest to understand. Where as the "Use Existing Transaction if Possible" option has a promiscuous nature with the data control frame, the "<No Controller Transaction>" could be considered chaste in its relationship with the data control frame for the current task flow.

A "<No Controller Transaction>" task flow when started:

- Doesn't start a transaction on the data control frame like the "Always Begin New Transaction" option

- Doesn't check or enforce if a transaction is open on the data control frame
- Will not call finalize the data control frame transaction by calling the DataControlFrame commit() or rollback().

Be mindful the established rules still apply for the data control scope though. If an isolated data control scope is specified for the task flow a new data control frame and data controls will be instantiated. For a shared data control scope the previous task flow's data control frame and data controls will be shared.

In some ways while the "<No Controller Transaction>" option is restricted in its interaction with the data control frame, it does have a more liberal relationship with it's relating data controls and their transactions. The other task flow transaction options wish you to commit and rollback at the data control frame level by using task flow return commit and rollback activities. For the "<No Controller Transaction>" option you can call the data control's associated commit and rollback operations.

In terms of finalizing a task flow using "<No Controller Transaction>", it should not use commit or rollback task flow return activities, but rather a task flow return activity with its End Transaction property set to none.

### Task Flow Transaction Option Examples

The intention of the following section is to run through the combination of options to see how two bounded task flows interact with the different transaction and data control options. However there is 64 theoretical combinations making it impossible (and rather tedious) to cover all of them.

Rather than trying to cover the 64 combinations a select set of examples will be included to describe common use cases between two bounded task flows including how to:

- **Completely separate transactions** - isolate two bounded task flow transactions completely so that they can be committed or rolled back separately.
- **Guarantee joined transactions** - guarantee two bounded task flows share a transaction to save connection resources (and fail the task flow transaction join if they can't join).
- **Creating a flexible transaction regime** - create a bounded task flow if called will join the calling bounded task flows transaction if it exists, otherwise create its own.

- **Using the "<No Controller Transaction>" option** - this final bullet point wont talk to a use case, but rather how the "<No Controller Transaction>" option used in combination with the other task flow transaction options creates the most complex combination of options and should be used with caution.

Each use case will be diagrammatically expressed to assist reader's learning on what's happening under the covers. While the diagrams should assist readers to learn the transaction options, to further assist readers each use case will be backed by an online recording showing a real ADF application. Follow the URLs in each scenario to find these online resources.

## Completely Separate Transactions Example

The following section should be read in conjunction with the following recorded demonstration:

<http://bit.ly/ADFTranFundSepTranDemo>

The call center scenario described earlier in this document provides a good example of where an application should support completely separate task flow transactions for the current user. In order to achieve such separate transactions essentially both bounded task flows involved must use isolated data control scopes and the "Always Begin New Transaction" options. The following flow chart in Figure 10 describes how the task flows will interact with these options:

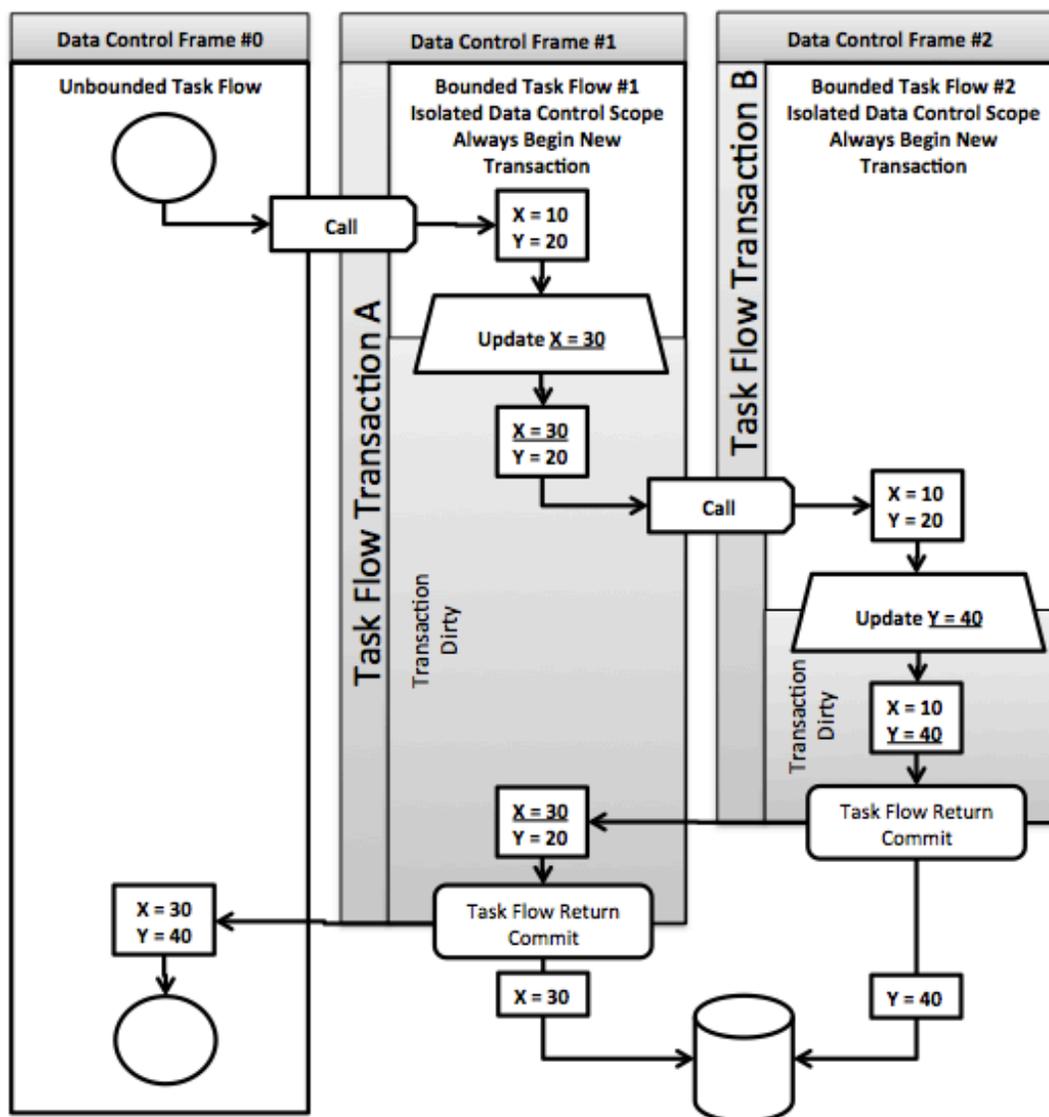


Figure 10 - Complete separate transactions task flow example

First the unbounded task flow starts with its own data control frame #0. A call to bounded task flow #1 with an isolated data control scope and "Always Begin New Transaction" set results in a new separate data control frame #1.

Bounded task flow #1 works on two entirely separate data control fields X and Y initially set to 10 and 20. During the processing of the bounded task flow the user updates X = 30, leaving Y still at 20. At the point of the update task flow transaction A associated with the data control frame #1 becomes dirty as data has been modified in the underlying data control.

Bounded task flow #1 then calls bounded task flow #2. As the 2nd bounded task flow is also using an isolated data control scope with "Always Begin New Transaction", another new separate data control frame #2 is created.

As the task flow transaction of the data control frame #1 has yet to be committed, and the new data control frame #2 has a separate connection to the database and a separate task flow transaction as a result, the values for X and Y are still the original values 10 and 20 which initially came from the database.

Then in bounded task flow #2 the user updates Y = 40 which also makes the underlying task flow transaction B for the data control frame #2 dirty.

Next the bounded task flow #2 finalizes via a task flow commit return activity which saves the changes for Y to the database. As X was not modified ADF has no changes to commit to the database.

On returning to bounded task flow #1, as it has a separate data control frame and task flow transaction the values for X and Y are still at 30 and 20 respectively. A task flow commit return activity saves the changes to X to the database, but as Y is not changed nothing is committed here.

Finally on returning to the unbounded task flow and refreshing the copies of X and Y from the database, both updated values 30 and 40 are returned respectively.

It's worth noting that if the 2nd bounded task flow's data control scope is set to shared, at runtime ADF will throw "ADFC-00020 <task flow name> requires a new transaction, but a transaction is already open on the frame" making this an invalid runtime option if a task flow transaction is already open on the caller's data control frame.

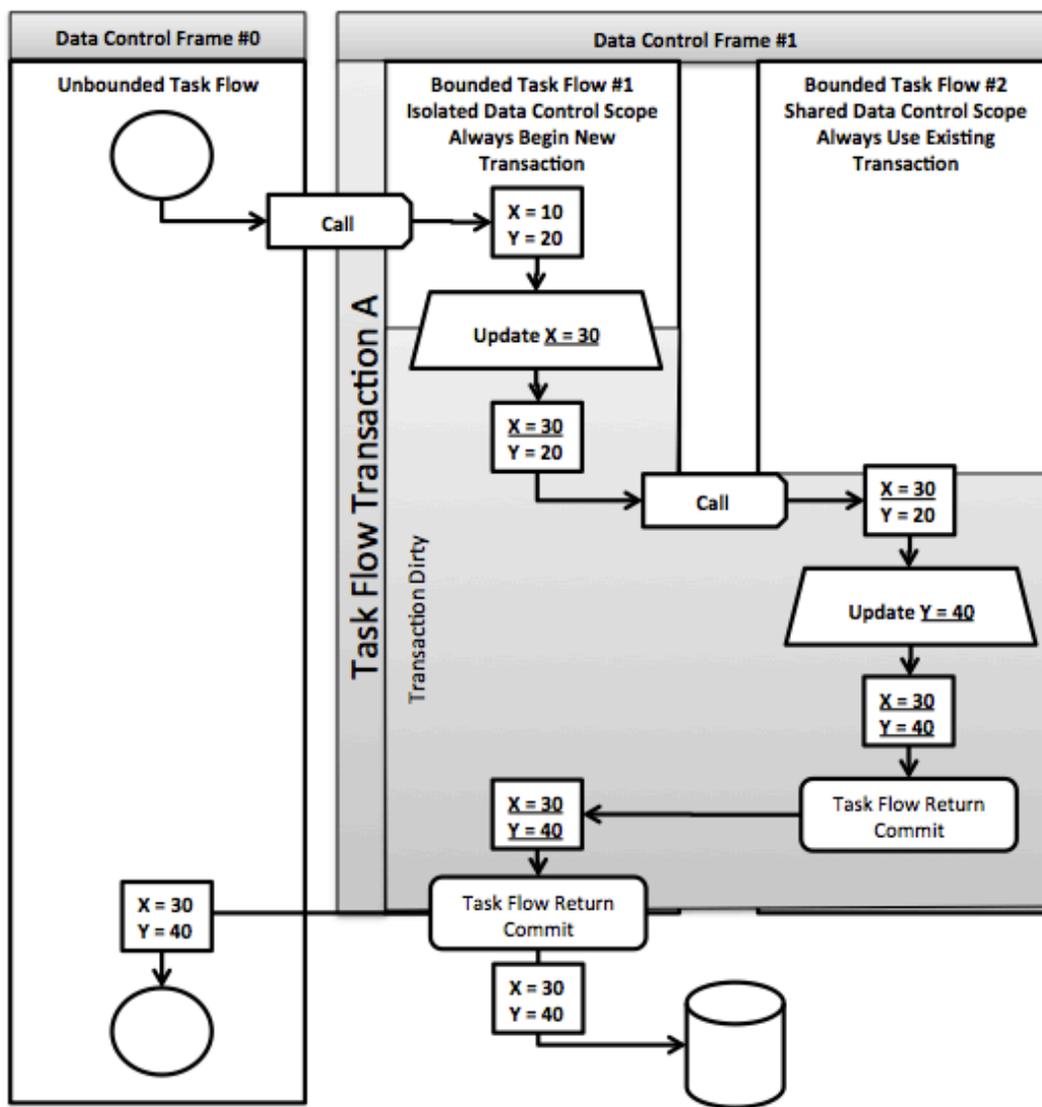
## Guarantee Joined Transactions Example

The following section should be read in conjunction with the following recorded demonstration:

<http://bit.ly/ADFTranFundJoinTranDemo>

Sometimes you will want to ensure that a bounded task flow can borrow the task flow transaction of its calling bounded task flow. For example if you built a bounded task flow to create the line items in an invoice, it wouldn't make sense for the invoice lines to be committed in a separate task flow transaction to that of creating the invoice in another bounded task flow. They can't exist without each other.

In order to guarantee that two bounded task flows join task flow transactions, the first task flow must start a task flow transaction with the "Always Begin New Transaction" option and the second task flow must use a shared data control scope and "Always Use Existing Transaction" options. The following flow chart in Figure 11 describes how the task flows will interact with these options:



**Figure 11 - Guarantee joined transactions task flow example**

First the unbounded task flow starts with its own data control frame #0. A call to bounded task flow #1 with an isolated data control scope and "Always Begin New Transaction" set results in a new separate data control frame #1.

Bounded task flow #1 works on two entirely separate data control fields X and Y initially set to 10 and 20. During the processing of the bounded task flow the user updates  $X = 30$ , leaving Y still at 20. At this point of the update task flow transaction A associated with the data control frame #1 becomes dirty as data has been modified in the underlying data control.

Bounded task flow #1 then calls bounded task flow #2. As the 2nd bounded task flow is sharing data control scope with the "Always Use Existing Transaction" option no new data control frame

is created, rather the 2nd bounded task flow uses the data control frame of the 1st. Because of this changes applied to X in bounded task flow #1 are visible to task flow #2. In turn updates to Y in task flow #2 upon a task flow commit return activity are visible to task flow #1 when it receives control not because of the commit, but rather the shared data control scope.

As a reminder the task flow return commit of the 2nd bounded task flow does not actually commit to the database, commit and rollback task flow return activities are only valid for bounded task flows that start a transaction. Only the commit of the 1st bounded task flow that established the task flow transaction actually finalizes the changes to the database.

The above scenario describes how to setup the correct settings to guarantee both bounded task flows share a task flow transaction. What it doesn't describe though is how the 2nd bounded task flow enforces an open task flow transaction. If you were to change the 1st bounded task flow's transaction option to "<No Controller Transaction>", at runtime when the 1st bounded task flow calls the 2nd with the "Always Use Existing Transaction" option set, at runtime ADF will throw "ADFC-00006: Existing transaction is required when calling task flow <task flow name>" and will stop, enforcing our mandatory task flow transaction option.

#### Creating a Flexible Transaction Regime Example

The previous ADFC-00006 error does show a problem with the previous scenario. What if you want a bounded task flow that ideally will share a task flow transaction with the caller, but if not available creates it's own? The option to use is the "Use Existing Transaction if Possible" on the 2nd bounded task flow with a shared data control scope.

As described earlier in this document in such a scenario where the 2nd bounded task flow detects a task flow transaction on the calling task flow, the "Use Existing Transaction if Possible" option in combination with the shared data control scope will default to the behavior of "Always Use Existing Transaction" with a shared data control scope. As such we won't show this diagrammatically here, simply refer back to the previous Figure 11.

Alternatively if the 2nd bounded task flow is using the "Use Existing Transaction if Possible" option in combination with an isolated data control scope, the 2nd bounded task flow will default it's behavior to "Always Begin New Transaction" with an isolated data control scope regardless what options the 1st bounded task flow uses. See Figure 10 again for the resulting scenario.

### <No Controller Transaction> Example

The following section should be read in conjunction with the following recorded demonstration:

<http://bit.ly/ADFTranFundNoContDemo>

The "<No Controller Transaction>" task flow transaction option in combination with the other task flow transaction options creates the most complex runtime behavior. While ADF does support this combination of options, because of its complexity it is recommended that you don't mix them. Either use the "<No Controller Transaction>" option for all your bounded task flows, or use the other task flow transaction options, but don't mix them unless you're very sure about the expected behavior.

To demonstrate this let's discuss the following scenario in Figure 12 where an isolated data control scope bounded task flow using the "<No Controller Transaction>" option calls a 2nd bounded task flow using a shared data control scope with the "Use Existing Transaction if Possible" option:

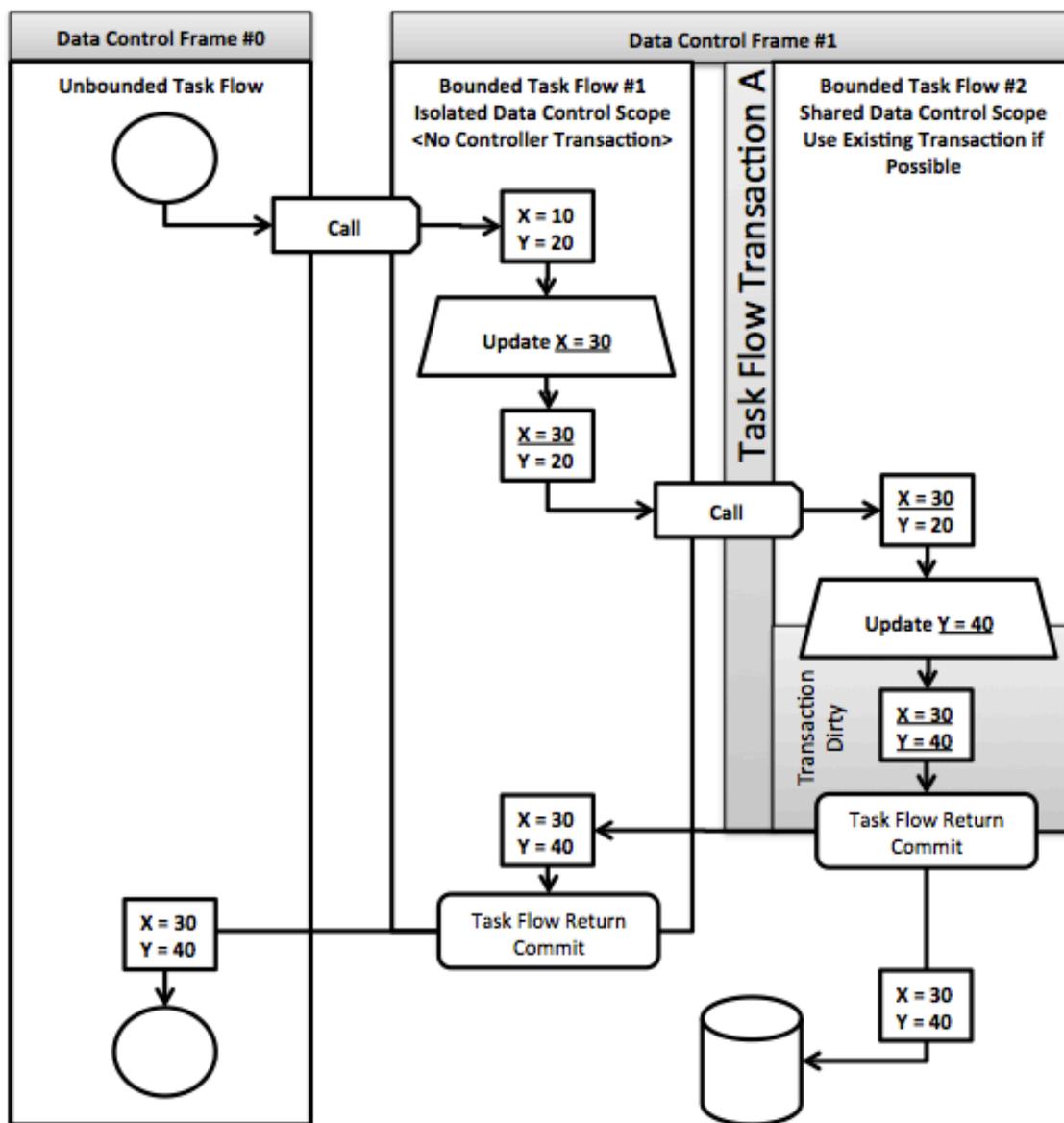


Figure 12 - <No Controller Transaction> task flow example

When the unbounded task flow calls the 1st bounded task flow, because the bounded task flow is using the isolated data control frame option a new data control frame is created. This guarantees a separate transaction to the unbounded task flow. However the bounded task flow will not at any point create and track a task flow transaction as its option explicitly says "<No Controller Transaction>".

Like the previous scenarios in the 2nd bounded task flow  $X$  is updated to 30, while  $Y$  retains its original value.

On calling the 2nd bounded task flow, because it is sharing data control scopes it can now see the X = 30 value and original Y = 20 value. As the "Use Existing Transaction if Possible" option is set, upon updating Y = 40 the 2nd bounded task flow starts a task flow transaction and it now believes it is the custodian of the task flow transaction.

Because the 2nd bounded task flow controls the task flow transaction, of special note at the conclusion of the 2nd bounded task flow, the task flow commit return activity sends the update for both X and Y to the database. This is completely different behavior to that described in Figure 11.

As such on returning to the 1st bounded task flow and executing the task flow commit return activity, nothing is sent to the database as bounded task flows without task flow transactions cannot commit or rollback the associated data control frame.

Regardless the end result when returning to the unbounded task flow it can still see the changes as they were indeed committed to the database.

If for any reason the 1st bounded task flow using the <No Controller Transaction> option further edited the data and needed to commit the changes, it cannot make use of a task flow commit return activity and instead should commit the associated data control using the commit binding from the Data Control Palette.

## Prematurely Terminated Task Flows

Typically a bounded task flow undertaking a task flow transaction should use the task flow return commit and rollback options. However as bounded task flows can be embedded in regions of a parent page or page fragment (potentially in a unbounded task flow or bounded task flow), the parent can for whatever reason refresh the embedded bounded task flow in a region, navigate away from the View Activity containing the bounded task flow, or itself be terminated by its caller. The effect of the parent doing this is the framework will prematurely terminate the child bounded task flow and it will also make a decision on how to terminate the task flow's transaction.

If the task flow has joined the transaction of the caller's data control frame, the framework will simply return the control to the calling task flow. The assumption being the parent should manually take care of tidying up the transaction in this case.

However if the called task flow is the initiator of the data control frame and transaction rather than the parent, essentially their transactions are separate, the parent cannot operate and tidy up the child's transaction in this case. So on behalf of the called task flow the framework will automatically rollback the called task flow's transaction.

If the called task flow is isolated and as a result has its own data control instance, the rollback undoes any work on the data control which is specific to the called task flow, and the undo work is encapsulated and does not effect anything else. However if the task flow is sharing data control scope where an instance of a data control is shared between the calling and called task flow, developers can experience a nasty little side effect if they're using ADF Business Components. Because the rollback issued by the called task flow is against a shared AppModuleDataControl, the rollback will go across task flow boundaries regardless that their task flow transactions are separated.

This may seem confusing but it does highlight the fact stated earlier in this document that the task flow transaction is an abstraction that sits above the transactions controlled by the data controls. Ultimately the data controls are what implement the real transactions, in this example a transaction controlled by an ADF Business Component application module. And as our task flows have shared data controls we see this spill over affect of the rollback.

The solution to avoid this scenario is to not share data controls.

An example scenario can be found in the following blog <http://bit.ly/ICYZUp>.

## ADF Business Component Transaction Use Cases

Some readers of this paper may be aware of three blog entries by Chris Muir that describes the changing behavior of task flow transactions and ADF Business Component's Application Modules between ADF 11gR1 and 11gR2 releases. The three blog entries in chronological order are available as follows:

- <http://bit.ly/MOVC1S>
- <http://bit.ly/KOv5NY>
- <http://bit.ly/KBCx0P>

The first two blog entries articulate how ADF Business Components Application Modules in 11gR1 are "automagically" nested for chained task flows sharing the same task flow transaction. While this approach certainly works in most use cases it has a limitation that only the initiating Application Module's configurations were used to configure the "automagically" nested Application Module group. If the developer wanted to tune each individual Application Module, or to connect to a different database, it wasn't possible as the first Application Module's configurations were used to configure the overall behavior.

JDeveloper 11gR2 solves this problem by totally separating the ADF BC Application Modules at runtime, they all run as root Application Modules. This now allows each Application Module at runtime to run with its own configurations and be tuned separately.

However note that all the Application Modules within the task flow transaction will use the data source of the primary Application Module, effectively the data sources of any secondary Application Modules that join the task flow transaction are ignored. This presents a catch if the secondary Application Modules are designed to work with different database schemas, they will now throw error at runtime when they can't access the expected database components. As such with all Application Modules in an application using ADF Business Components it is necessary that they are designed to work with the same data source and database schema.

## Recommendations

When using the transaction and data control scope options Oracle suggests you follow these recommendations:

On the transaction options:

- The "<No Controller Transaction>" option does not imply your task flow cannot participate in a database transaction. Rather it simply doesn't participate in the data control frame transaction. As such don't ignore this option because of its slightly misleading name.
- If your application only has one data control, and as implication you have no need to group data control commits, the "<No Controller Transaction>" option is a viable choice by itself. It even supports multiple transactions simply by using an isolated data control scope.
- If a standalone bounded task flow could be potentially used as part of a number of steps in a greater logical transaction, for maximum reuse it should specify a shared data control scope with the "Use Existing Transaction if Possible" transaction option.

On the data control scope options:

- Isolated Data Control Scope tasks flows cannot share transactions, Shared Data Control Scope task flows can. If your application uses ADF Business Components, be wary of using multiple transactions in your application as every transaction is a connection for, and typically where an application gives a user one connection for each request, multiple transactions means multiple connections per user limiting your application's scalability. A potential mid solution is to allow use additional transactions and connections for task flows that you know will have a short life.

On committing and rolling back a task flow:

- If a task flow starts a data control frame transaction, it must conclude it by using a task flow return activity commit or rollback. Do not use the commit and rollback operations available for the data controls via the Data Control Palette.
- A "<No Controller Transaction>" task flow should not programmatically commit or rollback the data control frame it is participating in. Instead it should use the commit and rollback bindings of its associated data control.

Gotchas to watch out for:

- If separate task flows initiate separate transactions on the same ADF BC components, it's possible if the user in the separate task flows updates the same record, for the user to row-lock themselves.

## Summary

Oracle's Application Development Framework (ADF) provides ADF developers a level of power with task flows to spawn multiple transactions and share data controls. However developers need not only be aware of the features, but how the options can be used in combination to create efficient use of the features available.

---

### RELATED DOCUMENTATION

---

	Oracle® Fusion Middleware Fusion Developer's Guide for Oracle Application Development Framework 11g Release 1 (11.1.1.6.0)
	- Section 14 Getting Started With Task Flows <a href="http://bit.ly/adfdevguide111160s14">http://bit.ly/adfdevguide111160s14</a>
	- Section 16.4 Sharing Data Control Instances <a href="http://bit.ly/adfdevguide111160s164">http://bit.ly/adfdevguide111160s164</a>
	- Section 18 Introduction to Complex Task Flows <a href="http://bit.ly/adfdevguide111160s18">http://bit.ly/adfdevguide111160s18</a>
	- Section 18.3 Managing Transactions <a href="http://bit.ly/adfdevguide111160s183">http://bit.ly/adfdevguide111160s183</a>

## Credit

The author Chris Muir would like to thank Luc Bors and Chad Thompson for participating in the review of this document.