3rd June 2013

Workflows to subscribe to business events

A non-intrusive and highly specific method to launch processes after a certain application event occurs (compared to modifying seeded code, using Alert or worse, triggers) is to leverage the Business Events System (BES) if a relevant event is raised that can serve our needs.

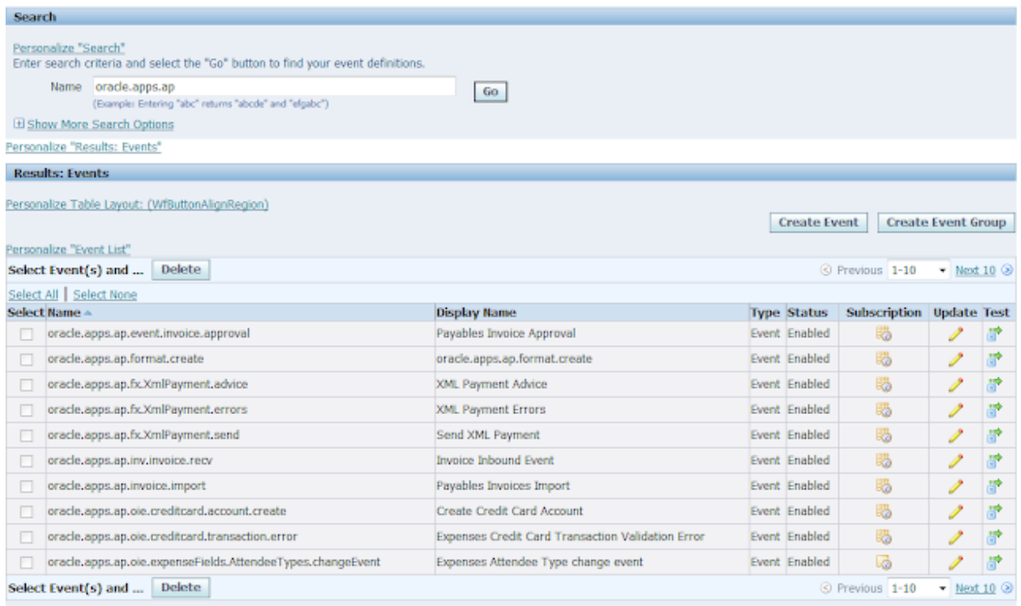
Here I'll show how to create a workflow subscription to a business event as well as document two possible scenarios on retrieving event specific information that will allow to implement whatever custom logic is required.

The difference on the scenarios is regarding what information is made available for the event when launched and how to retrieve it from within workflow. Both the subscription and information retrieving are fairly simple to implement, yet doing it the first time might frustrate you as there is no straightforward documentation and I find it lacking in the case of business events.

Note, this was done on Release 12, just look for the equivalent on 11i if something differs.

First things first, identifying the potential business event

Business events are usually named as oracle.apps.module.xxx.yyy.zzz and they can be queried from within "Workflow Administrator Web Applications" responsibility under "Administrator Workflow"/"Business Events" menu.

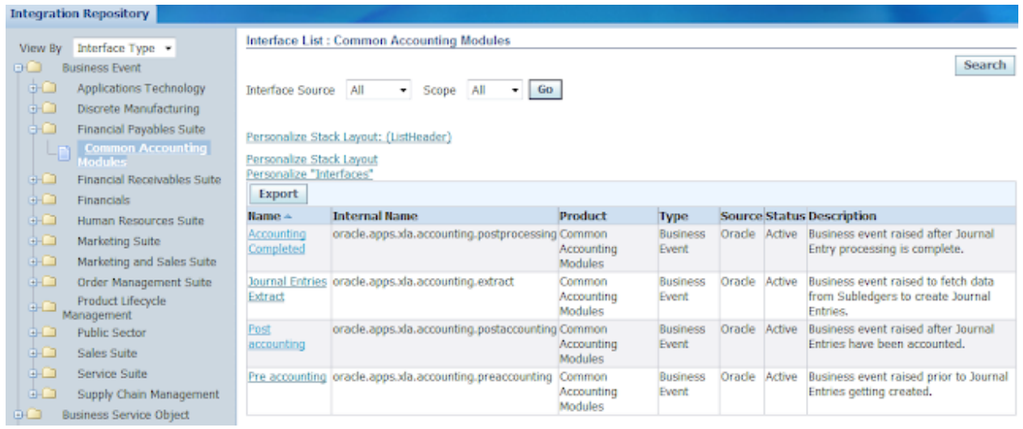


[http://1.bp.blogspot.com/-6UmUBYUUkYk/UavxQskJ0CI/AAAAAAAAAGw/PuBws5MAAtQ/s1600/bes_search.png]
Image 1. Search Business Events

If you have an idea on which module the event should be registered, just do a query as oracle.apps.module and click go as shown on the image above. The name and display name will give you a brief idea of at what moment the event is launched and allow you to look for more details in the code that fires it.

Another way to look for available events and hopefully getting a little bit of more info is to use the Integration Repository (iRep) which can be accessed by the the "Integrated SOA Gateway" Responsibility. On it, select "Interface Type" on the "View By" LOV and then navigate the tree by product family and module. Don't expect thorough information, though as

likely only a short description is what you'll find and nothing else.



[http://3.bp.blogspot.com/-UsgScryRYOc/Uavxt3qvG6I/AAAAAAAAAG4/94afJfqJqk4/s1600/bes_irep.png]
Image 2. Business Events on the Integration Repository

How to find how it is launched?

This is not strictly necessary but can provide additional details beyond the simplistic descriptions available as shown before.

As we have the name and the firing is done via an API, we can look for the name on the database code and luckily we'll find a match. If not, we can do so on other type of code such as forms, reports and java if we are inclined to to do so.

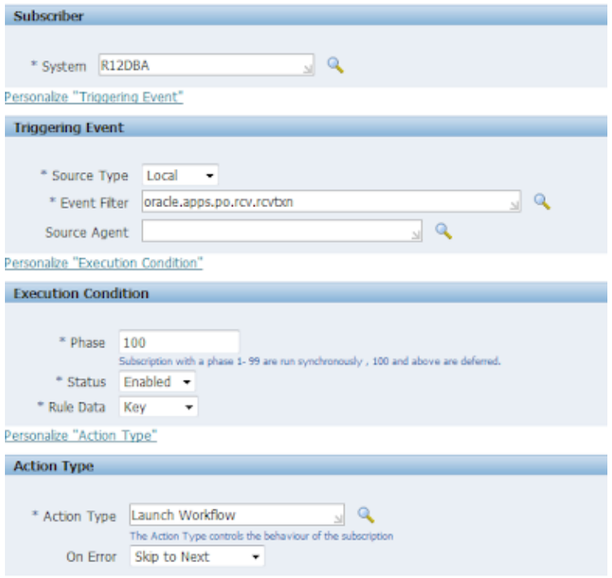
For the second example that I'll show, I was interested on running some logic after the AP Invoice Import process successfully loaded invoices to the application. As such, I was pretty sure the "oracle.apps.ap.invoice.import" was the event to use but just wanted to double check if fired after the payment schedules were created and also that it provided enough information to be able to identify the invoices created by it. Running a query on the database I identified the AP_IMPORT_INVOICES_PKG package, which happens to be called from the report associated with the "Payables Invoice Import" program.

The launching code is the following (edited for readability). The useful information for me here was knowing that the event message payload captured the request id which would prove good enough to link it to the newly defined invoices and also to peek at the parameters specified when submitting the import program in order to implement logic to proceed only if certain conditions were met i.e. a particular AP Source was specified. Notice also that the event key is just a sequence number, not linked to any particular data.

```
1  --7567527 PL/SQL Block for Enhancement to raise Business events after invoices are imported
2  BEGIN
3    l_parameter_list := wf_parameter_list_t(
4        wf_parameter_t(
5            'REQUEST_ID'
6          , to_char(AP_IMPORT_INVOICES_PKG.g_conc_request_id)));
7    SELECT to_char(AP_INV_IMPORT_EVENT_S.nextval)
8      INTO l_event_key
9      FROM dual;
10   wf_event.raise(
11       p_event_name => l_event_name
12     , p_event_key  => l_event_key
13     , p_parameters => l_parameter_list );
```

Subscribing to an event

The prerequisite is to have the workflow item and a runnable process for it defined on the database. On the event listing as show on Image 1, click on the subscriptions icon and then on the Create Subscription button on the next page.



[http://2.bp.blogspot.com/-VneHvexsXJo/UawJCFlyhfI/AAAAAAAAAHk/Zfro8xXCRS0/s1600/bes_subscribe1.png]

Image 3. BES Subscription Page 1

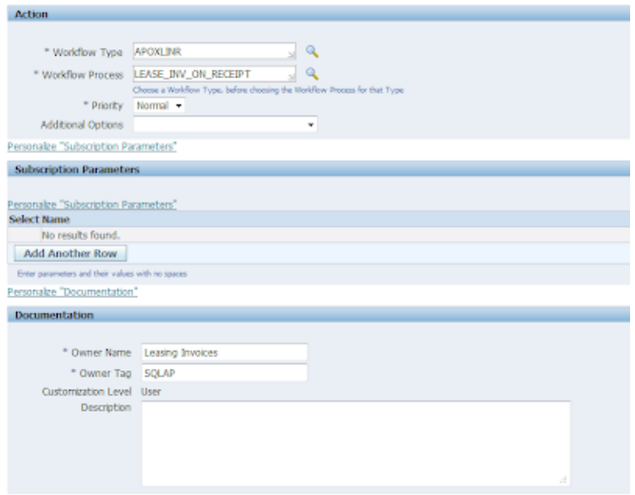
On the subscriber region, select the appropriate one, which should usually be the only one available to pick.

Leave default vales on the Triggering Event, as this relates to the event we are subscribing to.

For the execution condition, leave the Phase as 100, which means the workflow will be executed as deferred by the background engine. This is important not to affect the performance of the underlying code raising the event. For the Rule Data, this depends on what information we need; key means we are only interested on the event key, messages indicates that we are also interested on the message payload associated with the event being raised which will provide additional information for it in the form of parameters. For the first example shown below, use key, for the second one it must be Event.

On the action type, select "Launch Workflow" and then "Skip to Next". Skip to next means keep raising events even if one particular instance fails; the other option is to stop and rollback, wich could mess with standard functionality.

Click Next



[http://2.bp.blogspot.com/-rnjRVdqkn5w/UawLIdrk2uI/AAAAAAAAAH0/jxsKbq-MNQI/s1600/bes_subscribe2.png]

Image 3. BES Subscription Event Page 2.

Pick the item type and runnable process on the action regions.

For the documentation region, specify a descriptive name for the Owner Name and an application short name for the Owner Tag. The last one is validated even if no LOV is associated to it. In the example you can see SQLAP was specified as just using AP would have failed.

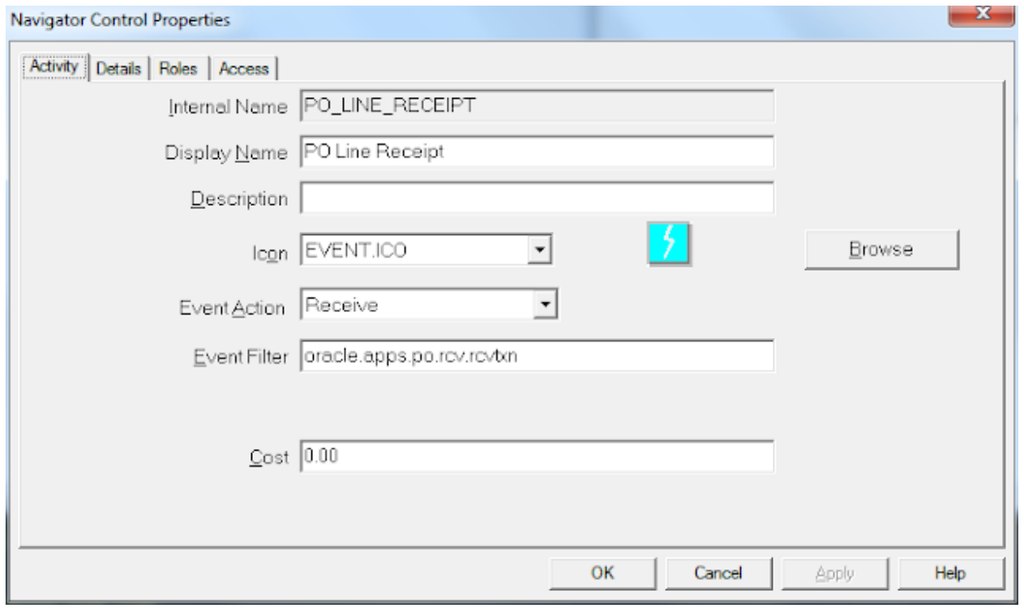
Select apply.

The Basic Case

As the event key also becomes the workflow item key, the simpler case is when the event key provides an id to identify a record from where we can extract information required for the process. This is easier because other than listening from the specific event, we then just have to extract the id from the key and not have to look onto the event message for it.

With that said, I'll use the Receive transaction event (oracle.apps.po.rcv.rcvtxn) as an example. The event key has two values separated by a hyphen, the second part being the value of the shipment_header_id from the rcv_shipment_headers table, i.e. 12345-98765.

On workflow builder we first have to create a new event either via the Events branch on the object navigator or selecting "New event" on the right click menu on the process designer window. Select "Receive" for the Event Action and the event name as the Event Filter.



[http://1.bp.blogspot.com/-Jv4BGDQNiUI/Uav1xzmYc2I/AAAAAAAAAHE/ZAD7cD8yiBQ/s1600/wf_event.png]
Image 4. Define Event.

Once defined, we must select "Start" for the Start/End LOV on the Node tab to indicate this will be the process start activity. This means that the entry point for this workflow will be a business event firing, only if such event matches the specified filter. The filter is not mandatory nor validated against an LOV when specified, meaning that partial or null filters can be typed which are useful if wanting to listen to a group of events (partial filter, i.e. oracle.apps.po.rcv) or to all of them (no filter at all). Beware that no % is used for partial filters.

Once we have the event defined as the starting node, we can continue diagraming the particular logic needed. In order to retrieve the id that allows us to identify the transaction, we create PL/SQL function and retrieve the id from the itemkey on the stored procedure:

```
1  b_sh_id => regexp_substr( itemkey, '([[:digit:]])*')) ;
2
```

From then on, it is just business as usual.

The not so basic Case

This builds on top of the basic case. As seen above in the code fragment for the invoice import event, the event key is just a sequence which does not relate to anything on the database. Here, the valuable information is on the event message as a parameter, in particular the REQUEST_ID.

We start by defining the event as shown on image 4, for this case the filter being oracle.apps.ap.invoice.import.

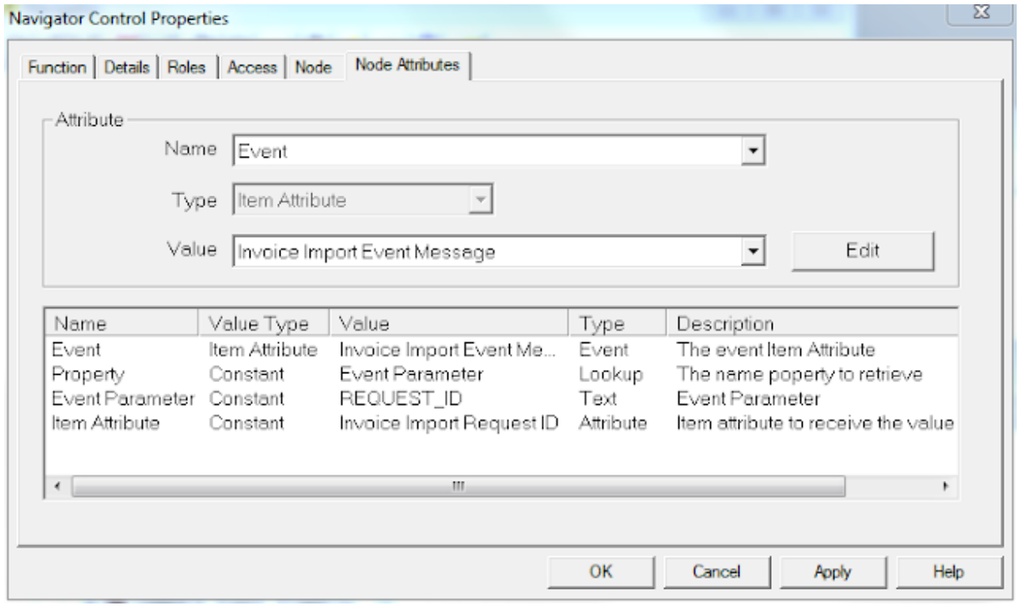
Now we need to capture the event message. We do this by defining an attribute of type event. Other than assigning a name to it i.e. "Invoice Import Event Message", no more particular properties are required. Remember to set subscription Rule Data definition to be message (Image 3).

Then we create another normal attribute to capture the REQUEST_ID parameter value, we set its type to number to match the data type of the value to store. I named it as "Invoice Import Request ID".

Back on the workflow diagram, reopen the starting event node properties and on the event details tab, specify the newly

defined event attribute ("Invoice Import Event Message") as the Event Message; this will capture the message on an attribute and allow to further use it to retrieve provided information.

In order to get the REQUEST_ID parameter we will leverage an standard function as the activity to follow the starting event node. Drag the "Get Event Property" function from the Standard Workflow and set the following values on the Node Attributes tab, based on the definitions we have done so far.



[http://3.bp.blogspot.com/-FG9ZUdLFhLs/Uav9Dtbb6iI/AAAAAAAAAHU/gqAuYPhpvzM/s1600/get_event_property.png]

Event: Invoice Import Event Message
This is the attribute defined of type event and the one holding the event message when it launches the workflow process.

Property: Event Parameter
We will be retrieving a parameter value from the event message.

Event Parameter: REQUEST_ID
Name of the parameter we are interested in.

Item Attribute: Invoice Import Request ID
Name of the attribute where the parameter value will be stored.

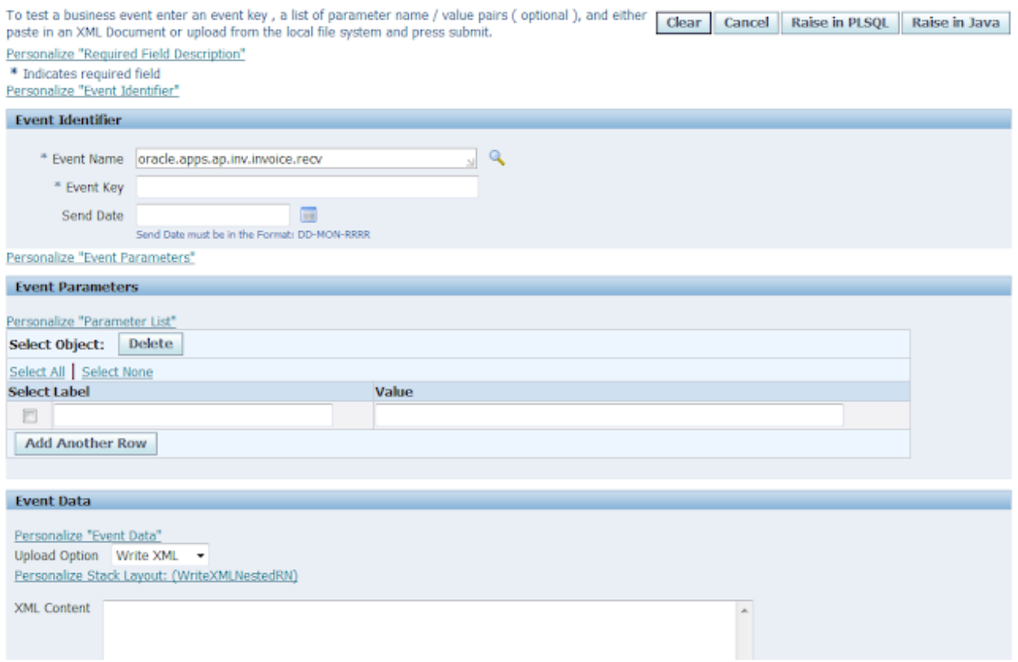
Once this is done, we can reference the newly stored value in the attribute defined for the request id and proceed with the required logic.

Testing the event

In order not to depend on users putting in transactions and given that initially we will be tweaking the process, we require a way to submit events as we see fit.

The easiest way is to use the Test functionality from the BES search, shown all the way to the right on the table in Image

1. Here we just specify a key for the event and click submit, which will bring up the following page.



[http://4.bp.blogspot.com/-2hSFskyKnds/UawRJMdcG6I/AAAAAAAAAIE/0_jyNZ8ctM4/s1600/bes_test.png]

Here, just specify an event key that suits your needs for testing. In the simple case, one that will provide the workflow process with data to carry out the defined logic. Once submitted, the subscribed workflow process will be launched and can be monitored.

For the second case which uses parameters, we can specify name value pairs on the "Event Parameters" region before submitting. In the example, that would be REQUEST ID as the label and a number on the value.

Another option is to programmatically submit the event, similar to what we saw on the code fragment above:

```
declare
  l_parameter_list.wf_parameter_list_t ;
begin
  l_parameter_list := wf_parameter_list_t(
      wf_parameter_t(
          'REQUEST_ID'
        , to_char(123456))); -- test value
  wf_event.raise(
      p_event_name => 'oracle.apps.ap.invoice.import'
    , p_event_key  => 'TEST001'
    , p_parameters => l_parameter_list );
end ;
/

```

Posted 3rd June 2013 by Cuauhtemoc Amox

Labels: api, cemli, custom, oracle applications, pl/sql, workflow