IBM

Business Process with BPEL4WS: Learning BPEL4WS, Part 7

Adding correlation and fault handling to a process

Level: Introductory

Rania Khalaf (rkhalaf@watson.ibm.com), Software Engineer, IBM TJ Watson Research Center
William Nagy (nagy@watson.ibm.com), Software Engineer, IBM TJ Watson Research Center

22 Apr 2003

In the previous article we examined correlation and fault handling in BPEL4WS. Now, we are going to extend the simple BPEL4WS process that we have been working with in the previous articles by adding the ability to communicate with a pre-existing process instance and to capture faults which may occur during its execution.

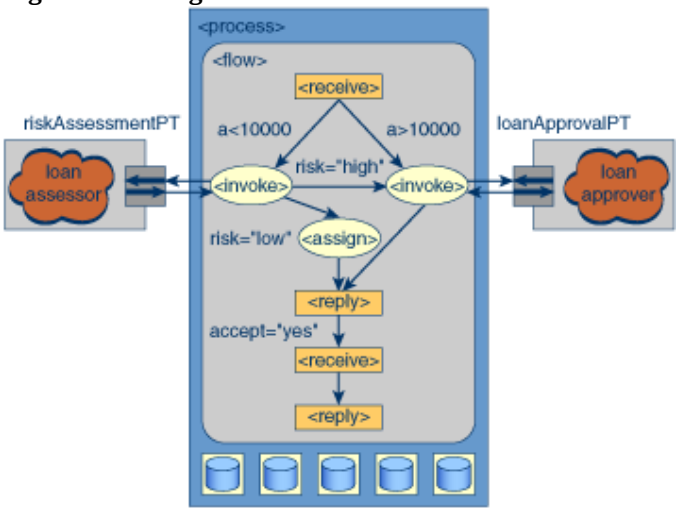
Introduction

Having explained correlation and fault handling in BPEL4WS in the previous column of this series, we now add these capabilities to the loan approval sample we have been working with to illustrate their use. In the scenario shown below, if a client is approved for a loan he may send a request to actually obtain that loan. Correlation is used to make sure that the second request goes to the correct process instance. Fault handling will be added to catch an explicitly thrown error that signals that the client is trying to obtain a loan for an amount larger than the one that was approved. If such a fault is caught, a reply is sent back describing the problem. We also show the effect of error handling with nested scopes and its effects on the links that leave a scope that has handled its error, as well as one that was not able to do so and had to rethrow the fault to its enclosing scope.

Obtaining the loan: adding correlation

To demonstrate correlation (and in the next section, fault handling) we're going to extend our loan processing example by adding the ability for the client to submit a request to actually obtain the loan that they have been approved for. We'll add a `<receive>` activity to accept the request to obtain the loan, and a `<reply>` activity to acknowledge it has been obtained. The resulting flow is shown in Figure 1.

Figure 1. Adding activities to demonstrate correlation



Now we need a way to make sure that the request to obtain the loan arrives at the same process instance that provided the initial approval -- this is where correlation comes into play. In order to correlate messages, we need to define a

<correlationSet>. Definition of the <correlationSet> requires that we know what WSDL properties we want to correlate on, so the first thing that we need to do is to define the <property>ies. In our case, we are going to assume that the first and last names of the requestor will give us a unique key for a process instance, and so we are going to create two <property>ies based on that assumption, applicantFirstName and applicantLastName. Next, we need to define the <propertyAlias>es for those <property>ies so that we can extract the necessary values from the messages that have been transmitted. We need to initialize the correlation set before the second <receive> activity, and we could conceivably do so in either one of the <invoke> activities, in the first <receive> activity, or in the first <reply> activity. The approval message does not have the correct information in it, so we can rule out the first <reply>. We don't always make it to both <invoke> activities, so the best place to initialize the correlation set will be on the first <receive>. As it happens, both <receive>s have the same WSDL message as input, so we will only need to define two <propertyAlias>es, one for each of the <property>ies. This is shown in Listing 1.

Listing 1. Use of propertyAlias

```
            <bpws:property name="applicantFirstName" type="xsd:string"/>
<bpws:property name="applicantLastName" type="xsd:string"/>

<bpws:propertyAlias propertyName="lns:applicantFirstName"
   messageType=
   "loandef:creditInformationMessage" part="firstName" query=
   "/firstName"/>

<bpws:propertyAlias propertyName="lns:applicantLastName"
   messageType=
   "loandef:creditInformationMessage" part="name" query="/name"/>

...

<portType name="loanApprovalPT">
  <operation name="obtain">
    <input message="loandef:creditInformationMessage"/>
    <output message="apns:approvalMessage"/>
  </operation>
</portType>

<slnk:serviceLinkType name="loanApprovalLinkType">
  <slnk:role name="approver">
    <portType name="apns:loanApprovalPT"/>
    <portType name="lns:loanApprovalPT"/>
  </slnk:role>
</slnk:serviceLinkType>
```

Once we have defined the <property>ies in the WSDL file, we can define the <correlationSet> in the BPEL document and set up its initialization and usage. We are going to add a new <correlationSet> called loanIdentifier, add the <correlationSet> to the first <receive> and place the *initiation* attribute on it with a value of 'yes' to indicate that we want the correlation set to be initialized at that point, and also add the <correlationSet> to the second <receive> so that it will be used to determine the correct instance to route the message to. This is shown in Listing 2.

Listing 2. correlationSet

```
<containers>
  <container name="request" messageType="loandef:creditInformationMessage"/>
  <container name=
   "acceptanceRequest" messageType="loandef:creditInformationMessage"/>
  <container name="riskAssessment" messageType="asns:riskAssessmentMessage"/>
  <container name="approvalInfo" messageType="apns:approvalMessage"/>
  <container name="error" messageType="loandef:loanRequestErrorMessage"/>
</containers>

<correlationSets>
  <correlationSet name="loanIdentifier" properties=
  "lns:applicantFirstName lns:applicantLastName"/>
```

```
</correlationSets>

...

<reply name="initial-reply" partner="customer"
  portType="apns:loanApprovalPT" operation="approve" container="approvalInfo">
  <target linkName="setMessage-to-reply"/>
  <target linkName="approval-to-initial-reply"/>
  <source linkName="reply-to-receive"
     transitionCondition=
     "bpws:getContainerData('approvalInfo', 'accept')='yes'"/>
</reply>

<receive name="acceptance-receive"
  partner="customer" portType=
  "lns:loanApprovalPT" operation=
  "obtain" container="acceptanceRequest">
  <target linkName="reply-to-receive"/>
  <source linkName="receive-to-grant"/>
  <correlations>
    <correlation set="loanIdentifier"/>
  </correlations>
</receive>

<reply name="grant-reply" partner="customer"
  portType="lns:loanApprovalPT" operation=
  "obtain" container="approvalInfo">
  <target linkName="receive-to-grant"/>
</reply>
```

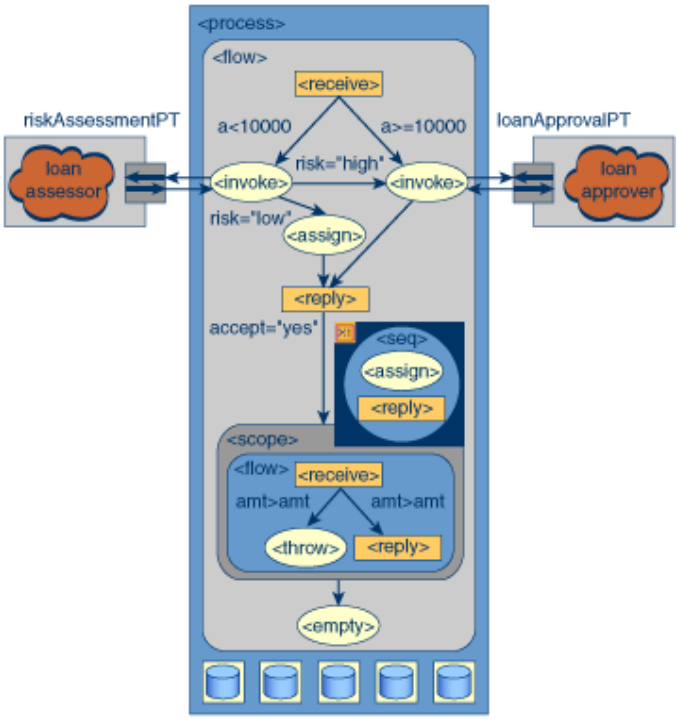
That's all that we need to do to enable correlation in our example. Now, whenever a message arrives for the second `<receive>`, the `applicantFirstName` and `applicantLastName` property values will be extracted and compared against the values stored within all of the instances of the process, and the message will be routed to the instance which has the matching values.

Error notification: using fault handlers

In order to show an interesting fault handling example, we will introduce a nested scope with an outgoing link. This scope will wrap the receive/reply/throw activities added in the previous section. Since a scope may only contain one activity, we will first wrap the three activities in a `<flow>`. The second addition is a link from this scope to an `<empty>` activity whose purpose is to show how the links are affected depending on whether or not a fault is caught and what scope level it was caught at.

Finally, we add a fault handler on our new scope. The idea is that if the customer tries to obtain a loan that is higher than the one they have been approved for, they should be notified. This can be achieved using a fault handler with an `<assign>` that puts an error message in a container, and a `<reply>` that sends that message back to the customer. The two will be enclosed in a `<sequence>` so that they happen one after the other. The resulting flow is shown in Figure 2.

Figure 2. Adding a fault handler on a nested scope



This is added to the BPEL file in Listing 3, with the new code in blue, and the existing receive/reply/throw in black. You will also need to add the last link (*scope-to-empty*) definition along with the others. Note that in the fault handler, we catch the same fault as that declared in the throw activity. A fault container is specified, which holds the error message. However, in the WSDL, the error message consists of an integer which contains the error code. We would rather send the customer a meaningful string. Therefore, we can reuse the `approvalInfo` container as shown in the `<assign>` below.

Listing 3. BPEL file

```
            <scope name="new-scope">
  <source linkName="scope-to-empty"/>
  <faultHandlers>
      <catch faultName="lns:loanProcessFault" faultContainer="error">
      <sequence name="fault-sequence">
        <assign>
           <copy>
            <from expression="'invalid request: amount too high'"/>
            <to container="approvalInfo" part="accept"/>
           </copy>
        </assign>
      <reply partner="customer" portType="lns:loanApprovalPT"
        operation="obtain" container="approvalInfo" faultName=
        "lns:loanProcessFault"/>
     </sequence>
    </catch>
  </faultHandlers>

  <flow name="inner-flow">
    <receive name="acceptance-receive" partner="customer"
      portType="lns:loanApprovalPT" operation="obtain" container=
      "acceptanceRequest"> ... </receive>
    <reply name="grant-reply" partner="customer"
      portType="lns:loanApprovalPT" operation="obtain" container=
      "approvalInfo"> ... </reply>
    <throw name="grant-failure" faultName="lns:loanProcessFault"> ... </throw>
  </flow>
</scope>

<empty name="the-last-one">
   <target linkName="scope-to-empty"/>
</empty>
```

Now let's consider what should take place once this runs. Say a client asks for a $200 loan and gets the approval. After the approval is sent the "new-scope" starts, makes its fault handler ready, and starts the inner flow. The receive activates and the process waits for the client to obtain the loan. The client then asks to obtain the loan, but puts in $400 for the amount. The link from the `<receive>` to the `<reply>` goes negative and, because of the suppression of join failures the `<reply>` quietly disables. The link from the `<receive>` to the `<throw>` goes positive and the fault is thrown.

Once the fault is thrown, all activities in the scope have to stop. In this case none are running anyway. Then the inner scope gets the fault and checks to see whether it has a handler for it. It does. The handler activates, and runs the short sequence that sends the client the reply "invalid request: amount too high." The handler completes, and the scope completes *normally:* the link from the scope to the empty activity evaluates to true and the empty activity runs. The process completes.

Consider for a moment the case that the fault handler was actually at the process level instead of on the inner scope. What would happen then? The inner scope would get the fault, stop its activities, realize it can't handle it, and rethrow it up to its parent scope which in this case is the actual process. Now the fault is coming out of the inner scope, and its link to the empty activity goes negative. The `<empty>` does not run. The process would catch the fault, send the reply as above and the process would end normally.

Running the processes

As with our earlier articles, if you want to run the processes you'll need to download and install the BPWS4J engine available from alphaWorks (see Resources). See also the Resources section to download the zip file with the code examples described below.

Because the samples use a modified client to access the additional functionality, you'll need to compile that before you can invoke the processes. Place the client source (Customer.java) in a directory called samples/dwsample, make sure that the soap.jar from the BPWS4J release's reqlib directory is in your classpath, and compile away. You can use LoanApprovalSample.sh or LoanApprovalSample.cmd to execute the client, just make sure that the classloader can find the samples/dwsample/Customer.class (e.g. add '.' to your classpath if you are executing the script from the parent directory of samples/dwsample.)

This version of the client requires an extra parameter, either 'approve' to gain initial approval for the loan (this makes it perform identically to the older client) or 'obtain' to obtain a loan that you have already been approved for. Remember that we're correlating based on the first and last names, so you'll need to use the same values in both your 'approve' and 'obtain' requests.

Note that both of our scenarios use the same services (i.e. all of the WSDL files except those belonging to the process itself) and the same client to test the functionality. We have broken the process modification down into two parts to make it easier to understand the changes. However, both processes have the same name, so if you run the first scenario you will have to undeploy it before you can run the second.

To run the examples, you'll need to follow the instructions for running the loan approval sample in the BPWS4J documentation, substituting the files included here when appropriate. Below is a list of BPEL and WSDL files for each scenario, along with some comments on how to test the functionality using the client.

Remember, if you would like to see what's going on behind the scenes, you'll need to edit the log4j.properties file in the webapps/bpws4j/WEB-INF/classes directory and set the log4j.rootLogger to DEBUG.

Correlation example

Use the files loanapprovalcorr.bpel and loanapprovalcorr.wsdl.

Deploy the process, and run the client as described above, first to get a loan approved and then to obtain the loan. Keep the parameters (first name, last name, amount) the same for both requests. If you've turned logging on, you will see something like the following when you try to obtain the loan (assuming that you were approved for it of course.):

```
DEBUG [correlation] Testing correlation set: loanIdentifier
DEBUG [correlation] Testing correlation information:
```

```
 PropertyValue[name:{http://loans.org/wsdl/loan-approval}applicantFirstName
 value:[JROMString: http://loans.org/wsdl/loan-approval:applicantFirstName: John]]
DEBUG [correlation] Testing correlation information:
 PropertyValue[name:{http://loans.org/wsdl/loan-approval}applicantLastName
 value:[JROMString: http://loans.org/wsdl/loan-approval:applicantLastName: Doe]]
DEBUG [correlation] Testing correlation information: Match found
```

If you try to obtain the loan a second time (without being re-approved), you should see something like the following in the logs:

```
DEBUG [correlation] Testing correlation set: loanIdentifier
DEBUG [correlation] Testing correlation information:
 PropertyValue[name:{http://loans.org/wsdl/loan-approval}applicantFirstName
 value:[JROMString: http://loans.org/wsdl/loan-approval:applicantFirstName: John]]
DEBUG [correlation] Testing correlation information:
 PropertyValue[name:{http://loans.org/wsdl/loan-approval}applicantLastName
 value:[JROMString: http://loans.org/wsdl/loan-approval:applicantLastName: Doe]]
DEBUG [correlation] Testing correlation information: Match not found
INFO [process] An existing process instance was not found
WARN [process] Neither a flow instance, nor a valid
instance creation receive could be found
```

You will get that message, because the obtain message is trying to access a process that has already terminated.

Fault handling example

Use the files loanapprovalfault.bpel and loanapprovalfault.wsdl.

Deploy the process (making sure that you undeploy the previous one first if you've just run it), and run the client to get a loan approved. Once the loan has been approved, run the client once more, but this time try to obtain more than you have been approved for. The log should say something like the following, and the client should get an exception back from the server:

```
DEBUG [flow] Event channel com.ibm.cs.bpws.runtime.events.EventChannel@5dd34bf7
  is processing event Fault event named
  {http://loans.org/wsdl/loan-approval}loanProcessFault
  from source Activity grant-failure with parent named grant-failure and
  enclosing scope named grant-failure
DEBUG [flow] Container error getting message null
DEBUG [flow] Found fault handller Activity fault-sequence with parent named
new-scope and enclosing scope named new-scope
DEBUG [base] Scope fault-sequence is running
DEBUG [flow] Starting sequence fault-sequence Activity
DEBUG [base] Sequence fault-sequence is running
DEBUG [flow] Sequence fault-sequence about to run null
DEBUG [base] Scope null is running
DEBUG [base] Assign null is running
DEBUG [flow] evaluating condition for link receive-to-grant
DEBUG [flow] about to
eval condition bpws:getContainerData('acceptanceRequest', 'amount')
  <= bpws:getContainerData('request', 'amount')
DEBUG [flow] about to fire link event receive-to-grantfalse
DEBUG [flow] Event channel com.ibm.cs.bpws.runtime.events.EventChannel@5c984bf7
is processing event
  com.ibm.cs.bpws.runtime.events.LinkEvent@43e68bea
DEBUG [flow] Process loanApprovalProcess running; waiting for event
DEBUG [bus] Processing request: [BPWSBus.BUSRequestAssign assignSrc=
com.ibm.cs.bpws.model.FromImpl@2c674be8
  assignDest =com.ibm.cs.bpws.model.ToImpl@2f3ccbe8 activity =
  Activity null with parent named
null and enclosing scope named null
clientData com.ibm.cs.bpws.runtime.AssignRT$AssignInfo@43078bea callback =
Activity loanApprovalProcess
  with no parent and no enclosing scope

Ouch, the call failed:
Fault Code = SOAP-ENV:Client
Fault String = An error occurred while processing the message.
Fault =
[Attributes={}] [faultCode=SOAP-ENV:Client] [faultString=
An error occurred while processing the message.]
  [faultActorURI=null] [DetailEntries=[(0)=
```

```
[name=accept] [type=class java.lang.String]
[value=invalid request: amount too high] [encodingStyleURI=
null]]] [FaultEntries=]
```

The other WSDL files that are used for both examples are loanapprover.wsdl, loanassessor.wsdl and loandefinitions.wsdl.

Next time

In the next article in this series, we will look at the switch and pick activities, and at compensation.

Download

Name	Size	Download method
ws-bpelcol7.zip	9KB	**HTTP**

Information about download methods

Resources

- Participate in the discussion forum.
- Please note that this articles refers to version 1.0 of the BPEL4WS specification. The latest version, BPEL4WS1.1, is now available, and an article describing the key differences between the two specifications will be available shortly.
- Download the BPWS4J engine from alphaWorks.
- Download the zip file with the code samples.

About the authors

Rania Khalaf is a software engineer in the Component Systems group at the IBM T.J. Watson Research Center. She joined IBM in 2001 after having completed her Bachelors and MEng degrees from MIT. Rania is a co-author of the IBM BPEL4WS engine, BPWS4J, available from alphaWorks. You can contact Rania at rkhalaf@watson.ibm.com.

William A. Nagy is a software engineer in the Component Systems group at the IBM T.J. Watson Research Center. He is co-author of WS-Inspection and co-developer of BPWS4J, Apache SOAP, WSTK, and WSGW. He holds a Masters Degree in Computer Science from Columbia University. You can contact William at nagy@watson.ibm.com.