IBM

Business Process with BPEL4WS: Learning BPEL4WS, Part 5

Adding links and manipulating data

Level: Introductory

Matthew Duftler (duftler@us.ibm.com), Software Engineer, IBM TJ Watson Research Center
Francisco Curbera (curbera@us.ibm.com), Manager Component Systems Group, IBM TJ Watson Research Center
Rania Khalaf (rkhalaf@watson.ibm.com), Software Engineer, IBM TJ Watson Research Center

11 Mar 2003

The previous example in Part 2 of this series showed how to build a simple BPEL4WS process that invokes a web service. This article takes that example and expands it into the loan approval process that is included in the BPEL4WS specification and the BPWS4J samples, illustrating the use of links, conditions, and the `<assign>` activity. Links connect activities together, and allow the specification of a condition on each that determines whether or not that link should be followed. Conditions in BPEL4WS are XPath expressions, and this article shows how they can incorporate the process's container data. The `<assign>` activity can be used to copy data into a container when the data is not copied directly as the result of an `<invoke>`.

Introduction

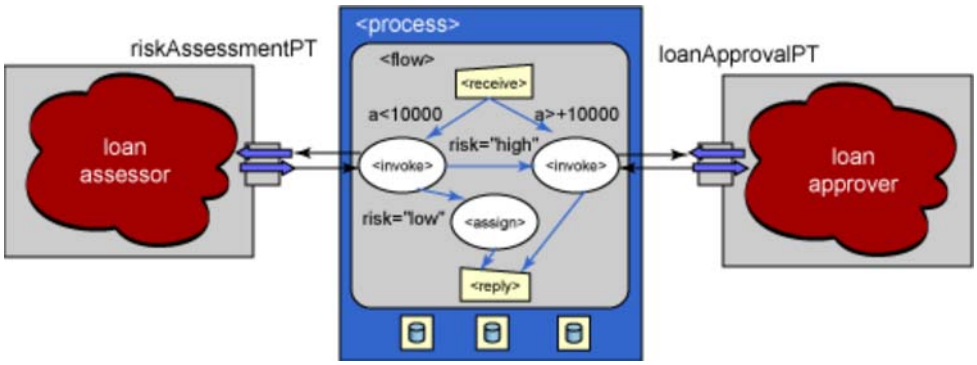
This article will expand the process we created earlier into the loan approval sample that is in the BPEL4WS specification and the BPWS4J samples. It illustrates two core capabilities for defining choreographies: defining flow of control using guarded links, and manipulating data with the `<assign>` activity. We assume that you have read and understood the prior example, and we take you forward from there. As with the earlier example, we will conclude by describing how the process will run, and the results of running it in the BPWS4J engine.

We will show you a process that handles the same loan request -- a customer sends a request for a loan, the request gets processed, and the customer finds out whether the loan was approved. Initially, the middle step consisted of simply invoking a financial institution's Web service and sending its reply back to the customer. Instead of this basic step, you want to employ some additional logic in the handling of the application. You can perform the following sequence of steps to try to determine whether you can grant the loan without going to the financial institution (the *loan approver*) for a full review; if the requested amount is high, it is always sent to the financial institution for review. If the amount is low, you invoke a new Web service, called the *loan assessor*, to determine the risk. If the assessor determines there's low risk giving the applicant the loan, then it can be approved. Otherwise, send it to the financial institution for a full review.

Setting up the process

In BPEL, this process is modeled by using a `<flow>` activity. Remember that a `<flow>` activity allows you to define links joining the activities contained within it, so that is where you will put the logic to do the above processing. Keep the `<receive>` and `<reply>` exactly as before, and add two `<invoke>`s -- one for the assessor and one for the approver. You also add an `<assign>` activity to put your message in the reply. Next, join the `<receive>` with links to the two invokes. These links are guarded by two conditions: if the amount is less than 10,000, then you want to invoke the assessor; if the amount is greater than or equal to 10,000, you want to invoke the approver. Then, link the assessor's invoke to the approver's with the condition that the risk is high, and to the `<assign>` with the condition that the risk is low. Finally, link the approver's invoke and the assign to the `<reply>` without specifying any condition (default). Keep in mind that for this process the flow of control follows the links, as governed by the value of the condition guarding each link. These concepts will be explained in greater detail below. The resulting structure is illustrated in Figure 1.

Figure 1. Internal view of loan approval process



Service descriptions

The WSDL description of the loan assessor is shown in Listing 1. The assessor service can perform one operation, *check*, that returns the level of risk associated with giving the customer a loan.

Listing 1. Loan assessor's WSDL snippet (loanassessor.wsdl)

```
<definitions
   targetNamespace="http://tempuri.org/services/loanassessor"
   xmlns:tns="http://tempuri.org/services/loanassessor"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
   xmlns:loandef="http://tempuri.org/services/loandefinitions"
   xmlns="http://schemas.xmlsoap.org/wsdl/">
<import namespace="http://tempuri.org/services/loandefinitions"
   location=
"http://localhost:8080/bpws4j-samples/loanapproval/loandefinitions.wsdl"/>
<message name="riskAssessmentMessage">
    <part name="risk" type="xsd:string"/>
  </message>
  <portType name="riskAssessmentPT">
    <operation name="check">
      <input message=
      "loandef:creditInformationMessage"/>
      <output message="tns:riskAssessmentMessage"/>
      <fault name="loanProcessFault" message=
      "loandef:loanRequestErrorMessage"/>
    </operation>
  </portType>
  <binding ...> ... </binding>
  <service name="LoanAssessor">....</service>
</definitions>
```

Add one more `serviceLinkType` to the process's definition to model its interaction with the new *assessor* partner (Listing 2). The added service link type states that if someone wants to be an assessor, he or she must have the risk assessment port type defined earlier.

Listing 2. Addition to loan approval WSDL

```
<slnk:serviceLinkType name="riskAssessmentLinkType">
    <slnk:role name="assessor">
      <portType name="asns:riskAssessmentPT"/>
    </slnk:role>
</slnk:serviceLinkType>
```

Creating the process

The next step is to define the process. Starting with the one created last time, remove the <sequence> and the <invoke>. Then, in order to incorporate the loan assessor into the process, add it as a partner with the assessor role in the serviceLinkType just defined, and add a container to receive its output message. To the list of partners, add the following:

```
<partner name="assessor"
        serviceLinkType="lns:riskAssessmentLinkType"
        partnerRole="assessor"/>
```

To the list of containers add the following:

```
<container
        name="riskAssessment"
        messageType="asns:riskAssessmentMessage"/>
```

Flows and links

The presence of links necessitates the use of a <flow> activity, whose definition is started below in Listing 3. The links to be used are defined on the <flow> activity itself by assigning them names.

After defining the links, they are used to link the <receive> activity to the two <invoke>s. (Note: Since two activities are linked together in BPEL using the name of the link, they both need to be contained in the <flow> in which the link was defined.) The definition of the activity that is the source of the link, in this case the <receive>, will have a <source linkName="[link_name] "> element for each outgoing link, and the definition of the activity that is the target of a link, in this case each of the <invoke>s, will have a <target linkName="[link_name]"> for each incoming link. Each link may have only one source and one target.

A link is in a default state before and during the time that its source activity is running. Once the activity completes, each link is activated with a boolean value that is the result of evaluating a *transition condition*. Transition conditions are defined in the <source> element referring to the link in order to explicitly evaluate a link's boolean value. For the <receive> activity, they evaluate whether or not the amount requested is less than 10,000. If no transition condition is defined, the default value used is *true*.

Define the links on the flow and specify the ones starting from the <receive> activity, along with their conditions. You also define the assessor's <invoke>, which is the target of the link *receive-to-assess*, and the source of two of its own links.

Listing 3. Definition of <flow> activity

```
<flow>
    <links>
      <link name="receive-to-assess"/>
      <link name="receive-to-approval"/>
      <link name="approval-to-reply"/>
      <link name="assess-to-setMessage"/>
      <link name="setMessage-to-reply"/>
      <link name="assess-to-approval"/>
    </links>
    <receive name="receive1" partner="customer"
    portType="apns:loanApprovalPT"
                operation="approve" container="request"
                createInstance="yes">
         <source linkName="receive-to-assess"
                transitionCondition=
                "bpws:getContainerData('request', 'amount')<10000"/>
        <source linkName="receive-to-approval"
                transitionCondition=
                "bpws:getContainerData('request', 'amount')>=10000"/>
    </receive>
   <invoke name="invokeAssessor" partner="assessor"
                portType="asns:riskAssessmentPT"
                operation="check" inputContainer="request"
```

```
              outputContainer="riskAssessment">
      <target linkName="receive-to-assess"/>
      <source linkName="assess-to-setMessage"
            transitionCondition=
            "bpws:getContainerData('riskAssessment', 'risk')='low'"/>
      <source linkName="assess-to-approval"
            transitionCondition=
            "bpws:getContainerData('riskAssessment', 'risk')!='low'"/>
    </invoke>
```

Conditions and data assignment

Referring to Figure 1, consider the link that goes from the `<receive>` activity to the assessor's `<invoke>`. In the flow definition above, this is the *receive-to-assess* link, and it is guarded by the transition condition `bpws:getContainerData('request', 'amount')<10000`. As with all types of conditions, transition conditions must be XPath expressions and must be boolean-valued. This particular XPath expression uses one of the XPath extension functions introduced by BPEL4WS: `bpws:getContainerData(...)`. The `bpws:getContainerData(...)` function can be employed by any expression wishing to retrieve data from a container within the process, and its signature is as follows: `bpws:getContainerData("containerName", "partName", "locationPath"?)` where `containerName` is the name of a container, `partName` is the name of a part within that container, and `locationPath` is an optional absolute location path within the specified part.

Refer back to the earlier example to see where the `bpws:getContainerData('request', 'amount')` function gets its data from. It first looks for the *request* container, which is defined in the loanapproval.bpel file. You can see from the definition of the request container that it holds a message of type `loandef:CreditInformationMessage`. By looking in the loandefinitions.wsdl file, you see that the message type `loandef:CreditInformationMessage` contains a part named *amount*, of type `xsd:integer`. You now know that invoking the `bpws:getContainerData('request', 'amount')` function will return an integer, and evaluating the XPath expression `bpws:getContainerData('request', 'amount')<10000` will return true if that integer is less than 10000, and false otherwise. As a result, the *receive-to-assess* link will be activated with the boolean value.

Take a look also at the *assess-to-setMessage* link to determine how its transition condition, `bpws:getContainerData('riskAssessment', 'risk')='low'`, is evaluated. You see from the process defined above that the container *riskAssessment* holds a message of type `asns:riskAssessmentMessage`, and you see from the loanassessor.wsdl file that the message type `asns:riskAssessmentMessage` contains one part: a part named "risk", of type `xsd:string`. So, invoking the `bpws:getContainerData('riskAssessment', 'risk')` function returns a string, which is then compared to the string 'low'. If they match, the entire XPath expression evaluates to true. Keep in mind that since the condition can contain basically any boolean-valued XPath expression, there are many different ways to achieve this same result. For example, instead of `bpws:getContainerData('riskAssessment', 'risk')='low'` you could write `contains(bpws:getContainerData('riskAssessment', 'risk'),'low')` or `starts-with(bpws:getContainerData('riskAssessment', 'risk'), 'lo')`.

At this point, there are two containers, and neither contains information that can be sent directly back to the customer; one contains the customer's original request information, and the other contains the string *low*. What needs to be sent back to the customer is the string *yes*, so you define an `<assign>` activity to accomplish this, as shown in Listing 4.

Listing 4. <assign> activity

```
<assign name="assign">
   <target linkName="assess-to-setMessage"/>
   <source linkName="setMessage-to-reply"/>
   <copy>
        <from expression="'yes'"/>
        <to container="approvalInfo" part="accept"/>
   </copy>
</assign>
```

As you can see above, the `<assign>` activity is the target of the *assess-to-setMessage* link, and the source of the *setMessage-to-reply* link. This assign contains one copy element, and it employs the general expression form of the `<from>` element. When using the general expression form of the `<from>` element, the expression can be anything XPath will allow,

provided it returns an XPath value type (string, number or boolean). In this case, the expression simply specifies the string *yes*, which is then copied into the *accept* part of the *approvalInfo* container. You see from looking at the earlier example that the *accept* part of the message contained in the *approvalInfo* container is of type `xsd:string`.

Finally, there are the approver's `<invoke>` and the `<reply>` (see Listing 5), which are the same as they were earlier, only they are connected to the rest of the activities with the links defined above. The reply sends the answer of whether or not the loan was approved, which by now should be in the container *approvalInfo*. Remember that the container will now be populated because either the `<assign>` put in a *yes* or the approver's `<invoke>` put its answer there. As described in the "Activities and In-Memory Model" article (see Resources), an activity that is the target of some links waits until all of its links have activated, and it has control from its enclosing activity. The default behavior of such an activity is that if at least one of its link activation values is *true*, then it starts running; otherwise, it ends abnormally and sends its links out false. So the approver can't run until it gets the values of both of its links, and only one of them has to be true.

Listing 5. Approver's <invoke> and the <reply>

```
    <invoke name="invokeapprover"
            partner="approver" portType="apns:loanApprovalPT"
            operation="approve"
            inputContainer="request"
            outputContainer="approvalInfo">
      <target linkName="receive-to-approval"/>
      <target linkName="assess-to-approval"/>
      <source linkName="approval-to-reply" />
    </invoke>
    <reply name="reply" partner="customer"
    portType="apns:loanApprovalPT"
    operation="approve" container="approvalInfo">
      <target linkName="setMessage-to-reply"/>
      <target linkName="approval-to-reply"/>
    </reply>
  </flow>
</process>
```

We briefly comment here on what happens when an activity's join condition is false. In the default case, that translates to what happens if all the links coming into an activity are false. We said earlier that the activity disables and send out its links false; this is only half the truth. What really happens is that this is considered a fault in BPEL, the activity throws a `joinFailure` fault, and if it is not caught, the entire process is disabled. However, one way to stop this from happening is to set the global `suppressJoinFailure` attribute on the `<process>` element. This will stop all `joinExceptions` from propagating. As you can see from the full BPEL file, this attribute is set to *true* in this example. We will discuss exceptions in more detail in a later article.

Putting it all together

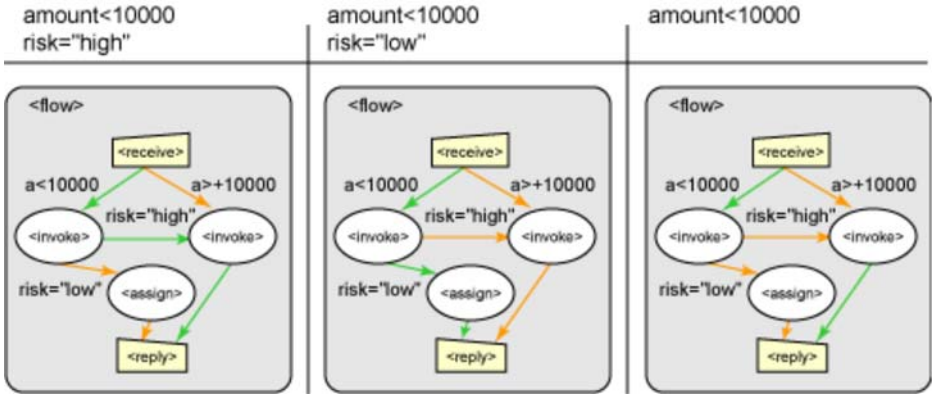
In this section, we show you the possible paths of execution through the process. The possible paths the process would follow are illustrated in Figure 2 below. (Note: To reduce clutter, only the essentials of the flow activity are shown.) Take a look at the left-most scenario in detail. The process starts once a customer has sent a loan request, and the two links coming out of the `<receive>` activate: one false, one true - depending on whether or not the amount requested exceeds 10,000. Assume for the first case the amount was very low and the true link is the one going to the assessor's `<invoke>`, which is now ready to run. The second link, the one going to the approver, goes false, but the approver `<invoke>` must wait for the value of its second link to come in before it can react. The assessor `<invoke>` runs.

Assume that the assessor returns saying the risk is too high. The link *assessor-to-approver* goes true, and the approver also runs since one of its links was true. It is worth noting that in BPEL, an activity with multiple incoming links can define its own condition, known as the *join condition*, on the incoming links. This would be used if the activity required a more sophisticated check than the default *or* on the incoming link values. The approver is invoked.

Once the approver's `<invoke>` completes, putting its answer in the *approvalInfo* container, the link leaving it goes true. At this point you may wonder how the reply will run since it has to wait for the `<assign>` to send the value down the second

link. It actually does get a value on that second link, and that value is false. Here's why: After the assessor's `<invoke>` completed, the *assessor-to-approver* link went true, and the *assessor-to-setMessage* link went false. The `<assign>`, having no other links, immediately disables and sends its link out false. The `<reply>` now has the value of both of its links and one of them is true. It sends the message to the customer and the process ends.

Figure 2. The flows in all possible complete runs of the loan approval process



Note: Green links are activated with 'true' and red ones with 'false.'

Running the process in the BPWS4J engine

As with our earlier articles, if you want to run the process, you'll need to download and install the BPWS4J engine available from alphaWorks (see Resources).

The process we have described here is the loan approval sample distributed with the BPWS4J release, with the fault handler removed. Fault handlers will be discussed and illustrated in a future article in this series.

Follow the instructions for running the loan approval sample in the BPWS4J documentation.

Remember, if you would like to see more of what is happening behind the scenes, go to the log4j.properties file in the webapps/bpws4j/WEB-INF/classes directory and uncomment line 24, which says:
`log4j.logger.bpws.runtime.flow.base=DEBUG`.

Next time

In the next article in this series, we will look at correlation and fault handling. This will be followed by another example BPEL flow that illustrates a concrete process incorporating both of these features.

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 Business Processes for Web Services Java Runtime from alphaWorks.
- Read the Business Process Execution Language for Web services specification (*developerWorks*, July 2002).

- To learn more about activities, read the article "Activities and In-Memory Model" (*developerWorks*, October 2002).

- Read these related articles: "Automating Business processes and transactions in Web services" and "Business Processes in a Web services World" (*developerWorks*, August 2002).

- Take a look at the last article in this series, "Learning BPEL4WS, Part 4: Creating processes with the BPWS4J editor," which describes design approaches to creating BPWS4J processes. (*developerWorks*, November 2002).

About the authors

Matthew J. Duftler is a Software Engineer in the Component Systems group at IBM T.J. Watson Research Center. He was one of the original authors of Apache SOAP, is the co-lead of JSR110, Java APIs for WSDL, and is a co-author of the IBM BPEL4WS engine, BPWS4J. You can contact Matthew at duftler@us.ibm.com.

Francisco Curbera is a Research Staff Member and the Manager of the Component Systems group at IBM T.J. Watson Research Center. He is co-author of the BPEL4WS, WSDL and WSFL specifications, and co-developer of BPWS4J, Apache SOAP and WSTK. He received a PhD in Applied Mathematics from Columbia University. You can contact Matthew at curbera@us.ibm.com.

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.

Share this....

Digg this story del.icio.us Slashdot it!