IBM

Business Process with BPEL4WS: Learning BPEL4WS, Part 2

Creating a simple process

Level: Introductory

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

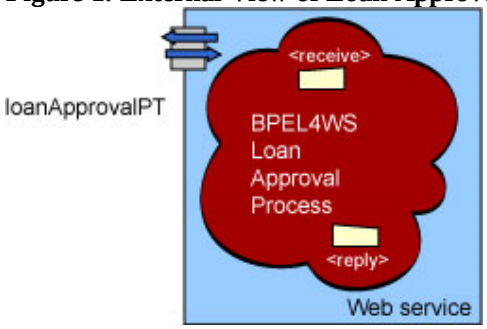
01 Aug 2002

The recently released Business Process Execution Language for Web Services(BPEL4WS) specification is positioned to become the Web services standard for composition. It allows you to create complex processes by creating and wiring together different activities that can, for example, perform Web services invocations, manipulate data, throw faults, or terminate a process. These activities may be nested within structured activities that define how they may be run, such as in sequence, or in parallel, or depending on certain conditions. This series of articles aims to give readers an understanding of the different components of the language, and teach them how to create their own complete processes. The first part of the series will take readers through creating their first simple process. Subsequent parts will extend the example in different ways to illustrate and explain the key parts of the language, including data manipulation, correlation, fault handling, compensation, and the different structured activities in BPEL4WS.

In order to demonstrate how activities may be created and aggregated with BPELWS, I will describe a simple example that processes loan requests. This article will illustrate the main aspects of a composition, as well as show how the WSDL descriptions of services relate to and are used by the BPEL4WS process definition. A complete process is created while explaining the use of partners for interaction, containers for holding messages, and the activities for interacting with the outside world, namely `<receive>`, `<reply>`, and `<invoke>`. In addition to describing how the process will run, I also show how to deploy and run it using the BPWS4J engine available on alphaWorks.

In this example (see Figure 1), 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 will involve sending the application to a Web services enabled financial institution and telling the customer what it decided. From the customer's point of view, the process will consume his application and then send him an answer. The diagram below shows this external view of the loan request process using the cloud diagram introduced in the BPEL4WS overview article. As I continue through this series of tutorials, I will cover additional aspects of the BPEL4WS language by adding levels of complexity to the step that processes the request.

Figure 1. External View of Loan Approval Process

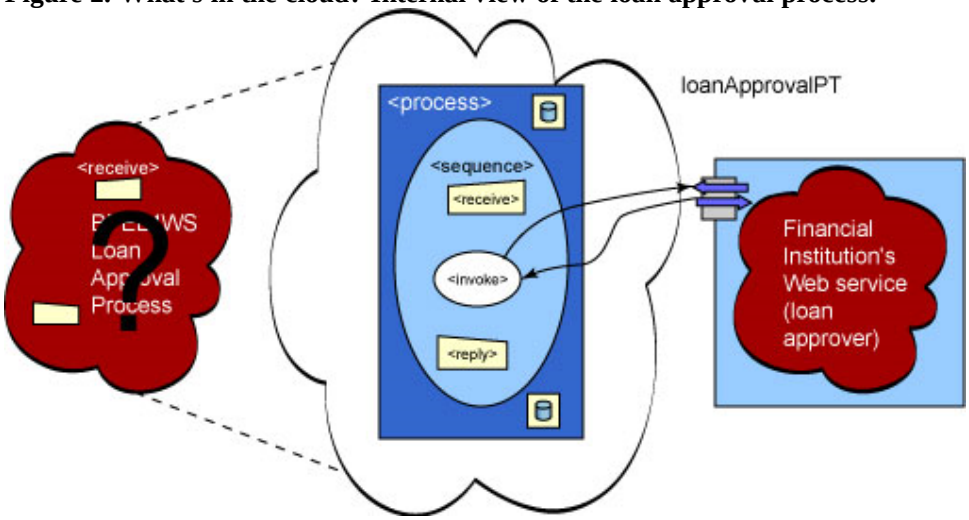


Setting up the process:

The behavior above consists of getting a message, then invoking the financial institution's Web service, and finally replying to the customer. These three actions are defined in BPEL using the `<receive>`, `<invoke>`, and `<reply>` activities. However, the process needs to define the relation of such simple activities to each other in order to know how and when to run them. Such relations are defined in BPEL by using structured activities that define restrictions on how to run the activities they enclose. In this example, you want the three to occur one after the other. This ordering may be achieved in BPEL using a <sequence> activity, that would contain first the `<receive>` to consume the message, followed by an `<invoke>` to talk to

the financial institution, and ending with a `<reply>` to send the answer to the customer. Therefore, the cloud above will contain a process that has a sequence of these three activities, and can invoke the financial institution as illustrated in Figure 2.

Figure 2. What's in the cloud? Internal view of the loan approval process.



Before you can begin fleshing out the process, you need to provide formal descriptions of the parties that will be involved and the messages that will be exchanged and manipulated.

Creating the service descriptions: Using WSDL

BPEL compositions rely heavily on WSDL descriptions of the involved services in order to refer to the messages being exchanged, the operations being invoked, and the portTypes these operations belong to. In the example, you will need the description of the financial institution and the process itself. Consider that the financial world uses a unified set of messages for describing loan information, and has those defined in the loan definitions in Listing 1.

Listing 1: Loan Definitions WSDL (loandefinitions.wsdl)

```
<definitions targetNamespace="http://tempuri.org/services/loandefinitions"
             xmlns:tns="http://tempuri.org/services/loandefinitions"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

  <message name="creditInformationMessage">
     <part name="firstName" type="xsd:string"/>
     <part name="name" type="xsd:string"/>
     <part name="amount" type="xsd:integer"/>
  </message>

  <message name="loanRequestErrorMessage">
     <part name="errorCode" type="xsd:integer"/>
   </message>

 </definitions>
```

Assume you know of a financial institution that provides a loan approval service and is described by Listing 2 below. It contains one a single operation, "approve", which it uses to decide the status of a loan request. The operation takes information about the customer as input, and outputs an approval message containing the answer. The definition for the input message is defined in the loandefintions WSDL above.

Listing 2: Loan Approver WSDL(loanapprover.wsdl)

```
<definitions targetNamespace="http://tempuri.org/services/loanapprover"
```

```
                    xmlns:tns="http://tempuri.org/services/loanapprover"
                    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                    xmlns:loandef="http://tempuri.org/services/loandefinitions"
                    xmlns="http://schemas.xmlsoap.org/wsdl/">

  <import namespace="http://tempuri.org/services/loandefinitions"
          location="http://localhost:8080/bpws-samples/loanapproval/loandefinitions.wsdl"/>

  <message name="approvalMessage">
    <part name="accept" type="xsd:string"/>
  </message>

  <portType name="loanApprovalPT">
    <operation name="approve">
      <input message="loandef:creditInformationMessage"/>
      <output message="tns:approvalMessage"/>
      <fault name="loanProcessFault"
             message="loandef:loanRequestErrorMessage"/>
    </operation>
  </portType>

<binding ...> ... </binding>
<service name="LoanApprover">....</service>
</definitions>
```

The process itself simply forwards the input and output messages to and from this service. Therefore, it will present the same description to the user by referencing the above portType. One more required thing is to define `serviceLinkType`s for the services used. The `serviceLinkType` defines up to two roles that refer to the portTypes that are provided and required by any two services it links together. In the case of this example, this `serviceLinkType` will be used to link the customer to the process, as well as the process to the loan approver. Only one role is required because both the process itself and the loan approver service provide the "approver" portType, and neither of them requires the user to support another portType. You create the code in Listing 3 below for the process:

Listing 3: Loan Approval WSDL(loan-approval.wsdl)

```
<definitions
       targetNamespace="http://loans.org/wsdl/loan-approval"
       xmlns="http://schemas.xmlsoap.org/wsdl/"
       xmlns:slnk="http://schemas.xmlsoap.org/ws/2002/06/service-link/"
       xmlns:xsd="http://www.w3.org/2001/XMLSchema"
       xmlns:lns="http://loans.org/wsdl/loan-approval"
       xmlns:apns="http://tempuri.org/services/loanapprover">

   <import namespace="http://tempuri.org/services/loanapprover"
           location="http://localhost:8080/bpws-samples/loanapproval/loanapprover.wsdl"/>
   <import namespace="http://tempuri.org/services/loandefinitions"
           location="http://localhost:8080/bpws-samples/loanapproval/loandefinitions.wsdl"/>

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

   <service name="loanapprovalServiceBP"/>
 </definitions>
```

Creating the process

All the requirements are now available for creating the process. You begin the definition with the `<process>` element, and include the namespaces that will allow it to refer to the required WSDL information, where the message definitions are defined (*http://..../loandefinitions*), the target namespace of the loan approver (*http://.../loanapprover*), and the target namespace of the

process's own WSDL (*http://.../loan-approval*). The process is now able to use the loan approver service as a component.

```
<process name="loanApprovalProcess"
           targetNamespace="http://acme.com/simpleloanprocessing"
           xmlns="http://schemas.xmlsoap.org/ws/2002/07/business-process/"
           xmlns:lns="http://loans.org/wsdl/loan-approval"
           xmlns:loandef="http://tempuri.org/services/loandefinitions"
           xmlns:apns="http://tempuri.org/services/loanapprover">
```

The next step is to declare the parties involved. Named partners are defined, each characterized by a WSDL `serviceLinkType`. For this example, the partners are the customer and the financial institution. The `myRole/partnerRole` attribute on a partner specifies how the partner and the process will interact given the `serviceLinkType`. The `myRole` attribute refers to the role in the `serviceLinkType` that the process will play, whereas the `partnerRole` specifies the role that the partner will play. This is illustrated in the partner definitions below. The loan approval process offers the functionality of the `loanApprovalPT` to the customer, and the financial institution in turn offers that functionality to the process. This relationship can be seen in Figures 1 and 2 above.

```
  <partners>
     <partner name="customer"
                 serviceLinkType="lns:loanApproveLinkType"
                 myRole="approver"/>
     <partner name="approver"
                 serviceLinkType="lns:loanApprovalLinkType"
                 partnerRole="approver"/>
  </partners>
```

After defining the partners, you are nearly ready to start adding the activities that form the composition. Let's review what you want the process to do. In order to ask for a loan, the customer sends the process a message, the process asks the financial institution whether it will accept the loan application, and replies to the customer with another message either accepting or refusing the application. How do you do this in BPEL? First of all, you need to put the incoming message where a BPEL activity can access it. In BPEL, data is written to and accessed from data containers which can hold instances of specific WSDL message types.

From the definition of the customer partner and the loanApprovalPT, it is clear that the customer will send a message of type `creditInformationMessage` and get a reply of type `approvalMessage`. Therefore, the following list of containers is added and are illustrated in Figure 2 as blue cylinders:

```
  <containers>
     <container name="request" messageType="loandef:CreditInformationMessage"/>
     <container name="approvalInfo"  messageType="apns:approvalMessage"/>
  </containers>
```

Interacting with the process: Receive, invoke, reply

A process may contain only one activity, which in this case will be the `<sequence>`. Now you can add to the *sequence* a *receive* activity that can take the customer's message and put it in the appropriate container. The definition of a *receive* activity must include the partner that will send it its message, and the port type and operation of the process that the partner is targeting this message to. Based on this information, once the process gets a message, it searches for an active *receive* activity that has a matching partner-portType-operation triplet and hands it the message. In order to avoid confusion, the specification states that there may not be multiple receive activities with the same partner-portType-operation triplet that are active at the same time. The activity will then place the message in the specified container and end. You start the *sequence* activity, and add the *receive* to it:

```
  <sequence>
     <receive name="receive1" partner="customer"
                 portType="apns:loanApprovalPT"
                 operation="approve" container="request"
                 createInstance="yes">
```

```
    </receive>
```

The next step is to ask the Web services-enabled financial institution whether or not it will accept the loan. This is done with a regular Web services invocation, defined in the process by an Invoke activity. When this activity runs it will make the specified invocation to the Web service using the message in its input container, put the answer it gets into its output container, and end. Note that the call will be made on the "approver" partner to perform the `approve` operation.

```
    <invoke name="invokeapprover"
              partner="approver"
              portType="apns:loanApprovalPT"
              operation="approve"
              inputContainer="request"
              outputContainer="approvalInfo">
    </invoke>
```

In order for the process to respond to the customer's request, it uses a Reply activity. Once a reply activity is reached, the partner-portType-operation triplet it has is used to figure out whom to send the reply to. Therefore, in order to reply to the message that arrived through the Receive activity, you would need a Reply activity with the same triplet. In this case, you want to tell the customer what the financial institution decided, so the message to be sent will be found in the output container of the invoke: `approvalInfo`. After the reply, the process ends. You close the sequence and process tags.

```
    <reply name="reply" partner="customer" portType="apns:loanApprovalPT"
             operation="approve" container="approvalInfo">
    </reply>
  </sequence>
</process>
```

Putting it all together

Once a process is deployed, it waits until somebody starts it. If you noticed, the *receive* contains an attribute called "createInstance" which is set to true. That shows us an entry point into the process. Figure 3 illustrates how the loan approval process will run.

Figure 3. Running the loan approval process.

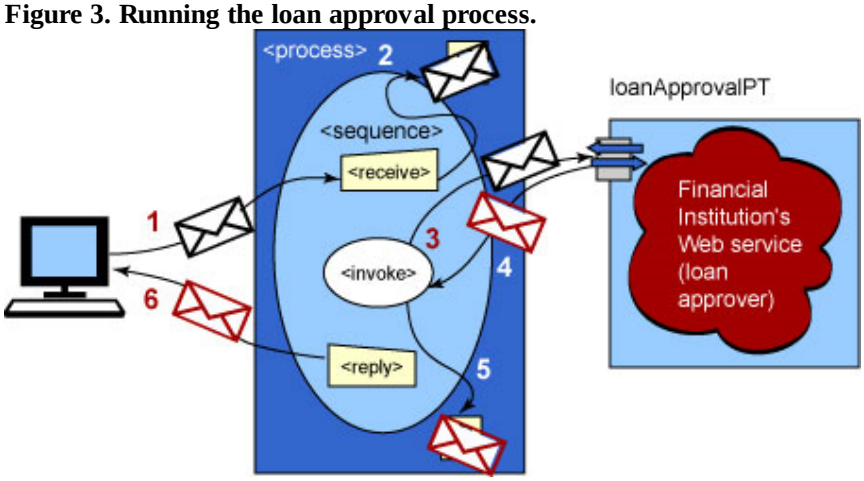


Figure Note: The numbers on the arrows indicate the order in which the steps occur. The black envelope is the message containing the loan request. The red envelope is the message containing the answer to that request.

Once a client sends a message to a process manager with the appropriate triplet, a process instance is created and starts running. In the given example, the process would start up the *sequence*, which would in turn start the *receive*. The message has arrived so it will be put into the "request" container. The *invoke* will then occur. After the message that resulted from the invocation is placed in the "approvalInfo" container, the *reply* will take it and send it to the customer at which point that

instance of the process ends. Multiple instances of the same process may be running simultaneously. When I explain more complex processes, you will see how correlation is used to route messages to and from the correct instance of the same process.

Running the process in BPWS4J

If you would like to run the process you have just created, you will need to get the BPWS4J engine available from alphaWorks. (See Resources.) Follow the installation instructions provided, and don't forget to put the bpws4j-samples.war file in the appropriate directory for your application server to find it.

The process you have here is a simplified version of the loan approval sample available with the BPWS4J release (which is also the one in the specification). You will use the loan approver service provided in that sample, as well as the WSDLs of both the loan approver and the process itself (loanapproval.wsdl). Those files are identical to ours, except that the process's WSDL contains an extra `serviceLinkType` for an additional partner that their process uses.

Look up the information for running the BPWS4J samples, which is in the documentation of the engine. You will follow the first three steps of the instructions exactly for running the loan approval sample. These changes are specifically in steps four and five of how to use the Loan Approval sample. Instead of giving it the loan-approval.bpel file provided with the release, you want to give it the one you have just created.

To create the BPEL file you will be using, cut and paste the process defined above (from the beginning to the end of the `<process>` element) into a file also called loanapproval.bpel and save it to some other directory so as not to overwrite the sample. Now, when you deploy the service as described in step 4, give it the WSDL file of the process as specified in documentation, and the BPEL file you just created (see Resources for source code files).

The deployment Web page will then ask you for the WSDL file of the loan approver partner only. Give it the file as described in the documentation. To know whether you have accidentally deployed the BPEL file that came with the engine instead, you can check yourself by seeing whether it asks you for another partner's WSDL called the loan assessor. If it does that then you gave it the BPEL file in the samples instead of ours and you need to start over. So for step 5, you will do exactly what the documentation says except for the last sentence about deploying an "assessor" partner. You are now ready to run the process. Follow the steps for executing the BPWS4J loan approval sample. You should get a *yes* or *no* when you ask for a loan. The reason the client works with your BPEL file is that the entry points to both processes are the same, and the names are the same.

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, that says:

```
log4j.logger.bpws.runtime.flow.base=DEBUG
```

This will show you when each of the activities you created started running.

```
DEBUG [base] Scope loanApprovalProcess is running
DEBUG [base] Sequence null is running
DEBUG [base] Receive receive1 is running
DEBUG [base] Invoke invokeapprover is running
DEBUG [base] Reply reply is running
```

Next time

In the next part of this article, I will go through some more parts of the BPEL4J language and illustrate their usage by adding more activities to the loan approval example. In order not to be confusing, the additions will keep bringing the sample closer to the one in the specification and BPWS4J release. In the meantime, you may want to read the other articles available about the language and the runtime.

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.
- Check out the first installment of this column, Business processes, Part 1.
- Download the Business Processes for Web Services Java Runtime from alphaWorks.
- Read the Business Process Execution Language for Web services specification.
- Read these related articles: Automating Business processes and transactions in Web services and Business Processes in a Web services World.

About the author

Rania Khalaf is a Software Engineer in the Component Systems group at the IBM TJ 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 the author at rkhalaf@watson.ibm.com.

Share this....