**Developer: J2EE & Web Services**

**A Hands-on Introduction to BPEL**

*by Matjaz B. Juric*

**Learn how to create an example business process that combines a set of fictional travel-related Web services and then deploy it to the Oracle BPEL Process Manager runtime environment.**

|  |  |
| --- | --- |
|

|  |
| --- |
| **Downloads for this article** [Sample code](http://www.oracle.com/technetwork/articles/bpel-part1-support-128639.zip) [Oracle BPEL Process Manager & Designer](http://www.oracle.com/technetwork/database/options/spatialandgraph/devlic-091771.html)  |

 |

[Business Process Execution Language for Web Services](http://www-128.ibm.com/developerworks/library/ws-bpel/) (BPEL or BPEL4WS) is a language used for the definition and execution of business processes using Web services. BPEL enables the top-down realization of Service Oriented Architecture (SOA) through composition, orchestration, and coordination of Web services. BPEL provides a relatively easy and straightforward way to compose several Web services into new composite services called *business processes*.

In this article, you will learn how to create an example business process that combines a set of fictional travel-related Web services and then deploy it to the Oracle BPEL Process Manager runtime environment.

**BPEL Background**

First, some background. BPEL builds on the foundation of XML and Web services; it uses an XML-based language that supports the Web services technology stack, including SOAP, WSDL, UDDI, WS-Reliable Messaging, WS-Addressing, WS-Coordination, and WS-Transaction.

BPEL represents a convergence of two early workflow languages; Web Services Flow Language (WSFL) and XLANG. WSFL was designed by IBM and is based on the concept of directed graphs. XLANG, a block-structured language, was designed by Microsoft. BPEL combines both approaches and provides a rich vocabulary for description of business processes.

The first version of BPEL was developed in August 2002. Since then, many major vendors have joined (including Oracle), resulting in several modifications and improvements, and the adoption of version 1.1 in March 2003. In April 2003, BPEL was submitted to the Organization for the Advancement of Structured Information Standards (OASIS) for standardization purposes, and the Web Services Business Process Execution Language Technical Committee ( [WSBPEL TC](http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel" \t "_blank)) was formed. This effort has led to even broader acceptance in industry.

Within the enterprise, BPEL is used to standardize enterprise application integration as well as to extend the integration to previously isolated systems. Between enterprises, BPEL enables easier and more effective integration with business partners. BPEL stimulates enterprises to further define their business processes, which in turn leads to business process optimization, reengineering, and the selection of the most appropriate processes, thus further optimizing the organization. Definitions of business processes described in BPEL do not affect existing systems, thereby stimulating upgrades. BPEL is the key technology in environments where functionalities are already or will be exposed via Web services. With increases in the use of Web services, the importance of BPEL will increase as well.

**Orchestration versus Choreography**

Web services usually expose operations of certain applications or information systems. Consequently, combining several Web services actually involves the integration of the underlying applications and their functionalities.

Web services can be combined in two ways:

* Orchestration
* Choreography

In orchestration, which is usually used in private business processes, a central process (which can be another Web service) takes control of the involved Web services and coordinates the execution of different operations on the Web services involved in the operation. The involved Web services do not "know" (and do not need to know) that they are involved in a composition process and that they are taking part in a higher-level business process. Only the central coordinator of the orchestration is aware of this goal, so the orchestration is centralized with explicit definitions of operations and the order of invocation of Web services. (See Figure 1)



**Figure 1:** Composition of Web services with orchestration

Choreography, in contrast, does not rely on a central coordinator. Rather, each Web service involved in the choreography knows exactly when to execute its operations and with whom to interact. Choreography is a collaborative effort focusing on the exchange of messages in public business processes. All participants in the choreography need to be aware of the business process, operations to execute, messages to exchange, and the timing of message exchanges. (See Figure 2.)



**Figure 2:** Composition of Web services with choreography

From the perspective of composing Web services to execute business processes, orchestration is a more flexible paradigm and has the following advantages over choreography:

* The coordination of component processes is centrally managed by a known coordinator.
* Web services can be incorporated without their being aware that they are taking part in a larger business process.
* Alternative scenarios can be put in place in case faults occur.

BPEL supports two different ways of describing business processes that support orchestration and choreography:

* **Executable processes** allow you to specify the exact details of business processes. They follow the orchestration paradigm and can be executed by an orchestration engine.
* **Abstract business protocols** allow specification of the public message exchange between parties only. They do not include the internal details of process flows and are not executable. They follow the choreography paradigm.

Now, let's walk through the creation of an executable BPEL business process; the code for it can be downloaded and deployed to the [Oracle BPEL Process Manager](http://www.oracle.com/technetwork/middleware/bpel/overview/index.html). We will assume that Oracle BPEL Process Manager has been successfully installed according to the installation instructions and that it uses the default port 9700. If another port has been selected during installation, the examples have to be modified accordingly.

**Building a Business Process**

A BPEL process specifies the exact order in which participating Web services should be invoked, either sequentially or in parallel. With BPEL, you can express conditional behaviors. For example, an invocation of a Web service can depend on the value of a previous invocation. You can also construct loops, declare variables, copy and assign values, define fault handlers, and so on. By combining all these constructs, you can define complex business processes in an algorithmic manner. In fact, because business processes are essentially graphs of activities, it might be useful to express them using Unified Modeling Language (UML) activity diagrams.

In a typical scenario, the BPEL business process receives a request. To fulfill it, the process invokes the involved Web services and then responds to the original caller. Because the BPEL process communicates with other Web services, it relies heavily on the WSDL description of the Web services invoked by the composite Web service.

Let's take an example. A BPEL process consists of steps; each step is called an "activity." BPEL supports primitive as well as structure activities. *Primitive* activities represent basic constructs and are used for common tasks, such as the following:

* Invoking other Web services, using <invoke>
* Waiting for the client to invoke the business process by sending a message, using <receive> (receiving a request)
* Generating a response for synchronous operations, using <reply>
* Manipulating data variables, using <assign>
* Indicating faults and exceptions, using <throw>
* Waiting for some time, using <wait>
* Terminating the entire process, using <terminate>

We can then combine these and other primitive activities to define complex algorithms that specify exactly the steps of business processes. To combine primitive activities, BPEL supports several *structure* activities. The most important are

* Sequence ( <sequence>), which allows us definition of a set of activities that will be invoked in an ordered sequence
* Flow ( <flow>) for defining a set of activities that will be invoked in parallel
* Case-switch construct ( <switch>) for implementing branches
* While ( <while>) for defining loops
* The ability to select one of several alternative paths, using <pick>

Each BPEL process will also define partner links, using <partnerLink>, and declare variables, using <variable>.

To understand how business processes are described with BPEL, you will define an oversimplified business process for employee travel arrangements: The client invokes the business process, specifying the name of the employee, the destination, the departure date, and the return date. The BPEL business process first checks the employee travel status, assuming that a Web service exists through which such checks can be made. Then the BPEL process will check the price for the flight ticket with two airlines: American Airlines and Delta Airlines. Again assume that both airline companies provide a Web service through which such checks can be made. Finally, the BPEL process will select the lower price and return the travel plan to the client.

Then, we will build an asynchronous BPEL process. We will assume that the Web service for checking the employee travel status is synchronous. This is reasonable because such data can be obtained immediately and returned to the caller. To acquire the plane ticket prices we use asynchronous invocations. Again, this is a reasonable approach, because it might take a little longer to confirm the plane travel schedule. We assume that both airlines offer a Web service and that both Web services are identical (i.e. provide equal port types and operations) to simplify our example.

In real-world scenarios, you will usually not have the choice about the Web services but will have to use whatever services are provided by your partners. If you have the luxury of designing the Web services and BPEL process at the same time, you will want to consider which interface is better. Usually you'll use asynchronous services for long-lasting operations and synchronous services for operations that return a result in a relatively short time. If you use asynchronous Web services, the BPEL process is usually asynchronous as well.

When you define a business process in BPEL, you essentially define a new Web service that is a composite of existing services. The interface of the new BPEL composite Web service uses a set of port types through which it provides operations like any other Web service. To invoke a business process described in BPEL, you have to invoke the resulting composite Web service. Figure 3 shows a schematic view of our process.



**Figure 3:** Example BPEL process for travel arrangements

In develop the sample BPEL process, you will go through the following steps:

* Get familiar with the involved Web services
* Define the WSDL for the BPEL process
* Define partner link types
* Develop the BPEL process:
	+ Define partner links
	+ Declare variables
	+ Write the process logic definition.

**Step 1: Inventory the Involved Web Services**

Before you can start writing the BPEL process definition, you have to become familiar with the Web services invoked from our business process. These services are called *partner Web services*. This example has the Employee Travel Status Web service and the American and Delta Airlines Web service, which have identical WSDL descriptions. (Again, the Web services used in this example are fictional.)

**Employee Travel Status Web Service.** The Employee Travel Status Web service provides the EmployeeTravelStatusPT port type through which the employee travel status can be checked with the EmployeeTravelStatus operation. The operation will return the travel class that an employee can use, which can be economy, business, or first. (See Figure 4.)



**Figure 4:** The Employee Travel Status Web service

**Airline Web Service.** The Airline Web service is asynchronous; therefore, it specifies two port types: The first, FlightAvailabilityPT, is used to check the flight availability using the FlightAvailability operation. To return the result, the Web service specifies the second port type, FlightCallbackPT. This port type specifies the FlightTicketCallback operation.

Although the Airline Web service defines two port types, it implements only the FlightAvailabilityPT. FlightCallbackPT is implemented by the BPEL process, which is the client of the Web service. The architecture of the Web service is schematically shown in Figure 5:



**Figure 5:** The Airline Web service

**Step 2: Define WSDL for the BPEL Process**

Next, we have to expose the business travel BPEL as a Web service. The second step is therefore to define the WSDL for it. The process has to receive messages from its clients and return results. It has to expose the TravelApprovalPT port type, which will specify an input message. It also has to declare the ClientCallbackPT port type, used to return the result to the client (asynchronously, using a callback). This process is illustrated in Figure 6.



**Figure 6:** The WSDL for the BPEL process

**Step 3: Define Partner Link Types**

The third step is to define the partner link types. Partner link types represent the interaction between a BPEL process and the involved parties, which include the Web services the BPEL process invokes and the client that invokes the BPEL process.

In our example, there are three different partners: the client, the employee travel status Web service, and the airline Web service. Ideally, each Web service should define the corresponding partner link types (in the WSDL). (In real-world scenarios, this may not be the case.) Then we can wrap the partner Web service with a WSDL that imports the WSDL of the Web service and defines the partner link types. Alternatively, we can define all partner links in the WSDL of the BPEL process. However, this approach isn't recommended as it violates the principle of encapsulation.

For our example, we define three partner link types, each in the corresponding WSDL of the Web service:

* travelLT: used to describe the interaction between the BPEL process client and the BPEL process itself. This interaction is asynchronous. This partner link type is defined in the WSDL of the BPEL process.
* employeeLT: used to describe the interaction between the BPEL process and the employee travel status Web service. This interaction is synchronous. This partner link type is defined in the WSDL of the employee Web service.
* flightLT: describes the interaction between the BPEL process and the airline Web service. This interaction is asynchronous and the airline Web service invokes a callback on the BPEL process. This partner link type is defined in the WSDL of the airline Web service.

Each partner link type can have one or two roles and for each role we must specify the portType it uses. For synchronous operations, there is a single role for each partner link type because the operation is only invoked in a single direction. For example, the BPEL process invokes the EmployeeTravelStatus operation on the employee travel status Web service. Because it is a synchronous operation, the BPEL process waits for completion and gets a response only after the operation is completed.

For asynchronous callback operations, you have to specify two roles. The first role describes the invocation of the operation by the client. The second role describes the invocation of a callback operation. In our example there is an asynchronous relation between the BPEL process and the airline Web service.

As you already know, three partner link types are required: two specify two roles because they are asynchronous, and one specifies a single role because it is synchronous.

Partner link types are defined in the WSDL within a special namespace: <http://schemas.xmlsoap.org/ws/2003/05/partner-link/>. First let's define the travelLT link type in the BPEL process WSDL used by clients to invoke the BPEL process. The first role required is the role of the travel service (that is, our BPEL process). The client uses the TravelApprovalPT port type to communicate with the BPEL service. The second role, travelServiceCustomer, characterizes the client to which the BPEL process will perform a callback on the ClientCallbackPT port type:

<plnk:partnerLinkType name="travelLT">

 <plnk:role name="travelService">

 <plnk:portType name="tns:TravelApprovalPT" />

 </plnk:role>

 <plnk:role name="travelServiceCustomer">

 <plnk:portType name="tns:ClientCallbackPT" />

</plnk:role>

</plnk:partnerLinkType>

The second link type is employeeLT. It is used to describe the communication between the BPEL process and the employee travel status Web service and is defined in the WSDL of the employee Web service. The interaction is synchronous, so we need a single role, called employeeTravelStatusService. The BPEL process uses the EmployeeTravelStatusPT on the employee Web service:

<plnk:partnerLinkType name="employeeLT">

 <plnk:role name="employeeTravelStatusService">

 <plnk:portType name="tns:EmployeeTravelStatusPT" />

</plnk:role>

</plnk:partnerLinkType>

The last partner link type, flightLT, is used to describe the communication between the BPEL process and the airline Web service. This communication is asynchronous. The BPEL process invokes an asynchronous operation on the airline Web service. The Web service, after it has completed the request, invokes a callback on the BPEL process. Therefore we need two roles. The first role describes the participation of the airline Web service to the BPEL process, which is the airline service ( airlineService). The BPEL process uses the FlightAvailabilityPT port type to make the asynchronous invocation. The second role describes the participation of the BPEL process to the airline Web services. For the airline Web service, the BPEL process is an airline customer, hence the role name airlineCustomer. The airline Web service uses the FlightCallbackPT port type to make the callback. This partner link type is defined in the WSDL of the airline Web service:

<plnk:partnerLinkType name="flightLT">

 <plnk:role name="airlineService">

 <plnk:portType name="tns:FlightAvailabilityPT" />

 </plnk:role>

 <plnk:role name="airlineCustomer">

 <plnk:portType name="tns:FlightCallbackPT" />

 </plnk:role>

</plnk:partnerLinkType>

Understanding partner link types is crucial for developing a BPEL process specification. Sometimes it helps to make a diagram of all the interactions. Once the partner link types are defined, you have finished the preparation phase.

**Step 4: Create the Business Process**

Now you are ready to start writing the BPEL process definition. Typically, a BPEL process waits for an incoming message from the client, which starts the execution of the business process. In our example the client initiates the BPEL process through sending an input message: TravelRequest. Then the BPEL process invokes the Employee Travel Status Web service, sending the EmployeeTravelStatusRequest message. Because this invocation is synchronous, it waits for the EmployeeTravelStatusResponse message. Then the BPEL process makes a concurrent asynchronous invocation of both Airline Web services by sending them the FlightTicketRequest message. Each Airline Web service makes a callback, sending the TravelResponse message. The BPEL process then selects the more appropriate airline and makes a callback to the client with the TravelResponse message.

You start with an empty BPEL process outline that presents the basic structure of each BPEL process definition document:

<process name="BusinessTravelProcess" ... >

 <partnerLinks>

 <!-- The declaration of partner links -->

 </partnerLinks>

 <variables>

 <!-- The declaration of variables -->

 </variables>

 <sequence>

 <!-- The definition of the BPEL business process main body -->

 </sequence>

</process>

Let's first add the required namespaces. Here you have to define the target namespace and the namespaces to access the Employee and Airline WSDLs and the BPEL process WSDL. You also have to declare the namespace for all the BPEL activity tags (here the default namespace, so you do not have to qualify each BPEL tag name). The BPEL activity namespace must be http://schemas.xmlsoap.org/ws/2003/03/business-process/:

<process name="BusinessTravelProcess"

 targetNamespace="http://packtpub.com/bpel/travel/"

 xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"

 xmlns:trv="http://packtpub.com/bpel/travel/"

 xmlns:emp="http://packtpub.com/service/employee/"

 xmlns:aln="http://packtpub.com/service/airline/" >

...

**Partner Links.** Next you have to define the *partner links*, which define different parties that interact with the BPEL process. Each partner link is related to a specific partnerLinkType that characterizes it. Each partner link also specifies one or two attributes:

* myRole: Indicates the role of the business process itself.
* partnerRole: Indicates the role of the partner.

The partner link can specify a single role, which is usually the case with synchronous request/response operations. For asynchronous operations it specifies two roles. In our example, you'll define four roles. The first partner link is called client and is characterized by the travelLT partner link type. The client invokes the business process. You need to specify the myRole attribute to describe the role of the BPEL process ( travelService). You have to specify the second role: partnerRole. Here, this is travelServiceCustomer, which characterizes the BPEL process client.

The second partner link is called employeeTravelStatus and is characterized by the employeeLT partner link type. It is a synchronous request/response relation between the BPEL process and the Web service; you again specify only one role. This time it is the partnerRole, because you describe the role of the Web service, which is a partner to the BPEL process.

The last two partner links correspond to the airline Web services. Because they use the same type of Web service, we specify two partner links based on a single partner link type, flightLT. Here you have asynchronous callback communication, so you need two roles. The role of the BPEL process ( myRole) to the airline Web service is airlineCustomer, whereas the role of the airline ( partnerRole) is airlineService:

 <partnerLinks>

 <partnerLink name="client"

 partnerLinkType="trv:travelLT"

 myRole="travelService"

 partnerRole="travelServiceCustomer"/>

 <partnerLink name="employeeTravelStatus"

 partnerLinkType="emp:employeeLT"

 partnerRole="employeeTravelStatusService"/>

 <partnerLink name="AmericanAirlines"

 partnerLinkType="aln:flightLT"

 myRole="airlineCustomer"

 partnerRole="airlineService"/>

 <partnerLink name="DeltaAirlines"

 partnerLinkType="aln:flightLT"

 myRole="airlineCustomer"

 partnerRole="airlineService"/>

 </partnerLinks>

**Variables.** Variables in BPEL processes are used to store, reformat, and transform messages. You usually need a variable for every message sent to the partners and received from them. In this process you need seven variables. Let's call them TravelRequest, EmployeeTravelStatusRequest, EmployeeTravelStatusResponse, FlightDetails, FlightResponseAA, FlightResponseDA, and TravelResponse.

For each variable you have to specify the type. You can use a WSDL message type, an XML Schema simple type, or an XML Schema element. In this example use WSDL message types for all variables:

 <variables>

 <!-- input for this process -->

 <variable name="TravelRequest"

 messageType="trv:TravelRequestMessage"/>

 <!-- input for the Employee Travel Status Web service -->

 <variable name="EmployeeTravelStatusRequest"

 messageType="emp:EmployeeTravelStatusRequestMessage"/>

 <!-- output from the Employee Travel Status Web service -->

 <variable name="EmployeeTravelStatusResponse"

 messageType="emp:EmployeeTravelStatusResponseMessage"/>

 <!-- input for American and Delta Web services -->

 <variable name="FlightDetails"

 messageType="aln:FlightTicketRequestMessage"/>

 <!-- output from American Airlines -->

 <variable name="FlightResponseAA"

 messageType="aln:TravelResponseMessage"/>

 <!-- output from Delta Airlines -->

 <variable name="FlightResponseDA"

 messageType="aln:TravelResponseMessage"/>

 <!-- output from BPEL process -->

 <variable name="TravelResponse"

 messageType="aln:TravelResponseMessage"/>

 </variables>

**BPEL Process Main Body.** The process main body specifies the order in which the partner Web services are invoked. Usually it starts with a <sequence> that allows you to define several activities that will be performed sequentially. Within the sequence, you first specify the input message that starts the business process. You do that with the <receive> construct, which waits for the matching messagein our case the TravelRequest message. Within the <receive> construct, you do not specify the message directly. Rather we specify the partner link, the port type, the operation name, and optionally the variable that holds the received message for consequent operations.

We link the message reception with the client partner, and wait for the TravelApproval operation to be invoked on port type TravelApprovalPT. We store the received message into the TravelRequest variable:

 <sequence>

 <!-- Receive the initial request for business travel from client -->

 <receive partnerLink="client"

 portType="trv:TravelApprovalPT"

 operation="TravelApproval"

 variable="TravelRequest"

 createInstance="yes" />

...

<receive> waits for the client to invoke the TravelApproval operation and stores the incoming message and parameters about the business trip into the TravelRequest variable. Here, the variable name is the same as the message name, but this is not necessary.

Next, you need to invoke the employee travel status Web service. Before that however we have to prepare the input for this Web service. Looking at the WSDL, you'll see that you have to send a message that consists of the employee part. You can construct such a message by copying the employee part of the message that is sent the client. You would write the corresponding assignment:

...

 <!-- Prepare the input for the Employee Travel Status Web Service -->

 <assign>

 <copy>

 <from variable="TravelRequest" part="employee"/>

 <to variable="EmployeeTravelStatusRequest" part="employee"/>

 </copy>

 </assign>

...

Now you can invoke the Employee Travel Status Web service. You make a synchronous invocation, for which we use the <invoke> activity. You use the employeeTravelStatus partner link and invoke the EmployeeTravelStatus operation on the EmployeeTravelStatusPT port type. You have prepared the input message in the EmployeeTravelStatusRequest variable. Because it is a synchronous invocation, the call waits for the reply and stores it in the EmployeeTravelStatusResponse variable:

...

 <!-- Synchronously invoke the Employee Travel Status Web Service -->

 <invoke partnerLink="employeeTravelStatus"

 portType="emp:EmployeeTravelStatusPT"

 operation="EmployeeTravelStatus"

 inputVariable="EmployeeTravelStatusRequest"

 outputVariable="EmployeeTravelStatusResponse" />

...

The next step is to invoke both airline Web services. Again first prepare the required input message (which is identical for both Web services). The FlightTicketRequest message comprises two parts:

* flightData: This is retrieved from the client message ( TravelRequest).
* travelClass: This is retrieved from the EmployeeTravelStatusResponse variable.

Therefore, write an assignment with two copy elements:

...

 <!-- Prepare the input for AA and DA -->

 <assign>

 <copy>

 <from variable="TravelRequest" part="flightData"/>

 <to variable="FlightDetails" part="flightData"/>

 </copy>

 <copy>

 <from variable="EmployeeTravelStatusResponse" part="travelClass"/>

 <to variable="FlightDetails" part="travelClass"/>

 </copy>

 </assign>

...

The input data includes the data that needs to be passed to the airline Web services. Because it is in the same format, you can pass it directly (using a simple copy). In the real world, you usually need to perform a transformation. You could do that using XPath expressions with <assign>, use a transformation service (such as XSLT engine), or use the transformation capabilities provided by specific BPEL servers.

Now you are ready to invoke both airline Web services. Make concurrent asynchronous invocations; to express concurrency, BPEL provides the <flow> activity. The invocation to each Web service will comprise two steps:

* The <invoke> activity is used for the asynchronous invocation.
* The <receive> activity is used to wait for the callback.

Use <sequence> to group both activities. The two invocations differ only in the partner link name: AmericanAirlines for one and DeltaAirlines for the other. Both invoke the FlightAvailability operation on the FlightAvailabilityPT port type, sending the message from the FlightDetails variable.

The callback is received using the <receive> activity. Again, you use both partner link names. <receive> waits for the FlightTicketCallback operation to be invoked on the FlightCallbackPT port type. Store the result message in the FlightResponseAA and the FlightResponseDA variables respectively:

...

 <!-- Make a concurrent invocation to AA in DA -->

 <flow>

 <sequence>

 <!-- Async invoke of the AA Web service and wait for the callback-->

 <invoke partnerLink="AmericanAirlines"

 portType="aln:FlightAvailabilityPT"

 operation="FlightAvailability"

 inputVariable="FlightDetails" />

 <receive partnerLink="AmericanAirlines"

 portType="aln:FlightCallbackPT"

 operation="FlightTicketCallback"

 variable="FlightResponseAA" />

 </sequence>

 <sequence>

 <!-- Async invoke of the DA Web service and wait for the callback-->

 <invoke partnerLink="DeltaAirlines"

 portType="aln:FlightAvailabilityPT"

 operation="FlightAvailability"

 inputVariable="FlightDetails" />

 <receive partnerLink="DeltaAirlines"

 portType="aln:FlightCallbackPT"

 operation="FlightTicketCallback"

 variable="FlightResponseDA" />

 </sequence>

 </flow>

...

In this stage of the process, you have two ticket offers. In the next step, you have to select one. To do that, use the <switch> activity.

...

 <!-- Select the best offer and construct the TravelResponse -->

 <switch>

 <case condition="bpws:getVariableData('FlightResponseAA',

 'confirmationData','/confirmationData/Price')

 <= bpws:getVariableData('FlightResponseDA',

 'confirmationData','/confirmationData/Price')">

 <!-- Select American Airlines -->

 <assign>

 <copy>

 <from variable="FlightResponseAA" />

 <to variable="TravelResponse" />

 </copy>

 </assign>

 </case>

 <otherwise>

 <!-- Select Delta Airlines -->

 <assign>

 <copy>

 <from variable="FlightResponseDA" />

 <to variable="TravelResponse" />

 </copy>

 </assign>

 </otherwise>

 </switch>

...

In the <case> element, check whether the offer from American Airlines ( FlightResponseAA) is equal to or better than the offer from Delta ( FlightResponseDA). Use the BPEL function getVariableData and specify the variable name for this purpose. The price is inside the confirmationData message part, which is the only message part, but you still have to specify it. You also have to specify the query expression to locate the price element. Here, this is a simple XPath 1.0 expression.

If the American Airlines offer is better Delta's, copy the FlightResponseAA variable to the TravelResponse variable (which you finally return to the client). Otherwise copy the FlightResponseDA variable.

You have come to the final step of the BPEL business processreturning a callback to the client using the <invoke> activity. For the callback, we use the client partner link and invoke the ClientCallback operation on the ClientCallbackPT port type. The variable that holds the reply message is TravelResponse:

...

 <!-- Make a callback to the client -->

 <invoke partnerLink="client"

 portType="trv:ClientCallbackPT"

 operation="ClientCallback"

 inputVariable="TravelResponse" />

 </sequence>

</process>

|  |
| --- |
| **Next Steps** **Download a sample chapter from the book *[Business Process Execution Language for Web Services](http://www.oracle.com/technetwork/topics/1183-bpel-preview-ch4-128278.zip)***[**Visit the Web Services Technology Center**](http://www.oracle.com/technology/webservices/index.html) |

With this, you have concluded our first business process specification in BPEL. You can see that BPEL is not very complicated and allows a relatively easy and natural specification of business processes.

**Step 5: Deploy and Test**

Each BPEL process you deploy to the Oracle BPEL Process Manager requires a process descriptor. This process descriptor is not covered by the BPEL standard and is specific to the BPEL server. The deployment process descriptor is the only part of the implementation of a process on a given platform that must be re-written to run the process on a different BPEL engine. The Oracle process descriptor is an XML file specifying the following details about the BPEL process: BPEL source file name, BPEL process name (ID), WSDL locations of all partner link Web services, and optional configuration properties. The default filename for the process descriptor is bpel.xml, but you can use any other name:

<BPELSuitcase>

 <BPELProcess src="Travel.bpel" id="TravelProcessCh4">

 <partnerLinkBindings>

 <partnerLinkBinding name="client">

 <property name="wsdlLocation">

 Travel.wsdl

 </property>

 </partnerLinkBinding>

 <partnerLinkBinding name="employeeTravelStatus">

 <property name="wsdlLocation">

 http://localhost:9700/orabpel/default/Employee/Employee?wsdl

 </property>

 </partnerLinkBinding>

 <partnerLinkBinding name="AmericanAirlines">

 <property name="wsdlLocation">

 http://localhost:9700/orabpel/default/AmericanAirline/AmericanAirline?wsdl

 </property>

 </partnerLinkBinding>

 <partnerLinkBinding name="DeltaAirlines">

 <property name="wsdlLocation">

 http://localhost:9700/orabpel/default/DeltaAirline/DeltaAirline?wsdl

 </property>

 </partnerLinkBinding>

 </partnerLinkBindings>

 </BPELProcess>

</BPELSuitcase>

You are now ready to start the BPEL Process Manager. You can do this from the Start menu (if using Windows) or by executing the startOraBPEL script, which can be found in the c:\orabpel\bin directory (assuming Oracle BPEL Process Manager has been installed in c:\orabpel). It is recommended that you include this directory in the path for easy access.

The Oracle BPEL Process Manager includes an Ant utility called obant, which you can use to configure complex compilation and deployment scenarios. obant is just a wrapper around standard Ant, which sets the environment and then invokes the standard Ant Java task. To use it you have to prepare the corresponding project file, usually called build.xml. The project file for our travel example process looks like this:

<?xml version="1.0"?>

<project name="TravelProcessCh4" default="main" basedir=".">

 <property name="deploy" value="default"/>

 <property name="rev" value="1.0"/>

 <target name="main">

 <bpelc home="${home}" rev="${rev}" deploy="${deploy}"/>

 </target>

</project>

To compile and deploy our BPEL process you simply start obant from the command line.

Now that you have successfully deployed a BPEL process on the Oracle BPEL Server, it's time to execute it. Oracle BPEL Process Manager provides a BPEL Console through which you can execute, monitor, manage, and debug BPEL processes on a BPEL Server domain. The BPEL Console is accessible at http://localhost:9700/BPELConsole/. You have to click the process name, fill out the following form and click on the Post XML Message button:



**Figure 7:** BPEL Console

You now get a screen notifying you that the process instance is being processed asynchronously. You can select the visual flow of the execution, instance auditing, or instance debugging. The visual flow of the instance graphically shows the execution of a BPEL process instance. You can monitor the execution of the process and its state (running, completed, canceled, or stale):



**Figure 8:** Graphical view of the instance's flow

Oracle BPEL Process Manager provides several options for auditing, managing, debugging, and deploying of BPEL processes. For more information, see the [product documentation](http://www.oracle.com/technetwork/middleware/bpel/overview/index.html).

**Conclusion**

Now that you are familiar with the basic concepts of Web services composition with BPEL, you can delve more deeply into more advanced concepts. In my [next article](http://www.oracle.com/technetwork/articles/matjaz-bpel2-082861.html), I'll explain advanced BPEL features such as fault handling, scopes, compensation, concurrent activities, and event handling.

[Part 2: Advanced BPEL](http://www.oracle.com/technetwork/articles/matjaz-bpel2-082861.html)

**Matjaz B. Juric** holds a Ph.D. in computer and information science. He is the author of the book [*Business Process Execution Language for Web Services*](http://www.amazon.com/exec/obidos/tg/detail/-/1904811183/qid%3D1106325670/sr%3D1-5/ref%3Dsr_1_5/102-8657616-9037757?v=glance&s=books) (Packt Publishing). He is also the coauthor of *Professional J2EE EAI*, *Professional EJB*, *J2EE Design Patterns Applied*, and *VB.NET Serialization Handbook*, all published by Wrox Press. Matjaz has also contributed to *Java Developer's Journal*, *Java Report*, *Java World*, and other publications.

http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html