IBM

Business Process with BPEL4WS: Learning BPEL4WS, Part 6

Correlation, fault handling, and compensation

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

01 Mar 2003

The previous articles have covered the fundamentals of BPEL4WS, providing you with an understanding of the activities defined and how they can be combined together using structured activities and the `<link>` construct. In this article, we cover the advanced properties of the language that are essential to the definition and execution of a business process. BPEL uses correlation to match returning or known customers with a long-running business process, fault handling to recover from expected as well as unexpected faults, and compensation to "undo" already committed steps in case something goes wrong in the middle of the process or, for example, a client wishes to explicitly cancel a transaction.

Introduction

Now that you have learned the fundamental concepts of BPEL4WS, it is time to dive into some of the more advanced topics. This article will provide you with an overview of message correlation, fault handling, and compensation as it applies to a BPEL4WS environment.

Message correlation

Message correlation is the BPEL4WS mechanism which allows processes to participate in stateful conversations. It can be used, for example, to match returning or known customers to long-running business processes. When a message arrives for a Web service which has been implemented using BPEL, that message must be delivered somewhere -- either to a new or an existing instance of the process. The task of determining to which conversation a message belongs, in BPEL's case the task of locating/instantiating the instance, is what message correlation is all about.

In many distributed object systems, one component of the routing of a message involves examining the message for an explicit *instance ID* which identifies the destination. Although the routing process is similar, BPEL instances are not identified by an explicit instance field, but are instead identified by one or more sets of key data fields within the exchanged messages. For example, an order number may be used to identify a particular instance of a process within an order fulfillment system. In BPEL terms, these collections of data fields which identify the process instance are known as correlation sets.

Each BPEL correlation set has a name associated with it, and is composed of WSDL-defined properties. A *property* is a named, typed data element which is defined within a WSDL document, and whose value is extracted from an instance of a WSDL message by applying a message-specific XPath expression. In WSDL, a *propertyAlias* defines each such mapping. The mappings are message specific, hence a single *property* can have multiple *propertyAlias*es associated with it. For example, a WSDL document might say that property *name* corresponds to part *username* of WSDL message *loginmsg* and to part *lastname* of *ordermsg*. Together, the properties and propertyAliases provide BPEL authors with a way to reference a single, logical piece of information in a consistent way, even if it might appear in different forms across a set of messages.

Using correlation sets

To use a correlation set, a BPEL author defines the set by enumerating the properties which comprise it, and then references that set from *receive*, *reply*, *invoke*, or *pick* activities. A BPEL runtime uses the definition and references to determine the interesting pieces of information that it should examine during different points of executing the process. Each instance of the process has an instantiation of each of the correlation sets which are defined for the process. Each of these instantiations is

initialized exactly once during the execution of the process instance, and is subsequently only used during comparisons involving incoming and outgoing messages. If an attempt is made to reinitialize a correlation set, or to use one which has not been initialized, then the runtime will throw a *bpws:correlationViolation*. Extracting values for a correlation set referenced by an activity, either for initialization or comparison purposes, involves applying the *processAlias*, which corresponds to the particular WSDL message being examined, for each of the properties which comprise the referenced correlation set.

As *receive* and *pick* activities provide the entry points into a process, correlation sets often appear on them to enable message-to-instance routing. If a correlation set appears on a *receive* or *pick* activity, and it does not have the *initiation* attribute set to 'yes', then when a message arrives for that particular *receive* or *pick*, the values of the properties which comprise the correlation set are extracted from the incoming message and compared to the values stored for the same correlation set in all of the instances of that process, and the message is routed to the instance which has matching values. Similarly, correlation sets on *reply* and *invoke* activities, which deal with outbound operations, are often used to validate that outgoing messages contain data which is consistent with the data contained within specified correlation set instances. For example, if a correlation set appears on a *reply* activity, and it does not have the *initiation* attribute set to 'yes', then the values of the properties which comprise the correlation set are extracted from the outgoing message and compared to the values store for the instance's instantiated correlation set. If the values in the message which correspond to the correlation set are found to be different from those contained within the instance's correlation set, the runtime throws a *bpws:correlationViolation*. On all of the activities on which correlation sets may appear, the *initiation* attribute indicates to the runtime that the correlation set should be initialized using the values contained within the message which is input to the current activity. Correlation sets which appear within *invoke* activities have an additional attribute, *pattern*, which specifies during the execution of the *invoke* activity the correlation set that is to be applied. The *pattern* attribute may have a value of 'in', meaning that the correlation set is to be applied when the response comes back for the invocation, 'out', meaning that the correlation set is to be applied when the invocation is made, or 'both', which naturally means that it is to be applied during both phases.

Multiple correlation sets, some of which are initialized and some of which are used for comparison, can appear on a single activity. The current BPEL specification does not define the semantics of locating an instance based on multiple correlation sets.

Error handling and roll-back

When executing a BPEL process, errors might occur either in the services being invoked or within the process itself. BPEL provides a mechanism to explicitly catch such errors and handle them by executing subroutines specified in *fault handler* elements. Additionally, activities that have completed might later need to be undone because they form part of a longer transaction that had to be aborted. *Compensation handlers* allow the creator of the process to define certain actions that should be taken to undo a unit of work and restore the data to what it was before that work was done.

The handling of such situations usually affects a set of activities that are associated with each other. In BPEL, this is done by enclosing them in the *scope* structured activity. A *scope* provides the context for the activities nested within it, and it is where fault and compensation handlers are defined. Therefore, you can think of the scope as encapsulating a possibly compensatable, recoverable unit of work.

The entire process provides the global scope: it contains one main activity and allows the specification of fault and compensation handlers. It is also able to define data containers and correlation sets. However, the future directions of BPEL state that all scopes will be able to do the same, and that containers defined in a scope will be visible only to activities nested within that scope.

In order to understand how a scope catches or propagates a fault that originated within it, we first recap the life cycle of a BPEL activity. An activity remains dormant, waiting to be activated by its parent activity and its incoming links. Once it gets control from its parent, and all its incoming links come in, it can evaluate its join condition at which point one of two things can happen: it either throws a *joinFailure* fault because the join condition was not satisfied, or it runs. Assuming the latter, after it has run successfully, it ends by evaluating and firing all its outgoing links.

On the other hand, the activity might fault while it is running because it was an invocation that returned a fault, one of the built-in BPEL faults occurred while it was executing, or because it is a *throw* activity. In this case, it notifies the scope it belongs to. Upon receiving a fault, the scope must stop all its nested activities. One other reason can cause an activity not to run successfully; it might never receive all its links or control from its parent. For example, this will happen to an activity in

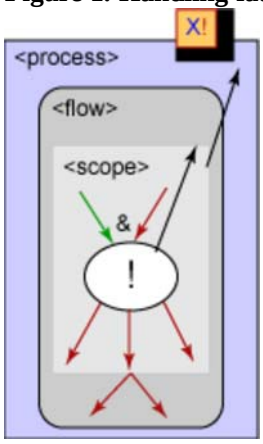
the same scope as the one that faulted but that was slated to run later on, or to an activity that is a branch of a switch statement that was never taken. In both of these cases where an activity cannot successfully run and complete, it needs to send out all its outgoing links with a negative value.

Handling faults

BPEL includes the ability to catch and handle errors at different levels of activity nesting. A fault handler can be defined on any scope and either bound to a particular kind of fault (defined by the fault's qualified name, or its message type) or to any faults not caught by a more specific handler. A handler simply contains an activity that will run in case an error occurs. For example, it might contain a reply activity that notifies a partner that an error has occurred.

As noted earlier, the first thing a scope does once it receives a fault is to stop all its nested activities. In the default case, the scope stops its activities, and then rethrows the fault to its parent and so on until the top-level process is reached. When a scope rethrows the fault, the scope itself then ends abnormally, and it sends out all its outgoing links with a negative value. However, handlers allow any scope to intercept a fault and take appropriate action. Once a fault is handled in a scope, the scope ends normally with the values of its outgoing links being evaluated as usual. This is illustrated in Figure 1.

Figure 1. Handling faults



In this figure, the activity faults because one of its links came in with a negative value and its join condition is a boolean *and* of the link values. It therefore throws the fault to its scope, and you see all the links leaving it go negative. The scope itself has no handler for the fault, so it rethrows the fault. The next scope up is the process itself. You see the links of the inner scope get negative values because it could not take care of the problem itself and so terminated abnormally. The process has a handler, takes care of the fault, and completes normally. It follows that if the handler had been on the inner scope, the links leaving it would have been evaluated regularly, and the fault would not have been thrown to the process.

The invoke activity exhibits a short-cut mechanism for defining handlers directly on its definition. The behavior is effectively that it becomes wrapped in a scope that will have these specified handlers.

BPEL defines a set of built-in faults that notify of errors such as assignment type mismatches, a scope forcing the termination of a nested scope, and activating a reply activity that doesn't have a matching receive activity. The BPEL engine is responsible for detecting these faults and throwing them to the appropriate scope.

Compensation

While a business process is running, it might be necessary to undo one of the steps that have already been successfully completed. The specification of these undo steps are defined using compensation handlers that can be defined at the scope level. Each handler contains one activity which is run when a scope needs to be compensated.

The activities within such a handler must see the container data to be that which it was when the scope completed. Due to activities sharing containers and loops caused by *while* activities, completing scopes that are compensation-enabled must save a snapshot of the data for the handler to possibly use later.

Once a scope completes successfully, its compensation handler becomes ready to run. This can happen in either of two cases: explicit or implicit compensation. Explicit compensation occurs upon the execution of a *compensate* activity. This activity may occur anywhere, and refers to the name of the scope that it wants compensated. Scopes that might be rolled back in this manner must therefore be named. BPEL further specifies that names of scopes must be unique within a BPEL process. When a

compensate activity is reached, it runs the compensation handler on the specified scope.

On the other hand, implicit compensation occurs when faults are being handled and propagated. Consider the scenario in which a scope *A* contains a compensatable scope *B* that has completed normally, but then another nested activity in *A* throws a fault. Implicit compensation ensures that whatever happened in scope *B* gets undone by running its compensation handler. Therefore, implicit compensation of a scope goes through all its nested scopes and runs their compensation handlers in *reverse order of completion* of those scopes.

Conclusion

In this article we have covered the advanced concepts of correlation and fault handling, including compensation. The next article will provide a runnable example that illustrates both the use of correlation for matching messages to the appropriate instances, as well as the use of a fault handler to catch and take care of errors in the process.

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 previous installments of the *Business Process with BPEL4WS* column.

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.