Business process choreography in WebSphere: Combining the power of BPEL and J2EE

by M. Kloppmann
D. König
F. Leymann
G. Pfau
D. Roller

Business processes not only play a key role in business-to-business and enterprise application integration scenarios by exposing the appropriate invocation and interaction patterns; they are also the fundamental basis for building heterogeneous and distributed applications (workflow-based applications). Business Process Execution Language for Web Services (BPEL4WS) provides the means to specify business processes that are composed of Web services as well as exposed as Web services. Business processes specified via BPEL4WS are portable; they can be carried out by every BPEL4WS-compliant execution environment. In this paper we show how the IBM J2EE™ application server, WebSphere® Application Server provides such an environment, called process choreographer environment, and how the extension mechanism built into BPEL can be used to leverage the additional capabilities of J2EE and WebSphere.

Web services based on the service-oriented architecture framework serve as the foundation for modern distributed, heterogeneous applications by providing a virtual component model.[1] Not only can new components be written as Web services, but existing applications also can be provided as Web services made available through a variety of formats and protocols in a vendor-independent and language-neutral form.[1] Web services are perfectly suited as the function layer of the two-level programming model that is the characteristic of workflow-based applications.[2]

Workflow-based applications are composed of two distinct pieces: a process model that describes the sequence in which the different activities making up the process model are being carried out (programming in the large) and the individual components that implement the various activities (programming in the small). In the Web services environment, process models are described using the Business Process Execution Language for Web services (BPEL4WS), abbreviated as BPEL in the rest of the paper.[3] The implementations of the activities are specified as Web services; the actual implementation can be done in any language and programming model. Particularly compelling is the fact that in BPEL, business processes are exposed as Web services, providing for a recursive aggregation model.[4]

The purpose of the process choreographer is to manage the life cycle of business processes, to navigate through the associated process model, and to invoke the appropriate Web services. Besides the navigation and invocation capabilities, the process choreographer must provide the appropriate quality-of-service (QoS) characteristics, such as maintaining a certain response time or ensuring particular security constraints.

The Java 2 Platform, Enterprise Edition (J2EE**) environment, as implemented by IBM's WebSphere* Application Server, provides an environment for the deployment and execution of mission-critical, enterprise-level applications.

The support of Web services is now complemented with the appropriate support for BPEL. In order to run BPEL processes on a J2EE application server, BPEL artifacts must be mapped to appropriate J2EE artifacts, for example, BPEL processes need to be rendered as EJB* (Enterprise JavaBeans**) -based components. A side effect of this mapping is the capability of Java** applications to have native access to business processes, which drives the demand for additional Java capabilities to be exposed within BPEL, such as the usage of Java as an expression language besides XPath (XML Path Language), or the ability of having Java types in addition to XML (eXtensible Markup Language) types. The extension mechanism built into BPEL can be used to add J2EE-related constructs to BPEL. Furthermore, WebSphere provides the support of people as a resource, a function that is extremely helpful in business processes to manage situations where human involvement is mandatory. The extension mechanism built into BPEL is also used to add WebSphere-related constructs.

In this paper, we describe the implementation of a BPEL-compliant engine in the WebSphere environment and the extensions that have been defined to address additional J2EE and WebSphere functions. In the next section we present the basic features of BPEL, focusing on the more advanced features of BPEL, such as long-running business processes and failure and compensation handling. In the third section we present a programming model for the J2EE and WebSphere environment that is based on BPEL. It illustrates how BPEL processes are mapped to the J2EE environment, how Java interfaces can be supported in addition to Web Services Description Language (WSDL) port types, and how Java is used as an expression language. Furthermore, it illustrates how BPEL can be extended to accommodate the support of humans as resources in a business process. In the fourth section we show how business processes are developed and then deployed in the J2EE environment. Last, in the fifth section we present the WebSphere-based infrastructure for the execution of business processes. We start with describing the functional capabilities and then have a detailed look at the QoS characteristics of the infrastructure.

Web services composition with BPEL

The BPEL specification was submitted in April 2003 to the Organization for the Advancement of Structured Information Standards (OASIS) for standardization. The submitted specification has been created jointly by IBM, Microsoft Corporation, BEA Systems, Inc., SAP AG, Inc., and Siebel Systems, Inc., and the BPEL Technical Committee at OASIS includes members from many infrastructure and application vendors throughout the industry.

In this section we describe those features of BPEL that are necessary to understand the running example we use for illustrating the key features of our engine. Comprehensive introductions to BPEL can be found in a number of papers (e.g., Reference 5). Subsequent sections continue to use that example, extend it, and map it into the J2EE world and onto the IBM J2EE application server, WebSphere.

Invoking several Web services in a certain order—control flow. The primary goal of BPEL is to specify the sequence in which a number of services can be invoked, where "sequence" includes parallel and conditional branches. The invoked services are specified using WSDL.[6]

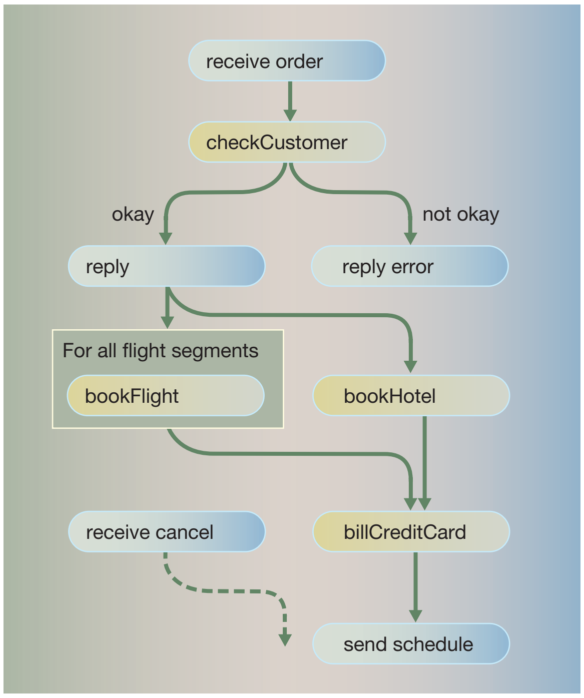
Suppose we want to create a business process that describes the steps to book a simple journey at a travel agent. We use a simplified example in which only a trip to a single destination is possible. The rough outline of such a business process is as follows: First, the customer data about the destination has to be checked. Then, the appropriate flight segments to and from that destination are booked. Also, a hotel at the destination is booked. Finally, payment from the customer is obtained via credit card.

The outline of that business process is shown graphically in Figure 1. The flights and the hotel are actually booked only if a check of the customer data indicates that the data are satisfactory; otherwise, there will be an error. Note that the operations for booking the flights and the hotel can be performed in parallel. An excerpt of the BPEL syntax rendering that process is shown in Figure 2.

In BPEL, a step calling out to a Web service is rendered as an *invoke* activity. An invoke activity specifies the Web service operation that is to be called when the activity is reached during execution of the process. The concrete syntax of the invoke activity in BPEL is shown in Figure 2, line 11, where the bookFlight operation provided by an external Web service is called.

BPEL provides several ways to aggregate activities. One very simple possibility is to use a *sequence*, which

Figure 1 Travel booking process



performs all contained activities sequentially. In our example, we have used a *flow*, which allows for the parallel execution of its contained activities, and the additional specification of ordering constraints using *links*. A link between two activities A_1 and A_2 specifies that A_2 can start only when A_1 has been completed. Links can have associated *transition conditions*, predicates evaluated at runtime that determine whether a link is actually followed or not. In our example, if the customer data were found to be satisfactory, both the links to the steps for booking the flight and the hotel are followed, whereas the link to the error step is not ("dead-path elimination"—see Reference 7).

BPEL provides additional constructs to describe the control flow of processes, such as *switch* activities for a structured way of making decisions, and *while* activities to perform iterations. The example uses a while activity to iterate over all flight segments, as shown in Figure 2, line 10.

Providing a composition as a Web service. A BPEL process is made available to potential exploiters as a Web service, too, implementing one or more operations of one or more port types. In business-to-business scenarios, entire conversations between participating partners are driven by processes. BPEL allows the capture of the signatures of the Web services involved in such conversations by means of a BPEL-provided WSDL extension: partner link types. A *partner link type* describes the interaction between two Web services by means of the roles each of the partners is playing and the interfaces each of the partners is providing.

The travel booking process interacts with a number of Web services. Each interaction is described by a *partner link* of a specific type. There is the interaction with the traveler (line 2), where the process provides operations of a port type called TravelAgentPT (book and cancel, not visible in the example), and expects the traveler to provide operations contained in the port type TravelerCallbackPT (with rcvTravelData and rcvErrorData operations, again not visible). Also, the process interacts with an airline reservation system to invoke operations of the flight reservation port type (line 3).

On some occasions, one role of a partner link type is empty because only one partner provides operations for the other one to call. No bilateral communication can happen in such a case, only a simple invocation (unilateral communication). In the example, the partner link between the travel booking process and the airline reservation system is of that nature, where the latter only provides operations and the former only invokes those operations. Thus, partner links allow for a uniform rendering of all kinds of interactions between two Web services, whether they are inbound only, outbound only, or truly bilateral.

A business process declares that it actually implements a Web service operation by means of a *receive* activity that defines the port type and operation on which the request is received from a specific partner. If the implemented operation happens to be a request-response operation, an associated *reply* activity at a later point in the process is responsible for delivering the result, which can also be a fault. In the example in Figure 2, the section from line 6 to line 9 shows how the travel booking process provides the book operation, with its two possible outcomes of a good or a fault result. On invocation of the book operation, the request reaches the receive activity in line 6. The *createInstance* attribute in the receive activity controls whether or not a new process instance should be created upon receipt of that mes-

Figure 2 Travel booking process—BPEL example

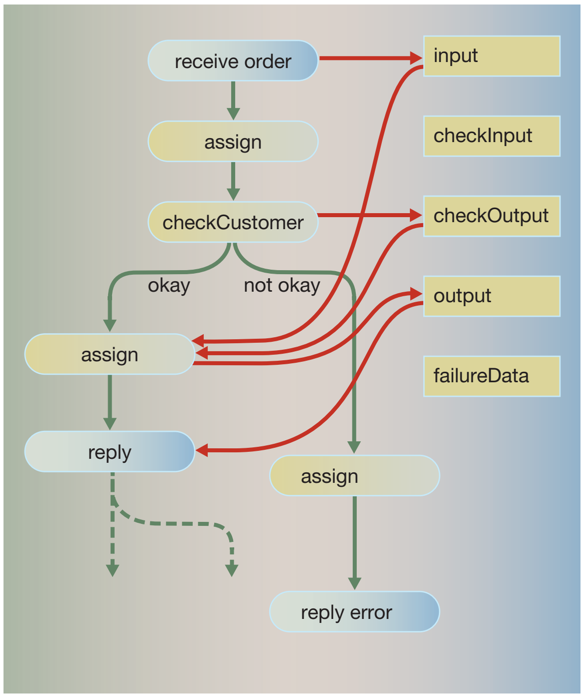
```
1   <process name="TravelBookingProcess"
                 xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"
                 xmlns:bpelp="http://www.ibm.com/websphere/v5.1/business-process/">
      <partnerLinks>
2         <bpelp:partnerLink name="traveler">
              <bpelp:myPortType name="TravelAgentPT"/>
              <bpelp:partnerPortType name="TravelerCallbackPT"/>
          </bpelp:partnerLink>
3         <bpelp:partnerLink name="airlineReservationSystem">
              <bpelp:partnerPortType name="FlightReservationPT"/>
          </bpelp:partnerLink>
          ...
      </partnerLinks>
      ...
4     <flow>
5         <links>...</links>

6         <receive partnerLink="traveller"
                   portType="travel:TravelAgentPT" operation="book"
                   createInstance="yes" .../>
          <assign>...</assign>
7         <invoke name="checkCustomer" partnerLink="internalServices"
                  portType="CheckOperationsPT" operation="checkCustomer".../>
          <assign>...</assign>
8         <reply name="reply" partnerLink="traveller"
                 portType="travel:TravelAgentPT" operation="book" .../>
          ...
9         <reply name="replyFault" partnerLink="traveller"
                 portType="travel:TravelAgentPT"operation="book"
                 faultName="failure" .../>
          ...
10        <while condition="getVariableData('currentLegIndex')  &lt;
                            getVariableData('input','flights','@noOfLegs')">
11            <invoke name="bookFlight" partnerLink="airlineReservationSystem"
                      portType="FlightReservationPT" operation="bookFlight" .../>
              <assign>...</assign>
          </while>
          ...
      </flow>
    </process>
```

sage. Following the receive activity, invocation of the customer check takes place in line 7. Thereafter, the reply to the book operation is sent, returning a good result if the check succeeded (line 8) or a fault otherwise (line 9). The final result of the process, the travel schedule, is eventually delivered asynchronously at a later time using a callback provided by the traveler side, which the process calls using a standard invoke activity. (See the send schedule activity, shown in Figure 1.)

As we have seen until now, besides ordering of activities, the BPEL description of a business process is only concerned with the signatures of the involved services, that is, their port types, operations, and message types, which are referenced through the notion of partner links. Binding of actual services to those partner links, described by their endpoint references[8] is mostly outside the scope of BPEL. This binding can be done during modeling of the business process, or at deployment time, or even at runtime, for exam-

Figure 3 Data flow using variables and assigned activities



ple, through lookups in a UDDI (Universal Description, Discovery, and Integration) directory.

There is one exception where endpoint references can appear in BPEL. Given that resolution of a concrete partner link's endpoint reference might be part of the business logic, BPEL allows the dynamic computing of such endpoint references and their assignment to a partner link as part of the business process model by means of assign activities (described later). Endpoint references may also be exchanged between business process instances by sending them either explicitly as a message part or implicitly as message context, for example, in a SOAP (Simple Object Access Protocol) header.

Handling Web service messages— data flow. Data are passed between BPEL activities using *variables*. Variables are typed either by complete WSDL messages that allow for the storage of an entire input or output message of a Web service operation, or by XML schema types that allow for the storage of individual entities that are relevant for the business process. Variables are scoped either by the business process itself (providing for global variables) or by a BPEL scope (a group of activities that forms an activity in itself, providing for local variables).

When a process is instantiated through a receive activity, the supplied data are stored in a variable. For each invoke operation that is part of the process, the input message of the invocation originates from a variable, and the result of the invocation is written back to another variable. Thus, data flows through the process by writing and reading variables.

BPEL provides *assign* activities to update variables from within a process. Through assign activities, results of one or more service invocations can be used for the invocation of a subsequent service invocation by creating the input message it requires. Figure 3 shows the assign activities needed to prepare data for the first few activities of the travel booking process, and also shows the concrete data flow implemented by the assign activity preparing the data for the reply activity.

As a special case, assign activities can also be used to assign partner link endpoint references to variable fields and variable fields containing endpoint references to partner links, providing the ability to treat endpoint references as data where needed. In the travel booking process, as part of the invocation of the book operation, the traveler could pass an endpoint reference to a third party that should receive the final schedule information, rather than the traveler himself. An explicit assign step in the process would then retrieve the message field containing the endpoint reference and assign it to the appropriate partner link through which the process sends out the final schedule.

Long-running processes and correlation. An instance of a business process represents the complete course of action needed to perform a concrete set of interactions with one or more partners in order to satisfy a certain business event, such as booking travel. These interactions can happen over a long period of time, be it a number of days or even months and years. For a certain request issued by a partner, it is necessary to distinguish whether a new business process is to be instantiated to satisfy that request, or whether the request should be directed to an already existing instance, and if so, to which particular one.

BPEL allows identifying business process instances by reusing existing identification information that is already passed in existing business messages, such

Figure 4   Correlation set representation—A. WSDL example and B. BPEL example

A

```
1  <bpws:property name="confirmationNumber" type="xsd:integer"/>

2  <bpws:propertyAlias  propertyName="travel:confirmationNumber"
                        messageType="travel:bookOutputMessage"
                        part="confirmationNo" query="/"/>
3  <bpws:propertyAlias  propertyName="travel:confirmationNumber"
                        messageType="travel:cancelInputMessage"
                        part="confirmationNo" query="/"/>
```

B

```
4  <process name="TravelBookingProcess"...>
      ...
      <correlationSets>
5        <correlationSet name="TravelBooking"
                         properties="travel:confirmationNumber"/>
      </correlationSets>

      <flow>
         ...
6        <reply partnerLink="traveller"
                portType="travel:TravelAgentPT" operation="book"
                variable="output">
7           <correlations>
               <correlation set="TravelBooking" initiate="yes"/>
            </correlations>
         </reply>
         ...
8        <receive partnerLink="traveller"
                  portType="travel:TravelAgentPT" operation="cancel"
                  createInstance="no" variable="cancelInput">
9           <correlations>
               <correlation set="TravelBooking" initiate="no"/>
            </correlations>
         </receive>
         ...
      </flow>
   </process>
```

as customer IDs and order numbers. As part of a WSDL extension defined by BPEL, *correlation properties* and their location within WSDL messages *(property aliases)* can be specified. Multiple correlation properties are then combined into *correlation sets*, which effectively are keys for business process instances.

Figure 4 shows the correlation set definition and its usage for the travel booking process. We use a confirmation number to identify specific instances of the process. In A, as part of the WSDL specifying the book and cancel operations, we define a property "correlationNumber" of type string in line 1, and two associated aliases, which identify the location of that confirmation number in the output message of the book operation (line 2) and the input message of the cancel operation (line 3), the two places where the confirmation number is actually used.

With these definitions in B, a BPEL correlation set is defined as part of the travel booking process, consisting solely of the confirmation number (line 5). This correlation set is used in the reply activity from the book operation of the travel booking process (line 7), and in the receive activity for the cancel operation (line 9).

Note that the correlation set is actually initiated by the reply activity, effectively assigning the confirmation number that is part of the outbound message of the book operation as a key to the process instance. In the inbound case of the cancel operation, the associated receive activity uses the already established correlation set as the key to enable the business process engine to locate the correct process instance.

In general, if an existing instance can be located for a receive activity, the message is routed to that instance. If no instance can be located, a new one is created, provided the *createInstance* attribute of the receive activity allows that by having a value of *yes*. If multiple instances qualify, then a runtime exception is raised, indicating an ambiguity caused by a correlation set that is not specific enough.

A BPEL-based programming model for J2EE and WebSphere

With BPEL, business processes can be specified in a platform-independent manner. Intentionally, the BPEL specification itself does not address mapping of BPEL processes to any particular runtime platform, be it standard (such as J2EE) or vendor-specific.

In order to actually run BPEL business processes on a J2EE application server such as the WebSphere Application Server, a mapping of BPEL artifacts to J2EE artifacts is required. In this section we describe such a mapping. In addition, we introduce a number of extensions to BPEL that allow business processes to directly exploit both J2EE standard features and WebSphere-specific features on top of the standard.

We have seen how Web services define a virtual component model for the usage of components, in a vendor-neutral form. One obvious model of providing such components is by implementing the components that use the J2EE programming model, rendering them as enterprise beans, and running them on a compliant application server.

To run BPEL processes using a J2EE application server, they are rendered in WebSphere as EJB-based components. This rendering also allows native Java-based applications to access and interact with those business processes. In addition, EJB-based components are available as Web services through the usage of rendering layers such as the Java API for XML-based Remote Procedure Call[9] (JAX-RPC) (for SOAP/HTTP, that is, Simple Object Access Protocol using HyperText Transfer Protocol), or message-driven beans (for SOAP/JMS, that is, Simple Object Access Protocol using Java Message Service, or other JMS-based protocols). As a consequence, business processes rendered as enterprise beans become available in those formats and protocols, too.

After BPEL-based business processes have been embedded into the J2EE environment, the potential need to natively support other J2EE features arises. This includes support for Java as an expression language (in addition to XPath, as provided by standard BPEL), or the ability to have BPEL variables defined by Java types. This ability allows a developer familiar with the Java language to use Java features seamlessly from within business processes, eliminating the gap between the Web services world and the Java world and thus enhancing developer productivity. We use the extensibility mechanism built into BPEL to add new J2EE-related constructs in a compliant way.

Finally, the WebSphere Application Server provides a number of unique features that are beyond the J2EE standard, such as support for directories containing information about the people in the enterprise and their organization, or support for extended transactions including compensation support. To allow the exploitation of these features by business processes, we also define a number of WebSphere-specific extensions to BPEL.

For easier reference, the language resulting from adding those extensions to the core BPEL language is called BPEL+ in this paper. Obviously, using these extensions makes a business process WebSphere-specific—benefits and drawbacks of using any extended features must be carefully balanced. Although on the one hand their use reduces portability of a business process, on the other hand it allows for a tighter integration with the underlying J2EE and WebSphere platform and easy exploitation of its capabilities. This is the usual design trade-off when balancing standard against proprietary features.

The following sections describe the BPEL+ extensions in more detail. First, we look at extensions that allow the incorporation of human tasks into business processes. Then, we show how BPEL processes are represented by enterprise beans, followed by extensions to allow the direct use of Java interfaces, methods, beans, and expressions in processes. We complete this section by describing the structure of enterprise applications that contain processes.

Involvement of people in business processes. Although the complete automation of business processes is a very important goal, business processes frequently involve humans. Either manual steps are needed on the regular path of a business process (such as an approval), or humans are needed to handle exceptional situations if normal execution of the business process fails.

In its current specification, BPEL does not allow the definition of human-based activities, nor does it allow assigning humans to be responsible for the business process as such. BPEL+ defines extensions to do both.

Figure 5 shows an excerpt of the travel booking process in which a manual step has been inserted to handle the exceptional case where automatic billing via the traveler's credit card failed to work. The idea here is that rather than canceling the process and invoking compensation on all the already performed bookings, it is more efficient to let a person actually call the client and attempt to fix the problem with the credit card.

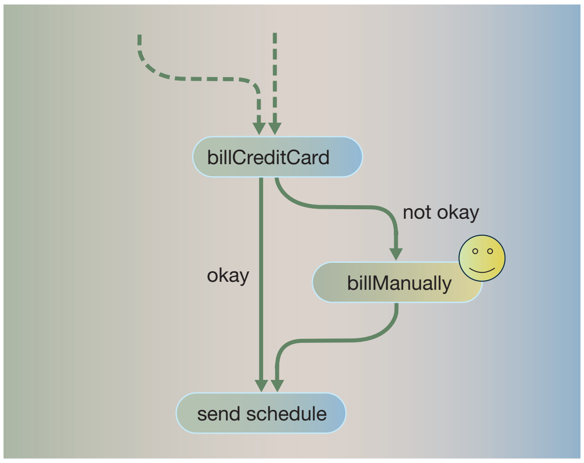
From the point of view of the business process, a human-based activity (also known as *staff activity*) is a step in the process where the associated Web service is not implemented by a piece of software, but is "implemented" by an action performed by a human. This step requires different processing.

When a human-based activity is performed, *work items* are created for it and distributed to all persons eligible to perform it. Eventually, one of the eligible users decides to work on the activity by claiming it. Claiming the activity provides the user with unique access to it. He or she can read the input data, perform whatever actions are needed (such as calling the traveler), and create the resulting data of this work as the output message (or a fault) of the activity. That completes the activity, and navigation of the business process continues.

It is the business process modeler who decides which users are assigned to perform a certain activity. Rather than assigning users through their user IDs, assignment queries against an organizational directory are used, involving properties such as group or role membership, or more fine-grained responsibilities. This approach allows the business process logic to be separated from the organizational aspects of the enterprise.

Syntactically, human-based activities in BPEL+ are also treated as special invoke activities. In particular, they have a signature defined by a port type and operation, and they read an input and produce an output variable (or fault). However, staff activities do not refer to a partner link, but instead define a set of potential owners by means of a staff assignment verb. Actually, BPEL+ uses elements from the staff support service of WebSphere to define those verbs. The definition of the "billManually" activity shown in Figure 6 (line 4) assigns all persons who are members of both the "Accounting Clerk" role and the "Travel Booking" role to the activity (line 6).

Figure 5 Travel booking process excerpt with manual step



Users participating in business processes receive work items for many activities from many business processes. A very common user interface for interacting with those work items and activities is by means of a browser-based Web client based on a portal, called the *process portal*. A central place is the work item list: A user can query for all work items that are currently assigned to him or her, applying filters on the type of activity or its current state and sorting work items by priority or age. From this list of work items, the user then selects one to work with the associated activity.

To interact with a selected activity, the minimal user interface displays the input message of the activity and allows the user to enter the resulting message and press a "complete" button. In more typical cases, however, customized user interfaces are associated with an activity. For instance, a complete portal page configuration can be specified as the user interface

Figure 6 People involvement in a process—BPEL+ example

```
   <process name="TravelBookingProcess"...>
       ...
1       <bpelp:administrator>
          <staff:membersOfRole roleName="Travel Booking Seniors"/>
       </bpelp:administrator>
       ...
       <flow>
          ...
2         <invoke name="bookFlight" partnerLink="flights"
                 portType="FlightReservationPT" operation="bookFlight"
                 inputVariable="bookFlightInput"
                 outputVariable="bookFlightOutput"
3.               bpelp:continueOnError="no"/>
          ...
4         <invoke name="billManually" partnerLink="bpelp:null"
                 portType="TravelAgentPT" operation="billManually"
                 inputVariable="billManuallyInput"
                 outputVariable="billManuallyOutput">
5.           <bpelp:staff>
                <bpelp:potentialOwner>
6                  <staff:membersOfRole role1="Accounting Clerk"
                                        role2="Travel Booking"/
                </bpelp:potentialOwner>
             </bpelp:staff>
          </invoke>
          ...
       </flow>
   </process>
```

of an activity, providing the user not only with the user interface for the activity itself, but also with support portlets required to effectively work on the activity.

In addition to normal users who are involved in business processes as participants, a second group, called *process administrators*, can be assigned to business processes. Process administrators are responsible for the administration and successful execution of running processes. Business process instances, in particular those that are long-lived, represent important assets of an enterprise. Being able to successfully complete their execution, even in the presence of unexpected faults, is thus an important goal. However, in many cases, automatic handling of all possible faults as part of the business process logic is not feasible. To deal with those cases, BPEL+ allows assigning process administrators to a process instance. This is a person or a group of people who are notified if anything goes wrong during execution of the process. Rather than automatically terminating and compensating the process, a process administrator is given the chance to manually repair the origin of the fault and then continue execution of the process. Process administrators are assigned to a process by means of a staff assignment expression based on group or role membership or some other organizational criteria, identical to those for staff activities.

BPEL+ also introduces the ability of an activity to stop processing and enter "manual repair mode." When this option is enabled for an activity, a fault that is raised by the activity and not caught locally is not propagated to the enclosing context. Rather, the activity is put into an "in error" state, and the process administrators receive a special work item for that activity, informing them about the error state. Using that work item, a process administrator can then inspect the failing activity. Repair is done by either retrying the activity or manually completing it, providing its resulting data. In either case, execution of the process can continue.

The example in Figure 6 shows the staff assignment for the travel booking process (line 1) and the definition of the bookFlight activity, with the option for manual repair enabled (line 3), indicating that there is no automatic continuation in case of an unhandled fault (the default for this option is "yes," the standard BPEL behavior).

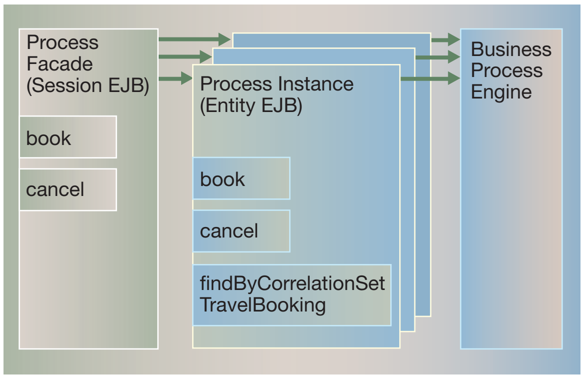
Mapping BPEL processes to J2EE. In general, a BPEL process is long-running, and its instances are stateful, that is, their variables, partner links, and correlation sets need to be stored persistently on disk, such as in a database. Hence, a business process in general is represented as an entity bean. Global variables of the process become fields of that entity. Likewise, partner links and correlation sets become fields of the entity. For each correlation set, finder methods are created on the home interface of the entity. The finder methods can be used to locate a particular instance of the entity based on the values of a particular correlation set.

The representation of the business process instance as an entity bean is completely transparent to its clients. The actual component a client uses to interact with the process is an associated stateless session bean, the facade session bean of the process. This session bean provides methods for the union of all operations that the BPEL process declares in its "my role" specification of all its partner link types. The facade session bean is responsible for managing the associated entity instances, dispatching requests based on the actual correlation set information. Figure 7 shows the facade session bean and the (potentially many) process instance entity beans for the travel booking process.

Actual execution of the business process is done by a combination of compilation and interpretation techniques, which are described in a later section of this paper detailing the architecture of the process execution infrastructure.

Using Java interfaces. BPEL+ makes a number of Java features accessible directly from the business process specification to simplify creating business processes for people familiar with Java. Included is the ability to directly use Java interfaces and methods. Rather than having to describe those interfaces and methods by WSDL port types and operations applying a JAX-RPC mapping, they can be referenced natively. This is true both for outbound calls (i.e., for invoke activities calling Java components) and for inbound calls (i.e., for receive, reply, and pick activities implementing Java methods).

Figure 7 Process facade session bean and process instance entity bean


Process Facade (Session EJB)
book
cancel
Process Instance (Entity EJB)
book
cancel
findByCorrelationSet TravelBooking
Business Process Engine

A typical example is the usage of an invoke activity to call a method of an enterprise bean. With BPEL+, the partner link type used in that invoke can make use of the already existing Java interface implemented by that enterprise bean, and the invoke activity can directly invoke a method of that interface. Figure 8 shows an example where the checkCustomer operation is implemented in that way. As a convention, we introduce a special name space identified by a java: prefix to designate Java artifacts. Note the BPEL+ shortcut notation for partner links where one of the roles is empty, which implicitly defines the appropriate partner link type (Figure 8, line 2). The partner link is mapped to a reference of the process entity bean, that is, to an entry in java:comp/env. This reference can then be bound to an actual "endpoint reference," which in the case of an enterprise bean is its standard JNDI (Java Naming and Directory Interface) name. This allows a seamless integration of J2EE components into BPEL+ processes. The invocation of the method for the enterprise bean (Figure 8, lines 4 and 5) is no different from that of a Web service.

In addition to invoking Java components, BPEL+ also allows a business process to provide its "my role" interfaces using Java interfaces. Thus, a process can natively implement a Java method, where its initiating receive activity consumes the input parameters and its final reply activity produces the output parameters or throws a Java exception.

Java variables and Java code snippets. BPEL supports process variables typed by WSDL messages or by XML schema types. Additionally, BPEL+ allows

Figure 8 Java extension for a process—BPEL+ example

```
1 <process name="TravelBookingProcess" xmlsns:java="..." ...>
     ...
       <bpelp:partnerLink name="servicesEJB">
2          <bpelp:partnerPortType name="java:com.travelAgent.Services"/>
       </bpelp:partnerLink>
     ...
3      <variable name="price" type="java:java.math.BigDecimal"/>
     ...
4    <invoke name="checkCustomer" partnerLink="servicesEJB"
5              portType="java:com.travelAgent.Services"
               operation="checkCustomer"
               inputVariable="checkCustomerInput"
               outputVariable="checkCustomerOutput">

       <source linkName="check2reply">
6          <bpelp:transitionCondition language="Java">
               return getCheckCustomerOutput() .isOkay() ;
           </bpelp:transitionCondition>
       </source>

       <source linkName="check2fault">
7          <bpelp:transitionCondition>
               <bpelp:otherwise/>
           </bpelp:transitionCondition>
       </source>

     </invoke>
     ...
8    <invoke name="addPrice" ...>
       <bpelp:script language="Java">
           BigDecimal newPrice = getPrice() ;
           newPrice.add(getHotelPrice()) ;
           setPrice( newPrice )  ;
       </bpelp:script>
     </invoke>
    ...
  </process>
```

variables to be typed by Java classes. For practical purposes, classes that are used as the variables of a process should adhere to the Java bean coding conventions. In particular, if they are used in long-running processes, they must implement java.io.Serializable, because they are persisted as part of the process state.

A Java variable is defined as any other variable, again using the special name space reserved for Java artifacts. Figure 8 shows an example for the definition of the price variable in the travel booking process (line 3).

For Java variables in particular, but also for other variables, a form of the standard assign activity is introduced by BPEL+ that allows the actual assignments to be coded in Java. This type of activity is called a *Java snippet*. Java snippets contain pieces of Java code that run inline with the execution of the business process, under the control of the process execution infrastructure. As such, Java snippets have access to the entire process context; for example, they can read and update arbitrary variables or partner links.

Java snippets actually run inside the J2EE context of the entity bean representing the process instance. Therefore, they have access to the variables of the process by accessing the fields of that entity, using standard get and set operations. Although Java snippets typically are used similarly to assign activities, they are not limited to reading and writing variables. Any code that is allowed to run inside an enterprise bean can also run in a Java snippet, so a Java snippet might actually do more advanced things, such as obtaining a value from a database or sending out a mail message.

Java snippets are syntactically rendered as an extension of invoke activities. They are bodies of implied methods that neither take nor return arguments, but might raise an exception in a form that can be caught by a BPEL fault handler as usual. Figure 8 has an example of a Java snippet that increases the value of the price variable (line 8).

An advantage of treating a Java snippet as a special case of an invoke activity is that both fault handlers and compensation handlers can be added to it. BPEL requires the specification of partner link, port type, and operation: because they are not needed for Java snippets, they are specified with dummy values.

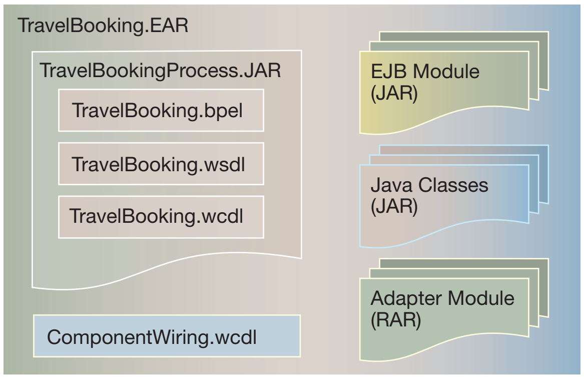
Conditions appear in a number of places in business processes: as transition conditions on control links of a flow, as join conditions on activities that are the target of links, and in switch and while activities. The BPEL default language for these conditions is XPath. As an extension, BPEL+ again allows the specification of conditions in Java. Actually, Java conditions are very similar to Java snippets in regard to their context, and hence, their programming model is the same as for Java snippets. Java conditions do differ from Java snippets, however, because conditions are bodies of methods that take no parameters, return a boolean value, and are not allowed to raise an exception.

As an example of a Java condition, consider the transition condition of the control link from the checkCustomer activity to the reply activity in Figure 8, line 6. This link must be followed if and only if the customer check was successful.

Figure 8 also shows another BPEL+ extension, namely the ability to specify "otherwise" transition conditions, which evaluate to true if and only if all of the explicit conditions originating from the same activity evaluated to false (line 7).

Enterprise applications with business process components. A business process itself is only one part of an application. Artifacts are also needed to specify the actual components that the process invokes, either by referencing an existing application or service (e.g., via an adapter that complies with Java Connector Architecture or via an endpoint WSDL), or by providing an implementation (e.g., as a session bean or Java class). As part of our mapping of BPEL processes to J2EE, we thus also have to describe how processes can be integrated into the packaging model.

Figure 9 Enterprise application archive with business process



The J2EE packaging model for applications is the Enterprise Application Archive (EAR) file. An EAR file contains Java archive (JAR) files for all the involved components of the application. WebSphere extends the notion of a JAR file so it can contain not only Java files but also BPEL+ files, as shown in Figure 9 and discussed in more detail in the section "Developing business processes."

There are a number of other artifacts in the EAR file that are worth mentioning. The TravelBookingProcess module not only contains the BPEL+ file with the specification of the process. It also contains an associated WSDL file that contains the definitions for all messages, port types, operations, correlation properties, and partner link types used by that process. Also, there is a *component description file* (WCDL, for WebSphere Component Description Language), which can be viewed as a generalized deployment descriptor for components such as business processes.

Such a component description specifies a component in terms of the interfaces that it provides (expressed either as Java interfaces or as WSDL port types), the references to other components that it needs (again, typed by either Java interfaces or WSDL port types), the references to other artifacts such as data sources that it needs, the quality of service properties that should be provided for it by the WebSphere run-time environment (such as transactions, compensation spheres, or method authorizations), and finally, its actual implementation (such as a BPEL+ process, an enterprise bean, or an adapter to a backend application).

Typically, an enterprise application contains multiples of those components that reference each other. A *component wiring file*, which is also shown later in Figure 14, captures the dependencies between components by connecting references to components.

Developing business processes

BPEL business processes are composite Web services that are assembled from other Web services. Conditions, representing business rules, define the order in which services are offered to or used from the outside world. WSDL port types are used to specify the interface of the services that are provided or consumed by a business process.

As discussed earlier, Web services provide a virtualization layer over many kinds of different implementations, for example, Java programs or Enterprise JavaBeans, and can therefore be considered a virtual component model. Composite and elemental Web services form a two-level programming paradigm. The business process level provides a flexible means for aggregating Web services and describing the interactions between them, including business-to-business protocols. The elemental services are implemented using standard programming languages such as Java, and hosted in application server platforms such as WebSphere.

The tools for workflow-based applications therefore include Java editors to develop low-level artifacts as well as editors to develop process artifacts. Which types of operations the process provides or consumes is specified on the business-process level. In the process model, the actual service endpoints that offer a service and the corresponding binding to their implementation infrastructure are not determined. It is the responsibility of the business process deployer to provide the binding between the service interfaces and their concrete WSDL or Java endpoints before a process becomes executable.

Business process modeling. Tools for the development of BPEL process models are provided by IBM and other vendors, both on a business process modeling level and on a technical level. In this paper, we focus on the latter, including the support for IBM extensions for the BPEL language, that is, BPEL+, which has been described in previous sections.

BPEL+ business processes are developed with WebSphere Studio.[10] The business process editor provided by WebSphere Studio is well-integrated with other editors like those for Java, J2EE, Web services, XML, XML schema, and so forth.

WebSphere Studio follows the two-level programming paradigm described previously. The user develops J2EE components with a Java editor. The J2EE components can then be offered as Web services and composed into business processes that represent higher-level services. Both J2EE components and business processes can be tested and debugged in the WebSphere Studio Unit Test Environment (UTE), which is in itself a complete J2EE environment.

Within the graphical WebSphere Studio process editor, the developer can drag and drop different kinds of BPEL+ activities into the process model canvas. Web services, called via invoke activities, can be combined with other types of BPEL or BPEL+ activities, and control links and conditions can be applied to control the order of Web service invocations.

For each invoke, receive, reply, and pick-onMessage activity, the process modeler specifies which partner link is associated with the activity. The name of each partner link is later associated with a particular Web service or Java endpoint. The resolved endpoint is either used to invoke a Web or Java service or to provide a Web or Java service to the outside world.

Business process data are represented by BPEL or BPEL+ variables, which can be defined with WSDL messages, XML schema types, or Java types. After the process variables have been defined, the BPEL activities can use the variables, for example, as request or response messages of invoke activities.

Business process deployment. When BPEL+ business processes are deployed into the WebSphere Application Server runtime infrastructure, the process model constructs are mapped to J2EE components as described previously. Mapping rules based on those defined by JAX-RPC[9] are applied when J2EE artifacts are created from corresponding BPEL+ language elements. Table 1 gives a detailed overview of this mapping for the various constructs involved. BPEL+ constructs such as custom properties are ex-

Table 1 Mapping of BPEL and BPEL+ language constructs to J2EE artifacts

BPEL or BPEL+ Construct	J2EE or Java Artifact
Process name space	Enterprise bean package name
Process model	Entity bean
Synchronous process interface	Stateless session bean facade
Asynchronous process interface	Message-driven bean facade
Receive or pick-onMessage activity	Enterprise bean method
Message with parts	Enterprise bean method signature, parameters
Variable	Entity bean field
Correlation set property	Entity bean field, used by finder methods
Correlation set	Entity bean finder method
Partner link	Local (java:comp/env) JNDI name for dynamic endpoint lookup
BPEL+ inline Java activity	Enterprise bean method
BPEL+ Java condition	Enterprise bean method
BPEL+ custom properties	Enterprise bean environment entries

plained below. Names for generated enterprise bean interfaces and methods are derived from BPEL process names, WSDL port type names, and WSDL operation names. Names of enterprise bean fields are derived from BPEL variable and property names. Additional enterprise bean methods are generated from BPEL+ inline Java code.

For all business processes, a facade stateless session bean is created that represents the intended interface for all EJB applications interacting with the process (see Figure 10, line 1). For each receive or pick-onMessage activity, a method on the facade session bean is generated (line 2).

For business processes that are executed as long-running, interruptible processes, an entity bean is created that holds the process instance state (see Figure 11, line 1). Variables of the business process hold the runtime data of a process instance. The entity bean has the same methods as the facade session bean (line 2). In addition, getter and setter methods for variables are available for use by Java script activities or Java conditions. BPEL properties that are used in correlation sets are mapped to individual fields on the entity bean (lines 3 and 4).

As a consequence, the extraction of a correlation set for a received message is performed by simply calling a finder method on the entity bean home interface (see Figure 12). The session bean methods invoke entity bean finder methods (line 4) and, optionally, the create methods (line 2) if no process instance is found and createInstance="yes" is specified for the BPEL activity. This approach completely hides the process instance correlation and instance creation from the partner that is sending a message to a process, which is in line with the BPEL paradigm of implicit instance creation.

Finally, this mapping to standard enterprise bean find/create methods also caters to process models with multiple receive createInstance="yes" activities (message rendezvous). Multiple messages correlated to the same process instance may arrive in any order, and, regardless of the arrival order, only the first arriving message must lead to the creation of a new process instance. The messages subsequently arriving must "join" the existing process instance.

BPEL+ processes are deployed into the WebSphere Application Server runtime platform in order to exploit platform-provided QoS attributes for their execution. At the same time, the IBM extensions for the BPEL language provide capabilities to easily integrate business processes with existing Java and J2EE applications. As was discussed earlier, the BPEL+ process modeler may specify Java interfaces for invoked or provided operations instead of WSDL port types. In addition, Java code may be inserted directly into the process model for simple calculations or data transformations. Furthermore, the modeler can specify conditions for the process logic in Java. Both Java activities and Java conditions are deployed as enterprise bean methods. This approach provides a well-defined runtime environment for the inline Java code. The Java programmer can exploit all J2EE platform capabilities in the same way as for every other EJB application. This includes all options provided for EJB deployment, such as those for transactional behavior or security.

Figure 10 Generated process session EJB (remote interface)

```
   package travelBooking;
   import ...;

1  public class TravelBookingSessionBean
            implements javax.ejb.SessionBean {

2        public OperationResult book( WSDLMessage input ) { ... }
         public void cancel( WSDLMessage cancelInput ) { ... }
         ...

   }
```

Figure 11 Generated process entity EJB (remote interface)

```
  package travelBooking;
  import ...;

1 public class TravelBookingBean
           implements javax.ejb.EntityBean  {

2       public OperationResult book( WSDLMessage input ) { ... }
        public void cancel( WSDLMessage cancelInput ) {  ... }
        ...

3       public abstract java.lang.String
             getCorrelationSetTravelBookingPropertyConfirmationNumber() ;
4       public abstract void
             setCorrelationSetTravelBookingPropertyConfirmationNumber(
                    java.lang.Integer newConfirmationNumber ) ;
        ...
  }
```

Custom properties can be used to parameterize a business process. They allow a process modeler to add attributes to processes. Although the value for BPEL properties can only be derived from messages, the value of custom properties can also be set in the process model, during deployment of processes, or at runtime, for example, by Java snippet activities. Custom properties can, for example, be used as configuration properties to customize the behavior of ready-made processes when they are deployed or to assign characteristics, such as importance or high business value, that have a default value set at modeling time that is overridden for particular process instances at runtime when a certain condition is met. Custom properties are mapped to environment entries on the EJB deployment descriptor, they can be overridden by the process deployer, and they may be accessed by inline Java code in the business process. If the inline code requires access to separately managed resources, the corresponding resource references can be added to the deployment descriptor of the generated business-process enterprise bean (business process bean). Note that the deployment descriptor is generated from corresponding specifications in the WCDL component declaration. It is not intended that the deployment descriptor be changed later. Figure 13 shows a sample snippet of a generated EJB deployment descriptor for a business process bean.

Figure 12 Generated process entity EJB (remote home interface)

```
package travelBooking;
import ...;

public classTravelBookingHome
        implements javax.ejb.EJBHome {

      public TravelBooking create ( PIID piid )
            throws EJBCreateException;
      public TravelBooking findByPrimaryKey( PIID piid )
            throws EJBFinderException;

      public java.util.Collection
            findByCorrelationSetTravelBooking (
                  java.lang.Integer confirmationNumber )
            throws EJBFinderException;
      ...
}
```

As mentioned earlier, BPEL+ processes are deployed as part of standard J2EE applications. Figure 14 details the involved artifacts that are relevant during deployment. During deployment, new J2EE artifacts are generated and are put into the same EJB-JAR file that already contains the BPEL and WSDL (and potentially XML schema type) artifacts. During deployment, code generators map the process model to enterprise bean interfaces and classes, and to EJB deployment descriptors. This phase of the process deployment operates similarly to a build environment that takes source code artifacts and produces the executable application. This archive may contain multiple processes, together with all the other well-known parts of an EJB application, EJB client, or Web client.

In addition to the code generation, the process deployment binds BPEL partner link specifications to concrete service endpoints in order to enable the runtime to execute invoke activities referring to these partner links. Before instances of the business process can be created and executed, it has to be determined how and when the partner link binding is going to happen. This may happen at different points in time, such as:

1. Static binding. The service endpoint may already be well-known when the process model is created. In this case, the WSDL definitions with the service endpoints are provided together with the BPEL process definition in the enterprise application archive, and a separate component wiring file constitutes the static relationship between the BPEL partner link and the WSDL port.
2. Deployment-time binding. The endpoint statically associated with a particular partner link (see 1) may be overridden at deployment time.
3. Runtime binding via locator. As an alternative to a relationship with a fixed endpoint, a locator expression can be deployed that is evaluated at runtime, for example, a UDDI query expression.
4. Runtime binding via assignment. The service endpoint address is sent to the process instance as an endpoint reference (see Reference 8) and mapped to the BPEL partner link with an explicit assign activity.

The service endpoint or locator is made known to the runtime by adding it to the global JNDI name space. When a process instance is executed, the business process engine resolves the partner link via a JNDI lookup. This approach provides flexibility for changing the association without complete redeployment of the process, which is important if process instances may run for a very long time. The lookup returns a Java representation of the WSDL definition with the service endpoint address and the binding used for the invocation.

Besides endpoints used by the process to invoke Web services, the deployed process also provides a Web service (or Java) endpoint. Each inbound receive, receive-reply, or pick-onMessage activity of a de-

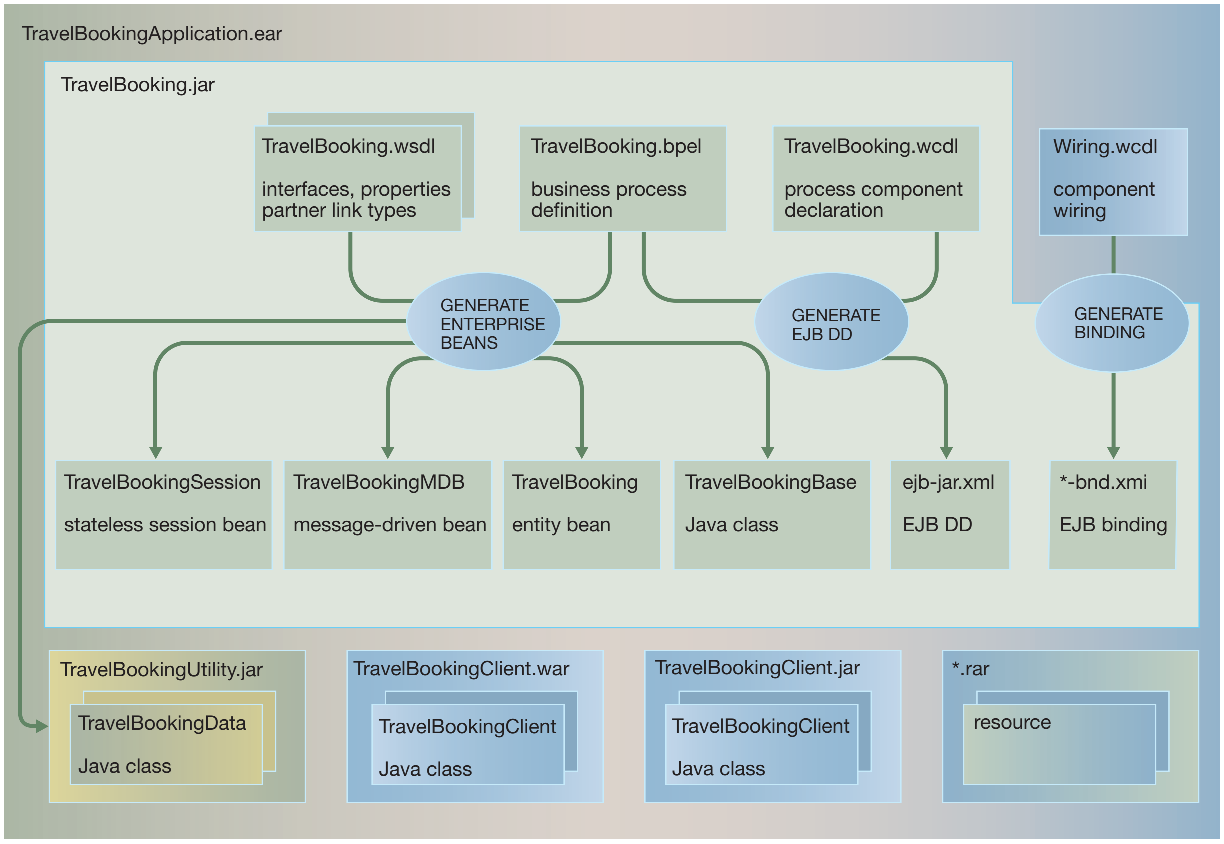
Figure 13 Generated process entity-bean deployment descriptor

```
<entity id= "TravelBooking">
 <ejb-name>TravelBooking</ejb-name>
 <local-home>TravelBooking.TravelBookingLocalHome</local-home>
 <local>TravelBooking.TravelBookingLocal</local>
 <ejb-class>TravelBooking.TravelBookingBean</ejb-class>
 <persistence-type>Container</persistence-type>
 <prim-key-class>java.lang.Integer</prim-key-class>
 <reentrant>true</reentrant>
 <cmp-version>2.x</cmp-version>
 <abstract-schema-name>TravelBooking</abstract-schema-name>
 <cmp-field id="CMPAttribute_1044271092444">
   <field-name>correlationSetTravelBookingPropertyConfirmationNumber
   </field-name>
  </cmp-field>
  <primkey-field>piid</primkey-field>
  <resource-ref id="ResourceRef_BPEL">
   <description/>
   <res-ref-name>jdbc/BPEL</res-ref-name>
   <res-type>javax.sql.DataSource</res-type>
   <res-auth>Container</res-auth>
   <res-sharing-scope>Shareable</res-sharing-scope>
  </resource-ref>
  <query>
   <description/>
   <query-method>
    <method-name>findByCorrelationSetTravelBooking</method-name>
    <method-params>
     <method-param>java.lang.Integer</method-param>
    </method-params>
  </query-method>
  <ejb-ql>select object(o) from TravelBooking o
   where o.correlationSetTravelBookingPropertyConfirmationNumber=?1
  </ejb-ql>
 </query>
</entity>
```

ployed process represents a service endpoint that can be consumed by a client or business partner. If a process model contains multiple asynchronous interactions with different business partners, it is sometimes necessary to dynamically determine the service endpoint that a business partner should use for the next interaction. Different partners may issue callbacks to the process via different endpoints. If the partner is local to the enterprise in which the process is running, the interaction may be performed with an endpoint that is only known locally and may have different QoS attributes than an endpoint used for business-to-business interactions. For this purpose, the runtime infrastructure determines the endpoint address of the process itself, which is intended to be used by a particular partner. The process may use an assign activity to map its own address that is supposed to be used by a particular partner to a variable and then send the address to that partner. The format for callback addresses is an endpoint reference or a global JNDI name in the case of a Java service. Similar to the lookup for invoked Web services discussed earlier, the process runtime infrastructure does a JNDI lookup for its own endpoint, which is wired to a partner link during process deployment.

When all artifacts in the enterprise application archive have been created, the standard WebSphere application installation process is initiated. The generated process beans are installed, and corresponding configuration repository entries are created. At

Figure 14 Enterprise application archive with generated process artifacts



the same time, the process model itself is translated into an executable process template and persisted into the business process engine database. The process template contains serialized objects optimized for the business process engine. Note that the same runtime representation is created from processes that have been modeled with the previous Version 5.0 of WebSphere Studio, which provides an upgrade path for existing installations of the business process engine.

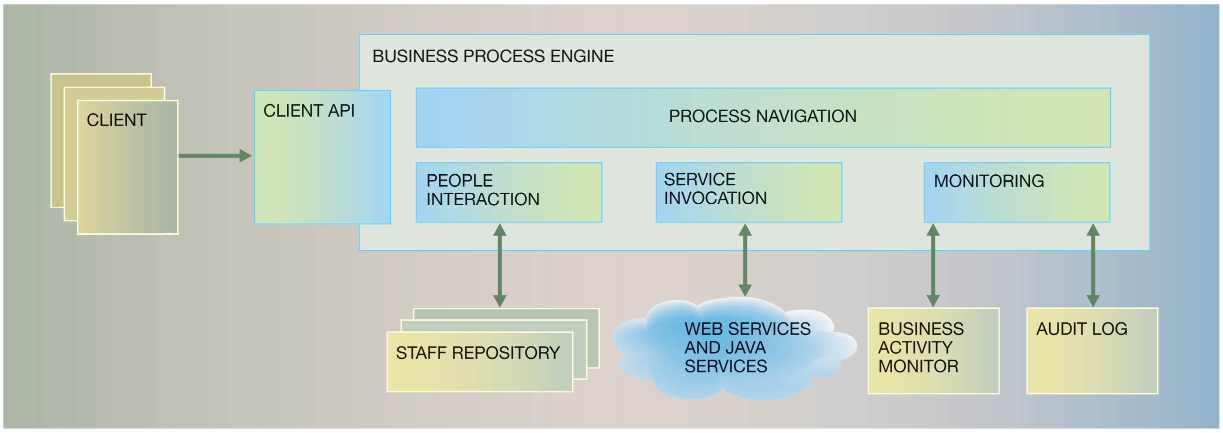
The complete sequence of artifact generation, deployment, and installation tasks can be performed either with or without human interaction. A typical scenario for an interactive process deployment would include manual steps for binding resource references to actual resources.

A process can be deployed into the application server runtime in one unattended step, that is, without user interaction, if all required service endpoints and the corresponding binding information are provided. This unattended deployment step may be initiated by a program. Automatic and unattended deployment is required for several e-business on demand* computing scenarios,[11] where processes have to be dynamically created, deployed, installed, executed, and de-installed.

Execution infrastructure for business processes

The previous sections described the programming model of BPEL and the extensions introduced by BPEL+, as well as the development and deployment of workflow-based applications. This section takes a look at the execution of business processes within the infrastructure provided by the process engine integrated in WebSphere. It first explains the architecture of the process engine and the application pro-

Figure 15 Business process engine architecture



gramming interface (API) it offers. Next, it talks about the different types of business processes that the engine supports and their respective characteristics. After having explained how the invocation of services is supported, the focus for the rest of the section is on the QoS aspects of workflow-based applications and the appropriate support by the WebSphere infrastructure. Performance, workload balancing, security, and high availability are the topics being discussed. The section closes with a subsection on business process engine topologies, summarizing the most important aspects of this section.

Architecture. We first look at the business process engine, which has been implemented as a special container, called the business process container. Figure 15 provides an overview of the business process engine that shows the major building blocks of the process engine and the major components with which it interacts.

The business process engine consists of the following building blocks:

- The client API used by client applications to start business processes and to interact with running business processes
- The process navigation component that navigates through the BPEL+ business process and decides which activities are to be executed in which order
- The human interaction component that interacts with external staff repositories to identify those who should work on a particular staff activity of a BPEL+ business process, and that provides the interfaces by which clients such as a process portal can work with the activities
- The service invocation component to invoke Web services or Java services
- The monitoring component that provides information for business activity monitoring tools or audit logs

Not shown are infrastructure components that are used by the process engine for its operation, such as the database that stores business process models and runtime information or the message queuing system that handles asynchronous messages.

Application programming interface. Clients typically interact with business processes using the enterprise bean facade interface specifically generated for each business process. As pointed out earlier, these specific interfaces are generated according to the port type or Java interface specifications of the business process and its partners.

An alternative way of accessing business processes is through a set of generic API verbs provided by the business process engine. Examples of those API verbs are *call* to start a process, *query* to retrieve a list of work items, or *claim* to start working on a staff activity (see Reference 12 for details). Enterprise bean as well as message rendering is provided for each of the different API verbs. The enterprise bean rendering, for example, is used by the Web client provided as part of the business process support in WebSphere. The Web client allows humans to start business processes and to interact with business process instances and activities.

Business process types. We use the transactional capabilities and process behavior that a business process exhibits as a means for identifying different types of business processes. The different transactional capabilities and process behaviors manifest themselves as different QoS characteristics of the appropriate workflow-based application.

Noninterruptible business process. The first type of business process is a noninterruptible business process. Noninterruptible means that the business process runs on one physical thread from start to end without interruptions. Noninterruptible business processes are also known as *microflows* or *micro script streams*.[13] As the name suggests, microflows are small in footprint and fast in execution. Microflows can have different transactional capabilities. A microflow can run within a distributed transaction, it can run as part of an activity session (see Reference 14, chapter 9), or it may not use transactions at all.

Microflows that run within a distributed transaction are a special case of atomic spheres.[13] They are not as restricted as atomic spheres in that they can also contain nontransactional activities. The global transaction ensures that all transactional activities of the microflow either succeed or are rolled back. Nontransactional activities do not participate in the global transaction; to preserve integrity they have to be undone using compensation.

Compensation support in WebSphere is integrated with transaction management. The WebSphere transaction log is used to store information required to undo activities, such as the name of the undo operation and the associated data. If a transaction is rolled back, then compensation is run as part of the rollback processing of the transaction manager.

Examples of nontransactional activities are activities that perform operations such as writing to a (nontransactional) file system, sending messages via SMTP (Simple Mail Transfer Protocol), J2EE connector operations where either the connector or the back-end system does not support transactional integration, or SOAP-based Web services (the latter will become transactional once the upcoming standard for Web services transactions[15,16] is broadly adopted).

Activity sessions provide an alternative unit-of-work scope to that provided by global transactions. An activity session can be longer-lived than a global transaction and can encapsulate global transactions. Activity sessions are used to scope or coordinate local transactions. They are used for microflows when it is not possible to use a global transaction, for example, when the microflow uses more than one resource that only supports one-phase-commit. Activity sessions provide a solution by coordinating the one-phase commit process. If an activity session rolls back, changes are undone.

Interruptible business processes. The second type of business process is an interruptible (or long-running) business process. Classic workflow systems have provided support for interruptible business processes for quite some time. A business process becomes interruptible when each step of the process is processed within its own physical transaction[17] (see Reference 13, chapter 4.7).

Navigation processing for an activity starts when the business process engine receives a navigation message in its navigation input queue. Each navigation message represents an incoming connector for an activity and carries the truth value for the connector indicating whether the connector has been evaluated to be true or false. The first action the process engine performs is to start a transaction; all subsequent actions are carried out under control of this transaction. Next, the process engine reads the message from the navigation queue. Then, the process engine reads the activity state from the database and checks whether all incoming connectors for the activity have been evaluated yet. If not, the activity state is updated with the truth value of the message, and the transaction ends. If all incoming connectors have been evaluated, the process engine evaluates the start condition of the activity. If the start condition evaluates to be true, the activity is executed. When the activity finishes, the truth value of all outgoing connectors is set to true (if the activity completes successfully) or false (if the activity completes in error), and a message is sent for each connector. Finally, the activity state is written to the database, and the transaction is ended. If the start condition of the activity evaluates to be false, processing depends on the setting of the suppressJoinFailure property of the activity. Standard processing as defined in BPEL[3] is to raise a fault. Changing the default to "handle join faults automatically" is achieved by setting suppressJoinFailure to true. In that case, the activity is skipped, and all outgoing connectors are assigned a truth value of false. If an activity is skipped, dead path elimination[6,13] is initiated; that means the truth value of false is propagated transitively along entire paths formed by consecutive links until a join condition is reached that evaluates to be true.

If a system error occurs, such as a deadlock in the database, the navigation transaction fails so that the original message remains in the navigation input queue, all database changes are undone, and the retry count is incremented. Aborting the navigation transaction and retrying the navigation transaction is carried out until either the message can be successfully processed or the retry count exceeds a configurable maximum number. If the retry count is exceeded, the message is moved into a hold queue. It will be up to an administrator to decide how to deal with the message. If the message is poisoned, that is, it contains invalid data that always result in an error condition when processed, the administrator can change the contents of the message and resubmit it to the navigation queue for processing.

Business processes need to be interruptible if they must wait for external stimuli or if they involve humans. Examples of external stimuli are events sent by another business process in a business-to-business interaction, responses to asynchronous invocations, or the completion of a staff activity. In these scenarios, running interruptible business processes is almost mandatory because the process engine cannot tie up resources waiting for the arrival of the external stimuli or the completion of the staff activity.

Service invocation. Activities in business processes interact with the outside world by receiving messages, sending messages, or by performing synchronous request-response operations on Web services and Java services. The business process references these services via partner links. At runtime, the partner links are used to find the associated services and then to invoke them.

A business process typically finds the services as described in the earlier subsection "Business process deployment." Alternatively, partner links can be bound to a service endpoint using an endpoint reference as defined by the Web Services Addressing specification.[8] For example, a partner has provided an endpoint reference as part of a request to the business process. The business process could then use this endpoint reference to send a message back to the partner or could even send the endpoint reference to another partner who could then use the endpoint reference to respond back to the original partner.

A business process invokes a service by exploiting the functions offered by a service invocation infrastructure. Examples of service invocation infrastructures are the Web Services Invocation Framework (WSIF),[18] JAX-RPC, or Java invocation. Using these infrastructures, the business process engine can invoke almost any kind of service. Concrete examples for services natively supported by WebSphere are Web services with a SOAP binding, enterprise services accessible through a suitable J2EE connector such as the connectors for Information Management System (IMS*), Customer Information Control System (CICS*), and SAP**, services with a message-based interface (JMS, IBM WebSphere MQSeries*), Java methods, or methods of an enterprise bean.

The incorporation of services not supported natively can be achieved by "wrappering" the services with a mechanism that is supported natively. Examples of those services are executables, native code libraries written in arbitrary programming languages, or database-stored procedures. Typically one would define those services as Web services using WSDL and then write the appropriate wrapper code.

Performance. The business process engine is built using the J2EE artifacts offered by the application server. It directly benefits from the performance characteristics of enterprise beans, message-driven beans, servlets, Java Database Connectivity (JDBC**) connections, and JMS connections.

Throughput and response time of a workflow-based application depend on the following factors:

- The type of business process used: Navigation costs of a noninterruptible business process (microflows) are much cheaper compared to interruptible business processes.
- The structure of the business process, such as the number of structured activities, the number of loops, the performance of services invoked by activities, and the amount of parallelism.
- The performance of components used by the process engine: Database and message-queuing systems with good performance characteristics have an immediate impact on the performance of workflow-based applications, particularly for interruptible business processes.

The performance of interruptible processes can be improved by giving hints to the process engine on how it can optimize the transactional behavior of business processes. Without any further optimization, each activity in an interruptible business process runs within its own physical transaction. By providing a transaction hint for an activity, the engine can select more flexible transaction boundaries, for example, by combining multiple activity invocations into one transaction. This reduces the navigation effort, resulting in better response time and throughput. A possible disadvantage of combining activities into one transaction is the increase in recovery costs; more work needs to be undone should an error occur. Another possible disadvantage is reduced concurrency because the larger transactions require resource locks to be held longer. Careful design decisions have to be made on the part of the modeler when specifying transaction hints.

To improve the response time of some business processes at the expense of others, a priority can be assigned with a business process template. The priority can be overridden at runtime for the individual business process instance, based on actual data with which the business process operates (such as "amount > 1M$") or by administrative action. The business process engine uses the priority as a hint to find out which business processes it should prefer during navigation.

Compiling business process definitions into executable code can further optimize the execution speed of business processes, even compared to the execution in a highly efficient business process engine, by reducing the number of instructions required for navigation. A compiled business process reduces the involvement of the process engine, in some cases even to the point where only a Java Virtual Machine (JVM*) is needed.

The performance advantages of compiling business processes are to some extent outweighed by reduced manageability and program maintenance, requiring substantially more administrative support. Despite these additional efforts, mainly for long-running processes, compilation works quite well for microflows. It reduces the overall number of instructions of the microflow so significantly that its performance comes close to hand-written Java code. This is less important for a microflow that invokes many remote services and performs extensive data mappings and calculations—the number of instructions required for navigation is small compared to the overall amount of instructions consumed. A more noticeable effect can be observed for microflows that script together Java operations and local EJB calls, in particular, if not only the business process is compiled but also the service invocations. Doing that ensures that simple service invocations such as a Java method call are performed inline in the compiled business process code instead of using an invocation framework such as WSIF.

For interruptible processes the disadvantages of compilation by far outweigh the benefits. Therefore, only streams within an interruptible process that are executed within one transaction are considered for compilation.

Workload balancing. Workload balancing is a core capability of the WebSphere Application Server. It ensures that incoming work requests are distributed to the application servers, enterprise beans, servlets, and so forth that can most effectively process the requests. Workload balancing occurs on stand-alone application servers and on application server clusters. Workload balancing is performed in several situations. For example, it occurs between application servers when a client sends an HTTP request from a Web-based client and when a request is sent to the enterprise bean, or it occurs between instances within one application server when an API message is sent to the message-driven bean API and when an internal message is received by the business process engine. Additional workload-balancing capabilities can be added to workflow-based applications by choosing an application topology that combines WebSphere clustering with clustered message queues. When clustered queues are used, workload balancing between application servers also occurs when sending messages to a message queue; we show an application topology that combines both WebSphere clustering and clustered message queues later in this paper.

Dispatching each activity of an interruptible business process to an arbitrary server is good from a workload-balancing point of view. It slightly increases navigation costs, though. A business process running within one application server is said to have *server affinity*. Affinity to an application server in WebSphere means affinity to a JVM process. If the business process engine can assume affinity, it can optimize execution of business processes by using advanced caching strategies to reduce the number of database operations; this improves performance of a particular process instance at the expense of reducing workload balancing. When one is to be preferred over the other depends on the requirements of the business scenario. The business process engine in WebSphere supports both goals by providing the appropriate tuning parameters.

Security. A critical QoS property is security, in particular for workflow-based applications where security not only means authentication of users and authorization of user requests accessing methods of a business process, but also resolution of staff queries to authorize users to work with an activity in a business process instance. Staff queries are defined by the modeler of the business process and executed at runtime to determine which persons are candidates for working on a particular staff activity. The result of a staff query is a set of persons. As a side effect of this mechanism, we obtain instance-based authorization.

If we assume that WebSphere uses the local operating system as its registry, this registry is used to authenticate client requests. When a business process client (Web client) connects to WebSphere, WebSphere enforces authentication of the client. The credentials entered by the client are checked against the entries of the WebSphere user registry, and if authentication succeeds, the client request is authenticated. The process engine stores both the caller principal name and the security context used to initiate the process as part of the persistent state of the business process instance. The caller principal is stored because the process starter has special rights when accessing a running business process. The security context is stored to allow enforcement of the same security constraints for subsequent activities running in separate transactions from the ones that applied to the process starter. This action effectively results in all activities of the process instance running on behalf of the process starter, even though they are invoked in separate transactions. Seamless and secure operation can only be ensured by reusing the proper security context.

To avoid manual synchronization of principals, the same repository should be used for staff queries and WebSphere user management. To support other than trivial staff queries (e.g., "is manager of") a rich-enough repository is required. WebSphere provides out-of-the-box support for Lightweight Directory Access Protocol (LDAP); other (maybe company-specific) repositories can be integrated using the respective plug-in points of WebSphere.

High availability. Making a WebSphere workflow-based application highly available[13] requires high availability (HA) support for all components involved: network dispatchers, HTTP servers, application servers, messaging system (see References 19 and 20), database (see References 21 and 22), and so forth. That means every component must be redundant, monitored for failure, and, in case of a failure, must no longer receive any requests. WebSphere workload management (WLM) provides out-of-the-box capabilities that improve application availability by supporting failover between application servers in a WebSphere cluster. If combined with HA software such as z/OS* Parallel Sysplex,[23,24] High Availability Cluster Multiprocessing (HACMP) on the AIX* operating system,[25] Microsoft Cluster Server (MSCS) on the Windows** operating system, or Sun Cluster on the Solaris** operating system, a WebSphere system can be made highly available.

Performance degradation as a result of component failures can be minimized by using appropriate HA software such as HACMP on AIX. HACMP not only takes over the IP (Internet Protocol) address (required to establish new network connections) to the backup system but also the MAC (Media Access Control) address of the failing system (thus keeping existing connections alive). As a consequence, pooled connections in WebSphere do not become stale, and workflow-based applications continue without performance degradation.

Business process engine topologies. We now take a look at some concrete topologies and analyze the capabilities that a particular topology offers for a workflow-based application. It should be noted that all of the following topologies can run the same interruptible and noninterruptible business processes. The only difference in topologies is the QoS that they are offering. The simple topology (see Figure 16) shows how the business process engine is integrated into WebSphere. The scenario uses a browser-based client interacting with a simple, nonclustered workflow-based application running in WebSphere. The business process engine uses the embedded message-queuing support provided by WebSphere and a database on a separate database server. The simple topology is used for smaller workflow-based applications with low throughput requirements. It provides simple load balancing within the application server. To further simplify the topology, clients, WebSphere, and database are run on one physical computer. WebSphere Studio uses this topology in its test environment.

The simple topology does not scale sufficiently for business process scenarios requiring higher throughput; a cluster topology using a stand-alone messag-

Figure 16 Simple topology

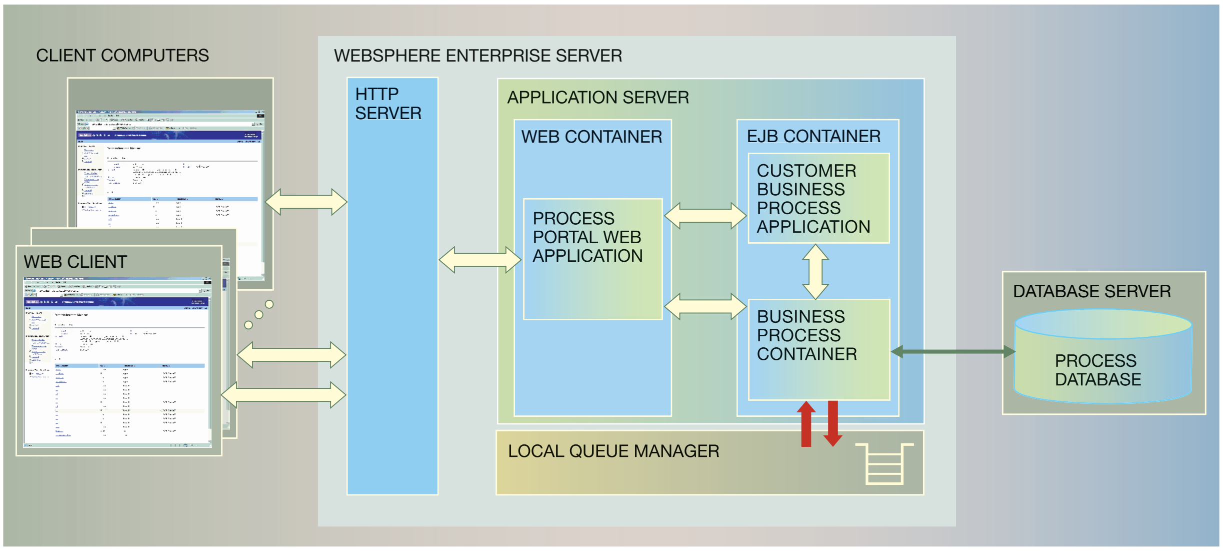


Figure 17 WebSphere Application Server cluster topology with stand-alone messaging server

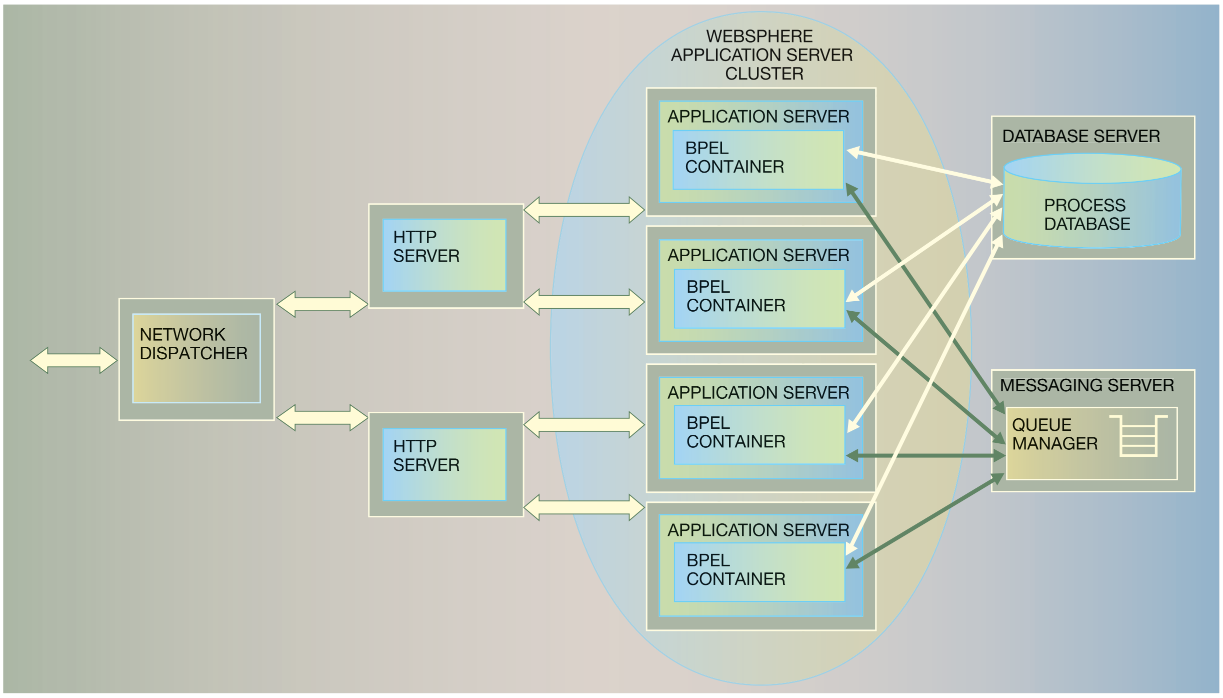
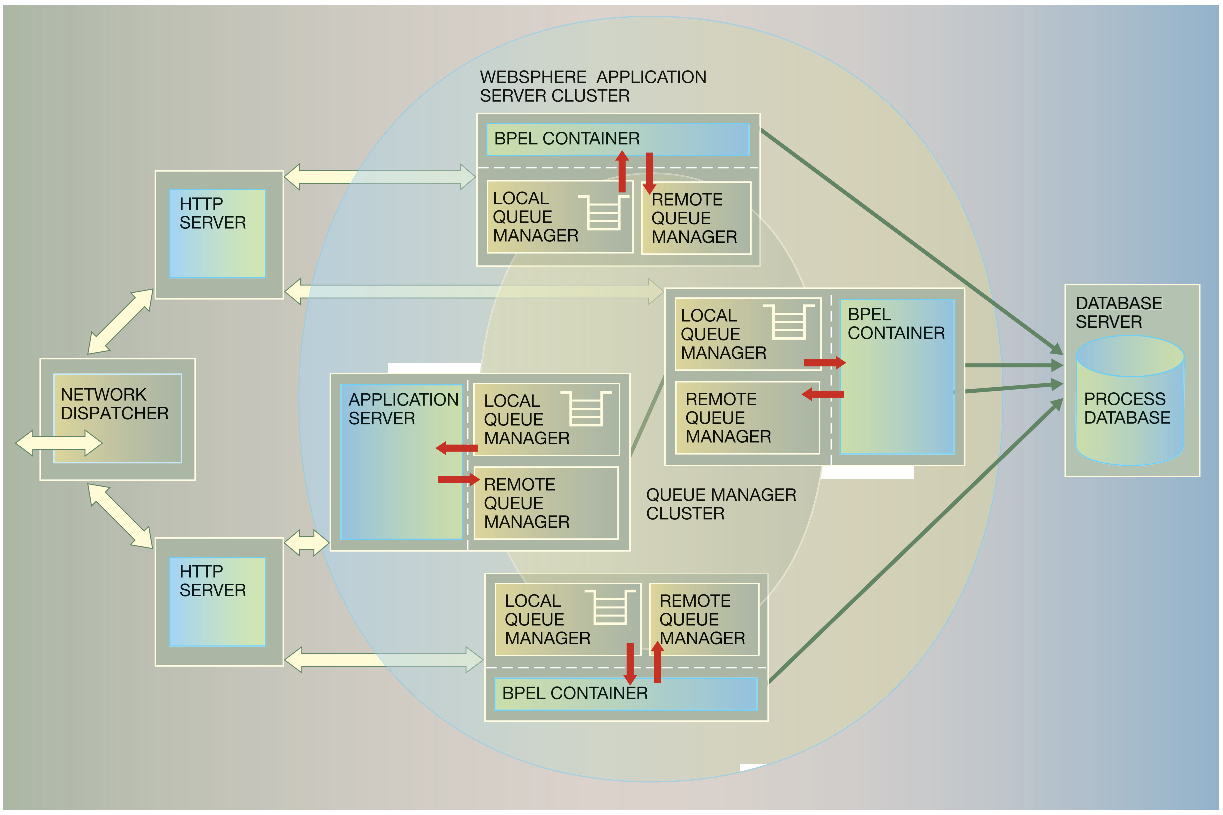


Figure 18 High-performance topology



ing server, as shown in Figure 17, is a better choice. Compared to the simple topology, the following changes have been made: A network dispatcher has been added to route requests to a set of HTTP servers that use a WebSphere Enterprise cluster. The business process engine is clustered as part of the application servers; that means the workload is shared by the various process engines. Because different process engines may be involved in handling a process instance, process instance information must be shared between the different process engines. Sharing is achieved by maintaining a remote database and a remote queue manager. Note that the clients sending the requests to the network dispatcher are omitted from the figure for simplicity.

The advantages of the cluster topology with a standalone messaging server are higher throughput and good workload balancing. The topology is scalable by adding WebSphere nodes to the cluster or by configuring additional servers with business process containers on existing nodes. Use of the latter is suggested on a powerful computer when the available resources are not sufficiently used by one server. Using a single remote queue manager makes getting and putting messages more expensive; also, the business process engine cannot ensure server affinity for a business process because there is no local queue manager assigned to each application server. Therefore, performance optimizations based on server affinity are not possible. The cluster topology for high performance as shown in Figure 18 defines what ultimately can be done to achieve maximum performance and scalability. Instead of a single messaging server, an MQSeries cluster is used that is interwoven with the WebSphere cluster.

The primary advantage of this topology is its excellent scalability. It also provides configurable workload balancing for running business processes. This is achieved by using MQSeries clustering. Each business process engine has a local queue manager with local queues assigned, allowing for more efficient reads. If a business process is running with server affinity, the process engine also uses the local queue for sending messages. If optimized workload balancing is required instead, a second queue manager is used. This queue manager has no local queues defined. It forms a cluster with all other queue managers. When the queue manager receives a message, it routes this message to a suitable queue in the cluster, and the associated business process engine reads the message from that queue.

For mission-critical applications, the processing of messages in a queue on a failing node should not be delayed until the node has been brought up again. To solve that problem, support for HA must be added to the individual components. To simplify the HA scope it is a good idea to separate the queue managers from the application servers as indicated by the dotted line in Figure 18. As long as there is still one queue manager for distributing messages and one for local access assigned to each application server, this has no impact on the tuning capabilities of the overall system, except that access to all queue managers uses the remote interface instead of local bindings, resulting in a slight performance penalty. Note that extending the above topology by using a database cluster instead of a single database is usually not required for workflow-based applications. The amount of data stored on behalf of the business process container even in large workflow-based applications can be handled easily by a single database server. A detailed discussion of this theme can be seen in Reference 13.

Summary

We have discussed how WebSphere provides process choreography support that is compliant with BPEL. We first showed how BPEL can be extended to provide for support of J2EE and WebSphere constructs; this enables developers familiar with the Java language to use Java features seamlessly within business processes, eliminating the gap between the Java world and the Web services world in the choreography space and thus enhancing developer productivity. Next, we described how business processes are developed using WebSphere Studio and how they are deployed into the WebSphere Application Server runtime infrastructure. Then, we presented the architecture of the business process engine and how it manifests itself to the outside world. We discussed in particular the QoS properties of the infrastructure, such as performance, high availability, and security. Finally, we presented some internal aspects of the business process engine.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Microsoft Corporation, or SAP AG.

Cited references

1. F. Leymann, "Web Services—Distributed Applications without Limits," *Proceedings of Database Systems for Business, Technology and Web BTW 2003*, Leipzig, Germany (February 26–28, 2003).
2. F. Leymann and D. Roller, "Workflow-Based Applications," *IBM Systems Journal,* **36,** No. 1, 102–123 (1997); also at http://researchweb.watson.ibm.com/journal/sj/361/leymann.html.
3. *Specification: Business Process Execution Language for Web Services (BPEL4WS) Version 1.1* (May 2003), http://www.ibm.com/developerworks/library/ws-bpel/.
4. R. Khalaf and F. Leymann, "On Web Services Aggregation," Fourth VLDB Workshop on Technologies for E-Services, *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003)*, Berlin (September 2003).
5. F. Leymann and D. Roller, *Business Processes in a Web Services World*, IBM Corporation, http://www.ibm.com/developerworks/webservices/library/ws-bpelwp/.
6. *Web Services Definition Language (WSDL) 1.1*, W3C Note (March 2001), http://www.w3.org/TR/wsdl.
7. F. Curbera, R. Khalaf, F. Leymann, and S. Weerawarana, "Exception Handling in the BPEL4WS Language," *Proceedings of the International Conference on Business Process Management (BPM 2003), Lecture Notes in Computer Science* **2678,** Springer-Verlag, Heidelberg (2003), http://tmitwww.tm.tue.nl/bpm2003/paper_curbera.htm.
8. *Specification: Web Services Addressing (WS-Addressing)* (March 2003), http://www.ibm.com/developerworks/webservices/library/ws-add/.
9. *Java APIs for XML-based RPC (JAX-RPC)*, JSR 101, Java Community Process, Sun Microsystems, Inc. (2003), http://jcp.org/en/jsr/detail?id=101.
10. F. Budinsky, G. DeCandio, R. Earle, T. Francis, J. Jones, J. Li, M. Nally, C. Nelin, Y. Popescu, S. Rich, A. Ryman, and T. Wilson, "WebSphere Studio Overview," *IBM Systems Journal*, **43**, No. 2, 384–419 (2004, this issue).
11. *Living in an On Demand World*, White Paper, On Demand Business Literature, IBM Corporation (October 2002), http://www.ibm.com/e-business/doc/content/literature/literature_ebusiness.html.
12. M. Kloppmann and G. Pfau, WebSphere Application Server Enterprise Process Choreographer—Concepts and Architecture, IBM Corporation (December 2002), http://www.software.ibm.com/wsdd/library/techarticles/wasid/WPC_Concepts/WPC_Concepts.html.
13. F. Leymann and D. Roller, *Production Workflow*, Prentice Hall, Inc., Upper Saddle River, NJ (2000).
14. R. High, E. Herness, K. Rochat, T. Francis, C. Vignola, and J. Knutson, *Professional IBM WebSphere 5.0 Application Server*, Wrox Press, Hoboken, NJ (2002).
15. *Specification: Web Services Transaction (WS-Transaction)*, (August 2002), http://www.ibm.com/developerworks/webservices/library/ws-transpec/.

16. *Web Services Coordination (WS-Coordination)*, IBM, Microsoft, BEA (September 2003), http://www.ibm.com/developerworks/library/ws-coor/.
17. F. Leymann, "Transaction Support for Workflows," (in German), *Informatik in Forschung & Entwicklung* **12**, No. 1, (1997).
18. D. König, M. Kloppmann, F. Leymann, G. Pfau, and D. Roller, "Web Services Invocation Framework: A Step towards Virtualization," *XMIDX 2003*, pp. 33–44.
19. G. Wallis, "MQSeries and High Availability, Part 1," *Mainframe Week*, Issue 20 (May 22, 2002), http://www.mainframeweek.com/journals/articles/0020/MQSeries+and+high+availability%2C+part+1.
20. G. Wallis, "MQSeries and High Availability, Part 2," *Mainframe Week*, Issue 21 (May 29, 2002), http://www.mainframeweek.com/journals/articles/0021/MQSeries+and+high+availability,+part+2.
21. M. Wright, An *Overview of High Availability and Disaster Recovery for DB2 UDB*, IBM Corporation (April 2003). http://www.ibm.com/developerworks/db2/library/techarticle/0304wright/0304wright.html?ca=dnp-215.
22. M. Rao, IBM DB2 Universal Database Clustering—High Availability with WebSphere Edge Server, IBM Corporation (May 2002), http://www.ibm.com/developerworks/ibm/library/i-cluster/?dwzone=ibm.
23. J. M. Nick, B. B. Moore, J.-Y. Chung, and N. S. Bowen, "S/390 Cluster Technology: Parallel Sysplex," *IBM Systems Journal* **36,** No. 2, 172–201 (1997), http://www.research.ibm.com/journal/sj/362/nick.html.
24. N. S. Bowen, J. Antognini, R. D. Regan, and N. C. Matsakis, "Availability in Parallel Systems: Automatic Process Restart," *IBM Systems Journal* **36,** No. 2, 284–300 (1997), http://www.research.ibm.com/journal/sj/362/antognini.html.
25. *High Availability Cluster Multi-Processing for AIX, Concepts and Facilities Guide*, SC23-4864, IBM Corporation (2003), http://publibfp.boulder.ibm.com/epubs/pdf/c2348640.pdf.

Accepted for publication December 1, 2003.

Matthias Kloppmann *IBM Deutschland Entwicklung GmbH, Schönaicher Strasse 220, 71032 Böblingen, Germany (Matthias-Kloppmann@de.ibm.com).* Mr. Kloppmann is a Senior Technical Staff Member with IBM Software Group's laboratory in Boeblingen. He is the lead architect for the Process Choreographer component in WebSphere. Since he joined IBM in 1986, he has worked on a variety of projects, focusing on the architecture of workflow systems and Web services. Mr. Kloppmann holds an M.Sc. degree in computer science and electrical engineering from the University of Stuttgart.

Dieter König *IBM Deutschland Entwicklung GmbH, Schönaicher Strasse 220, 71032 Böblingen (dieterkoenig@de.ibm.com).* Mr. König is a software architect for workflow systems at the IBM Germany Development Laboratory. He joined the laboratory in 1988 and has worked on the Resource Measurement Facility for z/OS®, MQSeries® Workflow, and WebSphere Process Choreographer. Mr. König holds a master's degree (Dipl. inform.) in computer science from the University of Bonn, Germany.

Frank Leymann *IBM Deutschland Entwicklung GmbH, Schönaicher Strasse 220, 71032 Böblingen, Germany (ley1@de.ibm.com).* Dr. Leymann is an IBM Distinguished Engineer and a member of the IBM Academy of Technology. He is the chief architect and strategist of IBM's workflow technology. As a member of the Software Group Architecture Board, he contributes to setting the direction of IBM's middleware. In addition, he is the co-leader of the team that is in charge of IBM's Web Services Architecture and is the technical lead of the Software Group Architecture Board for the architectural middleware aspects of on demand computing.

Gerhard Pfau *IBM Software Group, Schönaicher Strasse 220, 71032 Böblingen, Germany (gpfau@de.ibm.com).* Mr. Pfau is the chairman of the Technical Expert Council EMEA CR and a member of the WebSphere Foundation Architecture Board. He has worked on the architecture of WebSphere Application Server Process Choreographer since its inception and contributes to business process standardization efforts in the Java community. Mr. Pfau has published many articles and has given talks at various conferences about workflow, Java server technology, enterprise application integration, and other topics. Before working on business process architecture, he was the lead architect for IBM's SAP J2EE Connector.

Dieter Roller *IBM Deutschland Entwicklung GmbH, Schönaicher Strasse 220, 71032 Böblingen, Germany (rol@de.ibm.com).* Mr. Roller is an IBM Senior Technical Staff Member and a member of the IBM Academy of Technology. He joined IBM in 1974 as a junior programmer and has held several technical and management positions in his IBM career. He is a member of the workflow architecture team. His current focus is on BPEL4WS as a co-author of the specifications as well as a member of the appropriate OASIS Technical Committee. Mr. Roller is the co-author of a textbook about workflow systems. He is a part-time lecturer at the University of Cooperative Education, Stuttgart.