Chapter 1. Introduction

1.1 What are Business Processes?

Traditionally, an application contains code for the implementation of individual business functions (such as “create order entry”), and code for the flow logic of the application according to the business requirements (such as “high-value order messages have to be separately approved”). Whenever the business requirements change, the code must be changed. Usually these changes are to the business flow logic and only rarely do they affect the implementation of the individual business functions.

The use of business processes changes the way that applications are built. The process engine allows an application architecture that separates the description of the business logic (the flow logic) from the implementation of the business functions. The resulting application structure is known as a business process-based application. The control flow is managed by the business process management system, which is responsible for the invocation of the individual business functions according to the business logic.

1.2 What is WAS Enterprise Process Choreographer?

Process Choreographer is a component of WebSphere Application Server Enterprise Version 5. It supports the execution of short and long-running business processes involving J2EE constructs, Web services (both synchronously and asynchronously), people (via Web browser-based GUIs), and compensation.

1.3 Where to find more information

This paper describes the programming model for business processes in WAS Enterprise Process Choreographer. A companion whitepaper [1] provides additional background, including concepts of business processes, the architecture of the business process engine, scenarios and more. In this paper, it is assumed that you are already familiar with those concepts.
Chapter 2. Business Process Constructs

This chapter looks at the meta-model constructs available for the construction of business processes, and their properties. In contrast to other programming model descriptions, there is no textual notation available for these constructs, so the properties are described in prose.

2.1 The Business Process Itself

The business process itself serves as the “container” for all other constructs. It has a number of properties that describe its behavior and its interface.

2.1.1 Microflows (“non-interruptible processes”)

Microflows are short running business processes. The mental model for a microflow is that it implements an in-out operation. A microflow is invoked with some input parameters, and the caller waits while the microflow is executed synchronously by the WebSphere process choreographer engine. After a very short time, the result is returned to the caller.

This execution model has the following implications:

1. The steps performed on behalf of a microflow must also be synchronous steps. A microflow can contain no service activities with asynchronous bindings (see section 1.1.1), no event activities (see section 2.3.8), and no staff activities (see section 2.3.9).

2. The execution of a microflow is embedded in a single transaction. A microflow has no persistent state associated with it.

A business process is treated as a microflow if all the activities in it satisfy the criteria in 1, and if its flag “always run process as interruptible” is not set.

2.1.2 Long-running processes (“interruptible processes”)

Long-running processes are executed as a sequence of transactions that are chained together via JMS messages, with the current state of the business process being kept by the process choreographer engine as part of each transaction.

The mental model is that of a long-running entity that is triggered by an external event, such as an API call, a JMS message, or a SOAP service request, and then executes for a while. Long-running business processes can literally execute for days or weeks, and can even span the downtimes of the hosting environment. More information on the transaction and execution model of long-running processes can be found in [1].

A long-running process can involve all kinds of activities, including asynchronous service invocations, event activities and staff activities.

A business process is treated as a long-running process if it contains an activity of one of the types not allowed in microflows, or if its “always run process as interruptible” flag is set.
2.1.3 Process Interface

![Diagram of process interface](image)

Figure 1: Input, output, faults and events of a business process

Input, output and faults

A process interface is similar to that of a WSDL in-out operation: the process consumes an input message, and produces either an output message, or one of several possible fault messages.

There are two possible ways to specify the signature of that interface, depending on the usage pattern of the process:

1. If the process will be invoked synchronously (it is a microflow) or pseudo-synchronously (it is a long-running process invoked as a WSDL in-out operation via an asynchronous WSDL binding such as the JMS binding), then its signature can be defined by specifying a WSDL in-out operation with the appropriate input, output and faults. This matches the mental model of a microflow implementing a WSDL in-out operation.

2. If the process will be invoked asynchronously (it is triggered by a call to one operation, and is supposed to deliver its output, or faults, by calling another operation), then its signature must be specified by one in-operation for the process input (which later is used to initiate the process), and one in-operation for the process output and each of its faults (which will be called back to deliver the result or fault).

For the specification of the signature of the process, both cases are equivalent, as only the type information is relevant. However, for case 2 above, the session EJB generated for the process (see 5.2.1) will contain code to actually perform those callbacks. Still, when the generic interface is used, appropriate callbacks can be specified dynamically, and the actual operations specified for the process interface are ignored (see 0).

Events

The event activities belonging to a long-running process (see 2.3.8) require associated in-operations at the process interface, which are used at runtime to deliver the event, via a call to the process’ façade EJB, to its façade MDB, or to the general API (see 5.1). They are also typed by a WSDL message, which defines the type of the event data that is delivered to the event activity.

Correlation IDs (user-defined process instance IDs)

For input nodes and event activities, a piece of Java code can be specified that derives a correlation token from the message sent to the process instance. At runtime, the correlation token derived by the Java code associated with the input node is used as an alternative (user-defined) ID that is assigned to the process instance, while the correlation token derived by the Java code
associated with the event activity is used to retrieve the process instance with the matching ID that the event is to be sent to.

An example of Java code extracting the `customerID` and `orderedID` properties of the input or event message and using their concatenation as the correlation token to identify the actual process instance would be coded as in the following example. Note that the actual message is passed as a parameter called `message`, and the result is returned as a parameter called `correlationID`:

```java
correlationID = message.getCustomerID() + ':'
   + message.getOrderID();
```

### 2.1.4 People assigned to the business process

There are three types of users that can be assigned to business processes:

- **Process administrators** have specific responsibilities for running processes, for example, they can terminate a running process, and they are notified about errors.
- The **process starter** is allowed to start a new instance of the business process.
- **Process readers** are allowed to see running process instances and their properties.

For human interactions within a business process, you can assign people to work on an item.

- The **potential owner** gets the work item on his or her To-Do list, can work on it and complete it.
- The **editor** is allowed to work on the item, but cannot complete it.
- The **reader** can view details of the item, but is not allowed to change anything.

For more information about people assignments, see the description of staff activities in 2.3.9.

### 2.1.5 User-interface Properties

A business process can either be started programmatically and its results be processed programmatically, or it can be started manually using a graphical user interface (GUI), such as the Web client of Process Choreographer, where the results will also be displayed. If the Web client is being used, customized JSPs can be associated with the process, which render the user interface presented to the user. JSPs can be assigned for the following GUI actions:

- **ProcessInputMessageDisplay.** This action is used to request the user to enter values for the input message of a new process instance. Two JSPs can be assigned, one to render the form where the user enters the input values (Input Message JSP), and one to map the user entered form data to an appropriate input message (Message Mapping JSP).
- **ProcessOutputMessageDisplay.** For microflows only, this action is used to display the result. One JSP can be assigned to render the output message (Output Message JSP).
- **ProcessInformationDisplay.** For long-running processes only, this action is used to display information about the active process instance, including its result, once it becomes available. Two JSPs can be assigned, one to render the input message (Input Message JSP), and one to render the output message (Output Message JSP).

More information on the programming of those JSPs can be found in 5.1.5 and in [9].
2.1.6 Other Process Properties

Audit trail

This Boolean setting determines whether audit records are written during the execution of instances of this process.

Deletion upon completion

This Boolean setting determines whether instances of this process will be deleted automatically after they have run to completion, or whether they must be explicitly deleted.

Valid-from

The valid-from timestamp (date and time) allows versions of processes to be deployed that will only become active at a given time in the future. When a new process instance, of a given name, is created, the active version is used, that is the latest version with a valid-from time in the past.

2.2 Variables of the Business Process

Similar to programs developed using programming languages, business processes use variables to store any intermediate results, such as the process input, input to activities, or output from activities.

Variables are typed by WSDL messages [2].

2.2.1 Declarative access to variables

In a process, variables are normally accessed declaratively, with the business process engine performing the read or write operations implicitly. “Declarative” in this context means that the process modeler does not have to code read or write operations for variables; rather he or she simply assigns variables to input, output, and fault nodes.

At the process level, the variable associated with the process input is written when a new process instance is created; it receives the data that was passed as the input message of the process. Similarly, a variable is associated with the process output; it is read upon normal completion of the process instance and its content becomes the process result. The same is true for the variables associated with any faults; if the process instance terminates with a fault, the contents of the associated variable are returned as the fault message.

At the activity level, the variable associated with the activity’s input is read when the activity is activated and its content becomes the input for the invocation of the implementation associated with the activity, such as a Web service, an EJB method, or an interaction with a human. After the implementation of the activity completes, its result is written to the variable associated with the output terminal, which is either a regular output or a fault.

2.2.2 Programmatic access to variables

A special type of activity has no inputs, outputs and faults: Java snippets (see 2.3.4). For Java snippets, access to variables is via appropriate Java accessor methods, which are described in 5.1.4. Java snippets can read and write as many variables as needed.

Also, non-trivial transition conditions (see 2.4.3) are coded in Java and access variables using the same Java accessor methods. Transition conditions only read variables.
Figure 2: Implicit data flow resulting from usage of variable

2.2.3 Data flow

There is no explicit data flow representation. Rather, the data flow is determined implicitly by storing a result from one step in a variable and retrieving it from the variable in a later step. The left part of the figure shows an example for this, where the input data for the process is written to a variable, which is then used by the two subsequent activities.

If the input of an activity cannot directly use the output of a previous step, type mapping must be realized by a Java snippet. In the example in , the Java snippet combines the results of activities one and two to provide the input for activity three.

2.3 Activities

An activity is a step in the business process. Several different possible implementations for activities are described in the following subsections. All activities share some common properties, detailed below.

2.3.1 Common properties

Activities are labeled with a name. They can also have a description, to provide more information about the purpose of the activity. Both the names and the descriptions are limited in length, as they are displayed in work lists and are available for queries at runtime, and must adhere to the specifications for searchable columns in the underlying DBMS. This restriction does not apply to the documentation of an activity, which can hold an unlimited quantity of descriptive text, but is not searchable at runtime, and not displayed in work lists.

Activities have input, output, and fault terminals, which are associated with variables, and are used by the activity’s implementation as described in 2.2.1.

In addition, each activity has a flag to control whether audit trail information will be created for that activity at runtime or not.

2.3.2 Empty activity

An empty activity is a step in the process that does not have an associated implementation. Empty activities are mainly used during top-down modeling of processes, to sketch out the structure of the process without having to add concrete implementation details. When an empty activity is executed, it is effectively a no-op. It does not produce a fault, and does not update any variables.
2.3.3 Service activity

A service activity is a step in the process which is implemented by a service operation, that is, an operation from a WSDL port type [2]. It always has at least an input and an output terminal, and has as many fault terminals as the associated service operation declares faults.

The notion of a service is generic; it covers anything for which a WSDL binding definition [2] and an associated WSIF provider [3] are available, such as invocations of J2C adapters, EJB methods, Java methods, SOAP services and many others. The binding is specified implicitly by specifying the WSDL service that is to be used for the activity. The WSAD-IE tool [4] provides shortcuts to simplify the creation of activities that use some of the more common binding types, without having to explicitly create the associated WSDL definition.

Service activities can also be implemented by asynchronous services, such as a service made available via JMS messaging, using appropriate JMS WSDL bindings. Because of the nature of asynchronous operations, this kind of service activity can only be used in long-running processes (see 2.1.2).

Service activities can have a compensation operation specified (see 2.5.1), except for those implemented by an asynchronous service.

If more than one service activity must interact with the same instance of the service, the same partner name is associated with those activities that should share the instance of the service. During the execution of the process, the process engine reuses the WSIF port used to invoke the operation for all those invocations. This is needed for microflows to allow interaction with a JCA connector where multiple activities must share a session.

2.3.4 Block

A block is an activity that is itself implemented by a graph of activities. Blocks provide a way to structure complex processes into manageable units. They are like sub-processes (see 2.3.6), except that they are contained in the enclosing process and do not constitute stand-alone processes of their own.

There is a performance consideration associated with blocks: The entire sub-graph contained in a block is instantiated if and only if the block is actually reached by the flow. It can be more efficient to put parts of the process logic that are rarely executed (such as exception handling) into blocks.

2.3.5 Loop

A loop activity can be used to construct well-structured loops over sub-graphs. It is like a block, except that the loop has a condition associated with it that controls the repetitive execution of the sub-graph. The loop condition is specified in Java. The loop behaves as a while loop, that is, the contained sub-graph is executed as long as the condition returns true.

The following example shows a condition for a loop that is to be performed while a counter n stored in variable myVar is larger than 0. The example also shows the convention that the condition must access all variables via the appropriate accessor functions, while it returns its result as a parameter with the name result.

\[
\text{result} = \text{getMyVar().getN() > 0} ;
\]
2.3.6 Process activity – sub-processes

A process activity is a step in the process which is implemented by another process. It allows processes to be composed recursively from other processes that have been defined, deployed and versioned independently. In particular, a process can be used as a sub-process in many other processes. This distinguishes it from a block (see 2.3.4), which is inlined into, and is therefore only visible to, its enclosing process.

The only specific property of a process activity is the name of the referenced sub-process. At runtime, that version of the sub-process that is valid when the process activity is reached during execution will be used.

2.3.7 Java snippet

Java snippets provide a way to incorporate Java code inline into the process, to be executed inside the business process engine as part of the execution of the process. This code can access process variables and perform Java computations of any sort that are needed to write other process variables. The actual Java interfaces available are generally described in Chapter 5, those of specific interest to Java snippets in 5.1.4.

The following example Java snippet accesses the variable orderData to calculate the field amount of variable payment.

```java
float amount = getOrderData().getPrice() * getOrderData().getItemCount();
PaymentMessage newPayment = new PaymentMessage();
newPayment.setAmount(amount);
setPayment(newPayment);
```

2.3.8 Event activity

An event activity is a step in the process that is used to synchronize the execution of the process with an “external stimulus”, such as the arrival of a message or an API call. An event activity can wait for one or more incoming events. As soon as one of the awaited events is received by the process instance, the event activity completes, and navigation of the process continues from the output terminal of the event activity associated with the received event.

Events can only be used in long-running processes. See section 2.1.3 for more information on how events appear in the process interface, and how correlation IDs can be derived from the content of the message that is sent as part of the event.

In addition, an expression for the maximum wait time until the event activity expires at runtime can be specified. This specification uses the WAS Enterprise scheduler [5], and the calendar support provided by it. When the maximum wait time is exceeded at runtime, the activity expires and all its output terminals are activated with an empty message. See section 2.4.3 for an example that shows how the process logic can react when an event activity expires.

An example for a duration specified using the default calendar of the WAS scheduler would be “3 weekdays”.

2.3.9 Staff activity

A staff activity is a step in the process that is not executed automatically by a piece of code, but rather, manually performed through interaction with a person via a
client GUI program, such as a Web client running in a browser. Staff activities always have exactly one input and one output. The input delivers the values that are presented to the user as part of the information about the job she has to perform, while the output represents the values that the user returns upon completion of the job, such as approval information.

People are assigned to staff activities by means of *staff queries*. They are resolved against an organizational directory and return a list of people. The most important staff query is the one specifying potential owners of the activity; one of the potential owners at runtime will claim the activity to work with it, thus becoming its owner, and will ultimately complete it. There are additional, optional staff queries that allow for the specification of *editors* (people who are allowed to see and change the data associated with the activity, but who may not claim or complete it), and *readers* (people who are allowed to see the activity and its associated data, but cannot make any changes). Staff queries are specified by means of *staff verbs*. The concept of staff verbs and their translation into “real” queries against physical organizational directories is described in [6].

In its simplest form, the client program that people use to interact with Process Choreographer displays a generic user interface for each activity. To replace the default user interface, it is possible to associate custom JSPs with the activity, which will be used by the client program when data of this activity is displayed. Staff activities support the single GUI action `ActivityDisplay`, to which three JSPs can be assigned. The `Input Message` JSP is used to render the activity’s input message, the `Output Message` JSP is used to enter the activity’s output message, and the `Message Mapping` JSP is used to create the output message of the activity from the entered form data.

More information on the programming of those JSPs can be found in 5.1.5 and in [9].

Finally, staff activities can expire, similar to event activities, so it is possible to specify an expression for a maximum duration and an optional calendar interpreting it.

### 2.4 Control Links

While activities describe the individual steps of a business process, control links connect those steps and describe the overall control flow of the process.

#### 2.4.1 Execution order constraints

A control link between an activity $A_1$ and an activity $A_2$ specifies that $A_1$ must be finished before $A_2$ can be started. A control link between the input node and an activity specifies that the activity is started right after the process has started. Likewise, a control link between an activity and the output node specifies that the process ends after execution of that activity.

The activities belonging to a process must form a connected, cycle-free graph. That is to say, for each activity, all its terminals must be connected by at least one control connector, and that the graph of control links must not contain cycles. If loops are required as part of the process logic, a loop construct (see 2.3.5) must be used.

#### 2.4.2 Parallel paths, forks and joins

More than one control link can originate from a single output (or fault) terminal of an activity. This is a *fork* in the control flow of the process, which can create multiple parallel paths of execution for the process. Activities on parallel paths may be executed in parallel or in any order chosen by the process engine.
The opposite of a fork is a control flow join. A join synchronizes two parallel paths of execution and combines them back into a single path. The target activity of a join is only activated when all control links leading into it have been navigated, that is, all its predecessor activities have reached an end state.

2.4.3 Transition conditions

Control links can be conditionally navigated by associating a transition condition with them. The transition condition is evaluated at runtime and its result determines the truth value of the control link. If that value is true, the target activity is activated; otherwise, the target activity is skipped and dead-path elimination originates from it (see [1]).

Transition conditions are specified as a fragment of Java code that typically tests the values of variables. For example, testing whether the value of field in the variable myVar is larger than 42:

\[
\text{result} = \text{getMyVar().getField() > 42} ;
\]

Business Rules Beans [7] can be used in transition conditions to separate the specification of business policies from the actual code using it, and thus allow the business policy to be changed without having to change the business process definition.

Through the API (see 5.1), transition conditions can also access context of the business process they are running in. This allows a transition condition to test whether the preceding activity finished normally or expired:

\[
\text{result} = \text{activityInstance().getExecutionState() == ActivityInstanceData.STATE_EXPIRED} ;
\]

2.4.4 Join conditions

The target activity of a join is activated if and only if at least one of the incoming control links evaluates to true, so the implicit join condition of an activity is always “Any”.

If more complex join conditions are required, they can be coded as a transition condition from an empty activity, as shown in the figure to the right. In this example, A3 must only be activated after A1 and A2 have been completed, and only if both transition conditions p and q evaluate to true, which requires a join condition of “All” (as shown in grey). To achieve the required effect, an empty activity is introduced, and the necessary condition is specified in a transition condition on the control connector originating from that empty activity to A3 (as shown in red).

2.5 Compensation

The effects of long-running business processes cannot be undone by simply rolling back the current transaction, because intermediate results may already have been committed and made visible to the outside world. Compensation provides the means to return to a consistent state that was active before a business process was started, by performing the necessary actions needed to compensate those steps in the process that were already executed.

For compensation to work, undo operations must be specified for each activity, and the process itself must be enabled for compensation.
2.5.1 Specifying undo actions

An activity can optionally be associated with an “undo” operation. This operation must reverse the side effects of the operation that implements the activity (the “forward operation”). If, for example, the forward operation increments a field in a database, the associated undo operation must decrement the field by the same amount. If the forward operation sends out an e-mail containing an offer, the associated undo operation must send another e-mail containing the withdrawal of the offer.

Currently, undo operations can only be specified for service activities, and they must be service operations themselves. Note: It is still possible to compensate EJB or Java invocations, by first wrapping them into a service, and then invoking them from a service activity.

The forward operation is invoked with “presumed abort” semantics, which means that if the forward operation fails, it is assumed to have caused no side effects, and its associated undo operation is not called when the process is compensated. This means that for any sequence of activities A₁ ... Aₙ, the failure of forward operation Aₙ results in the triggering of undo activities Aₙ₋₁ ... A₁.

Part of the specification of the undo action must indicate whether the forward operation is transactional or not. If it is transactional, its side effects are only committed as part of the current transaction; if the current transaction fails, the forward action was not executed, so no undo operation has to be performed. On the other hand, if the forward action is not transactional, its side effects happen right away, and its associated undo operation must be called even when the current transaction fails.

2.5.2 Specifying a compensation sphere

The “run with compensation sphere” flag at the process level tells the process engine whether the process should be enabled for compensation at runtime. Only when this flag is set, will the engine write additional log information for those activities that have associated compensation operations, thus allowing it to compensate the process when necessary.

If this flag is not set (the default), compensation is disabled, even if some activities have associated undo operations.
Chapter 3. Enterprise Applications Containing Business Processes

This chapter describes the structure of an enterprise application that contains business processes.

3.1 Process template

A process template represents the definition of one particular business process that has been deployed and installed on a WebSphere node or cluster.

Process templates can be started and stopped individually. A process template can also be used to query information about the currently living process instances of that template.

3.2 Process module

A process module represents one or more business processes as a deployable and installable unit that contains one or more process definitions. In addition, a process module may contain deployed artifacts belonging to constructs of the business processes, such as the compiled code for transition conditions, loop conditions and Java snippets.

Process modules are packaged as FAR files. The FAR file format is used as the interchange format between the business process tool and the business process engine. FAR files are typically contained in an enterprise application EAR file.

3.3 EAR file structure for process-based applications

At deployment time, the artifacts necessary for execution are created, and are packaged together into the process module (FAR file) which is then put into the associated enterprise application archive (EAR) file.

The EAR file contains
- FAR files containing the process definitions: Each FAR file can contain one or more business processes.
- JAR files containing the Java classes used for Java-based activities as well as the WSDL files used for service activities and message definitions.
- WAR files are optional, they contain the JSPs used by the business process, for example, for starting the process or for human interaction.

3.4 Installing and uninstalling EAR files that contain business processes

For the administrator, installing the EAR file on the server is the same whether it contains process modules or not. The deployment process will process any FAR files it finds in the EAR file, checking their contents and writing them into the Process Choreographer database.

Before the EAR file can be uninstalled, all process templates that are part of the EAR must be stopped (so that no new instances of the templates can be created) and all running instance must have finished and have been deleted.
Chapter 4. Tooling and Debugging of Business Processes

WebSphere Studio Application Developer Integration Edition (WSAD-IE) is the graphical front end for creating business processes to run in Process Choreographer. The Integration Edition is based on WSAD, which itself is based on the WebSphere Studio Workbench, powered by Eclipse technology.

Among other editors and tools, WSAD-IE extends the WebSphere Studio Workbench with a process editor and process debugger.

4.1 WSAD-IE process editor

The process editor is a visual tool that lets you create a business process using existing Java beans, EJBs, and services (such as SOAP-based Web services, or JMS-based services).

Before you create your business process in WSAD-IE, you need a place to store it. WebSphere Studio Workbench stores all the objects in projects. Each project includes associated “rules”. For processes we need a project that takes care of creating all artifacts that are related to the business process. This is defined in a Service Project, which is a component available as part of the “Business Integration” perspective.

You create the process by adding nodes which are sequenced by control link connections. You can add nodes to handle external events and human interaction. In addition, you define the data for each of the used services as well as the mapping of data.

Both a bottom-up and a top-down approach is possible: You can either start by creating “placeholders” as nodes in the process first and refine them later by creating the appropriate implementations like EJBs or Web services, or you can start by creating the implementations like Java classes first and then use these implementations in the business process.

The process is stored within a Java package which in turn resides within a Service Project.

When the process is created, you can already define its interface, which describes the data provided at the start of the business process and the data that is the result. Data is always defined in the form of WSDL messages. The interface of the process is defined either as a request-response operation in WSDL (for synchronous calls) or two one-way operations (for asynchronous calls).

With each process, a Java class (called the “process backing class”) is generated that contains

- All transition conditions (see 2.4.3).
- All loop conditions (see 2.3.5).
- All Java Snippets (see 2.3.7).
- The method for calculating the correlation ID of the process instance (see 2.1.3).

Within this code, the user has access to the Process Choreographer API by using the methods processInstance(), blockInstance(), and activityInstance(), provided by the base class of the process backing class (see 5.1.4).
4.2 WSAD-IE process debugger

WSAD-IE provides a graphical process debugger which you can use to test and debug your business processes. It allows you to debug the control flow within the business process, view and manipulate data, and step into the Java code behind control links, loops and Java-based activities.

For debugging business processes, there is a special perspective provided called the “Process debug perspective”.

First you must attach the debugger to a process engine. This can either be in a remote WebSphere Application Server, or the application server integrated within WSAD-IE. The Unit Test Environment (UTE) allows you to develop, test and debug your business process including your Java code and JSPs within one environment.

Like in a Java debugger, the process debugger allows you to set breakpoints and to view and change data. Breakpoints are always set on control links. The data are the WSDL-based message types that you have defined for your business process. The actual values can be changed “on the fly” while you are debugging a process.

You can step over an activity, step into a Java-based activity (which will open the Java debugger for you) as well as running to the next breakpoint or to the end of the process.

Figure 3: The WSAD-IE Process Debug Perspective
Chapter 5. Interfaces to the Business Process Container

The business process container can be accessed using the Business Process Session EJB (“API”) or by using business process façades generated by WSAD-IE.

If you want to code against a generic interface, you must use the Business Process Session EJB (“API”). The other available interfaces, the generated façades, are process specific, so you can use them if you create code for a specific business process.

5.1 The Business Process Session EJB (“API”)

The Business Process Session EJB provides a generic, synchronous API for the business process container. This API covers the complete functionality of the business process container. Using this API, a client can invoke process-related functions, work with activities, or perform queries. A detailed description of the API can be found in [8].

To access the API remotely, you must create an instance of the session EJB class com.ibm.bpe.api.BusinessProcess using its home interface. An instance of the home interface is registered in the namespace of the server running the business process container. The JNDI name of the home interface is “com/ibm/bpe/api/BusinessProcessHome”. For details about how to access the remote API, refer to the Javadoc Package description in the WebSphere Application Server Enterprise Information Center [8].

If your client application is running on the same application server as the business process container, it is recommended to use the local EJB interface. The class name of the local interface is similar to the remote interface name, but with the prefix 'Local' added, for example, LocalBusinessProcess and LocalBusinessProcessHome. When performing a local lookup, it is not necessary to perform a narrow. The calling actions are the same regardless of whether you are using the local or remote API. The code for obtaining a local home instance looks like this:

```java
InitialContext context= new InitialContext();
LocalBusinessProcess process= home.create();
...
process.initiate(...);
```

5.1.1 Process-related functions

Process-related functions are the APIs to create and start business processes, to interact with running business processes, and to perform administrative actions on business processes.

Creating and executing business processes

There are three ways to create and execute a business process. These three ways are exposed by the API methods call, callWithReplyContext, and initiate. The methods call and callWithReplyContext are used to start processes that return output to the caller. Initialize
WebSphere Application Server Enterprise Process Choreographer Programming Model

is used to start a process that either produces no output or where the caller is not interested in the output produced.

Starting a business process using call means that the process is created and started synchronously, that is, waiting until it has finished, and its results are received. This way of starting a process is only possible for microflows (see 2.1.1).

The method callWithReplyContext is very similar to call, it allows invoking business processes and receiving a result. The only difference is that the result is passed back to the caller at a later point in time. To allow for that, the caller provides a callback that implements the com.ibm.bpe.api.ReplyContext interface. The callback is stored by the business process container as part of the process instance and used later to send the output of a process back to the caller. This way of starting a process is only possible for long-running, interruptible processes (see 2.1.2).

Starting a business process using initiate is different from the previous methods because it does not return the output of the process to the caller. Instead, the caller obtains an identifier, the process instance ID (PIID). This identifier can subsequently be used to interact with the running business process. Starting processes with initiate only makes sense for long-running, interruptible processes (again, see 2.1.2).

**Interacting with running business processes**

Interacting with running business processes is only possible for interruptible business processes. One possible interaction is to send an event to a business process waiting on an event activity (see 2.3.8). To accomplish that, the sendEvent methods accept either a PIID or the process instance name together with the name of the event and the event data to send. The caller must be a potential owner of the event or an administrator of the associated process instance.

Besides that, there are methods for accessing the instance state of a running business process. The list below shows the methods offered together with a short explanation.

- `getProcessInstance` returns an object that provides read access to the state of a running business process.
- `getActivityInstance` provides read access to a certain activity in a business process, identified by its activity name.
- `getVariable` allows you to read the value of a variable belonging to the business process, identified by its variable name.
- `getInputMessage`, `getOutputMessage`, and `getFaultMessage` provide read access to the input, output, or fault data of the process.

**Business process administration**

There are API methods to administer long running business processes. The forceTerminate method allows an administrator to terminate a running process. Terminating a business process also terminates all its child processes, its running or claimed activities and its ability to receive events. If the process has the “delete upon completion” flag set (see 2.1.6), it will also be deleted.

Processes that are in an end state (finished, terminated, compensated, or failed) can be deleted explicitly by an administrator using the delete method.
Accessing meta-information

The methods `getEventNames` and `getFaultTerminalNames` return information about the meta-model of the business process. The `getUISettings` method provides information to clients interacting with the business process.

5.1.2 Activity-related functions

Activity-related functions are the APIs that provide support for person activities, access to activity data, and administrative actions. All these functions require an activity instance ID.

Person activity functions

To work with person activities, the following methods are provided:

- `claim` reserves a person activity for exclusive access by a potential owner of the activity or an administrator.
- `cancelClaim` returns an activity that has been previously claimed without applying changes.
- `setOutputMessage` can be used to save an intermediate result.
- `complete` finishes the work on an activity and makes corresponding changes persistent.

Access to activity data

The following functions provide access to activity instance data.

- `getActivityInstance` provides read access to an activity.
- `getInputMessage`, `getOutputMessage`, and `getFaultMessage` provide read access to the activity’s input, output, or fault data.

Administration of stopped activities

Runtime errors can cause an activity to stop. The following methods allow an administrator to recover from this state.

- `forceRetry` forces a stopped activity to be repeated. You can retry the activity with the same input data or with new (potentially corrected) input data.
- `forceComplete` forces the completion of a stopped activity. You can provide output data for the activity. Forcing the completion of an activity instance allows the business process engine to continue navigating the process instance.

Accessing meta-information

The methods `getOutputTerminalNames` and `getFaultTerminalNames` return information about the meta-model of the activity. The `getUISettings` method provides information to clients that are interacting with the activity, such as the available JSPs.

5.1.3 Queries

Queries are used to retrieve a certain set of data from the Process Choreographer database. There are predefined methods for the most commonly used queries. Additionally, the query methods allow arbitrary user defined queries.
Predefined queries

The following functions provide access to predefined queries:

- `queryProcessTemplates` retrieves business process templates that are stored in the Process Choreographer database.
- `getWorkItems` returns the work item assignments for the logged-on user and the specified process instance or activity instance.

Ad-hoc queries and worklists

The following functions support ad-hoc queries:

- `query` retrieves selected object properties that are stored in the Process Choreographer database. The query is defined by specifying `select`, `where`, and `order-by` clauses. A filter or a threshold can be specified to restrict the number of tuples that will be returned. The tuples are sorted on the server according to the specified sort criteria.
- `newWorklist` allows an administrator to store a query with a particular name.
- `executeWorkList` executes a query defined by a worklist identified by the worklist’s name and returns the qualifying object properties as a result set.
- `deleteWorklist` allows an administrator to delete a worklist.

5.1.4 Process backing class and helper methods

WSAD-IE generates a process backing class for every business process. The process backing class has the following functions:

- Hold the generated methods providing type-safe access to the variables in a business process.
- Carry all methods containing the various user-defined Java code pieces (such as transition conditions) for a business process.
- Contain methods for all Java code snippets containing user-defined mapping code that are used by a business process.

The process backing class extends `com.ibm.bpe.data.ProcessBackingBase`. This base class provides helper methods required to implement Java code in the methods of the process backing class:

- `processInstance` returns the enclosing process instance.
- `blockInstance` returns the current block instance.
- `activityInstance` returns the current activity instance. If invoked from a transition condition, this is the activity from which the control link carrying the transition condition originated.

To access the data belonging to a business process from the Java code for transition conditions, loop conditions, or Java snippets, the methods to access the variables themselves are generated, as are classes containing “get” and “set” methods for each part of a variable. These are wrappers for the WSDL messages specified as input and output data of the business process and each of its activities.
5.1.5 Usage of the API in JSPs

When a process is modeled with WSAD-IE, you can specify business process specific JSPs ("custom JSPs") for the process (see 2.1.5) as well as for person activities (see 2.3.9). These JSPs allow you to extend the Web Client and should be taken into consideration if a message has non-primitive parts or if the message fields need additional information.

Custom JSPs can be specified for the input message (Input Message JSP) and the output message (Output Message JSP) for both processes and staff activities.

Custom JSPs are integrated into the existing Web Client JSP using JSP:include functionality (see Figure 4 for an example of a custom JSP for the process input message).

For each custom JSP that contains a form that receives user input data, you need a Message Mapping JSP. This JSP receives the input data, wraps it into a ClientObjectWrapper message object and forwards this message to the Web Client.

The MessageUtilities package offers the following convenience methods for the user:

- `getProcess(request)` for getting the Business Process bean’s local interface and hence access to the Process Choreographer’s API.
- `forwardMessageToController(…)` for setting up the environment and forwarding the request to the Process Choreographer engine.

Additional information like the ID of the activity or process instances is passed in via request parameters (WF_AIID, WF_PIID). Using the above information, you can use the Process Choreographer API to get the business process input/output message.

The custom JSPs are packaged as part of a Web application, together with all the other parts of the business process, into a single EAR file. This Web application is installed together with the business process.
Figure 4: Web Client with Custom JSP for the Process Input Message

The following code snippet shows the parts of one MessageMapping.jsp where the input message is filled, a new ClientObjectWrapper for this message is created, and the message is passed on to the Web client controller code.
5.2 Generated façades

Using the «Generate deploy code» action in WSAD-IE, three types of façades can be generated for a business process. Although normally only one façade will be used, it is possible to generate and use more than one.

5.2.1 Façade session EJB

A façade EJB is a generated session EJB that provides a specialized version of the generic ProcessTemplateData object that the API provides. There is one façade EJB per business process, which has a strongly typed interface to start it.

In addition to the initiate, call and sendEvent methods that are provided by the façade EJB like they are in the API, strongly typed methods are provided for starting a process and sending events.

The method to start a business process is named after the WSDL operation name used to define the process interface in WSAD-IE. If the WSDL definition is a one-way operation, it just kicks off the process and returns void. For a request-response operation, it returns when the flow is finished and provides the first part of the output data of the business process.

Event methods are generated if the process has event nodes. For each event belonging to the process, an associated method is generated, whose name is of the form <eventOperationName> (taken again from the WSDL definition of the event operation), and which is also strongly typed.

Note that the identifier, which is used to correlate the event to a process instance, must be defined in the Java class of the business process (see 2.1.3). To view the method in the process editor of WSAD-IE, click on the input node of the business process and click on the “Show Java” button in the palette.

5.2.2 Façade MDB

Like a façade EJB, a façade MDB allows you to send JMS messages in a custom format defined in WSAD-IE according to the WSDL interface definition for the business process. Receipt of that message by the MDB causes an initiate or sendEvent call to the associated process. On completion of the flow, the response is sent back to the reply-to queue that was specified in the request.
The wizard that generates the MDB allows the user to set their own JMS properties and define static values for them so that they will be added to the JMS header. In addition to application specific properties, you can add JMS-defined properties, and set values for them. Thus, only messages that are sent to this specific MDB will be received.

There are two message formats that can be selected: TextMessage and ObjectMessage. TextMessage carries the message as a text string (for example XML) that can be read by the client. The ObjectMessage can be either a Java object or an XML document.

5.2.3 Façade SOAP Service

The third option is to create a façade SOAP service by selecting SOAP as the inbound binding type. A WAR Web module containing the SOAP rpcrouter servlet is then included in the EAR file for the process. The client’s SOAP request is sent to the SOAP rpcrouter servlet, which invokes the façade EJB.
Chapter 6. References


