Chapter 4
Workflow Overview

A workflow is a JavaBean component that performs a specific set of tasks connected together in a causal way that enables ordering of the set of tasks based on task completion. The tasks can be work items or other workflows.

This chapter describes the basics of the Workflow application, including how the workflow engine executes a workflow, the workflow language that specifies the rules for connecting components to produce workflows, the classes used to build workflows, the types of work items, and persistent storage and how it affects the execution of workflows.

This chapter contains the following sections:

Workflow Execution **on page 59**

Workflow Language **on page 61**

Workflow Framework Classes and Types of Work Items **on page 62**

LDAP Model for Workflow **on page 63**

Persistent Storage **on page 64**

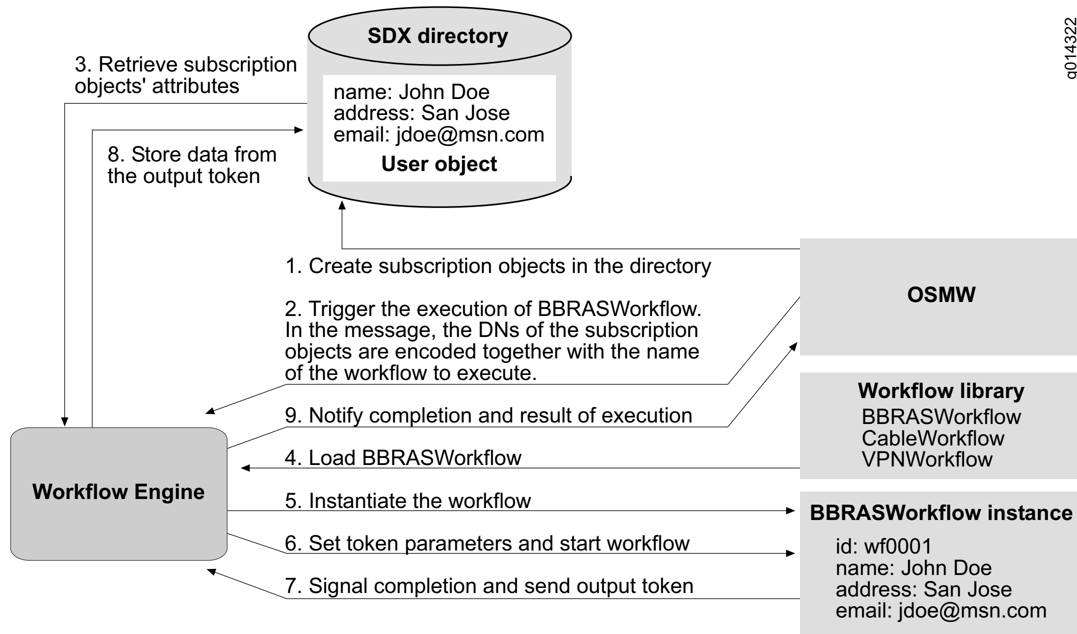
Work Item Life Cycle **on page 64**

Workflow Execution

The execution of a workflow is triggered by a message from an SDX component (see Figure 4). The message includes the distinguished names (DNs) of the objects related to the service being provisioned, together with the name of the workflow to be executed. The objects are stored in the directory.

The DNs are specified by the state machine controlling the life cycle of the target object or by the target object itself, which is always present in the message. The workflow engine uses the DNs to retrieve all the necessary information from those objects and pass them to the workflow. To avoid retrieving unnecessary information, a parameter list is associated with every workflow. This list contains common unique names valid in the realm of the workflow. A translation table maps those common names to LDAP names and is defined in the workflow configuration file.

Figure 4: Sample workflow life cycle: a subscription triggers a workflow named BBRASWorkflow



Receipt of the message from the SDX component instantiates the parameter list associated with the named workflow. All workflow names must end with Workflow. The associated parameter list, if stored in a JAR file, has the same prefix as the workflow but ends with WorkflowParameterList. For further information, see *Chapter 7, Work Item Libr ary*.

Driven by this parameter list, a series of LDAP operations are performed to retrieve the specified data and create a token containing the data. A token is the object that carries the data along with the components and determines their eligibility for execution. Finally, the workflow is instantiated, and this token is passed to it. This action starts the execution of the workflow.

Completion of the workflow is signalled by the return of a token to the workflow engine. The objects in the directory are updated with the data embedded in the token. The parameter list specifies which token parameters must be sent to the directory. The SDX component that initiated the process is notified via a TCP/IP message whether the workflow was successfully completed. Finally, the workflow is removed from the workflow engine.

Figure 4 shows a sample process where creation of a subscription for the user John Doe triggers the BBRASWorkflow.

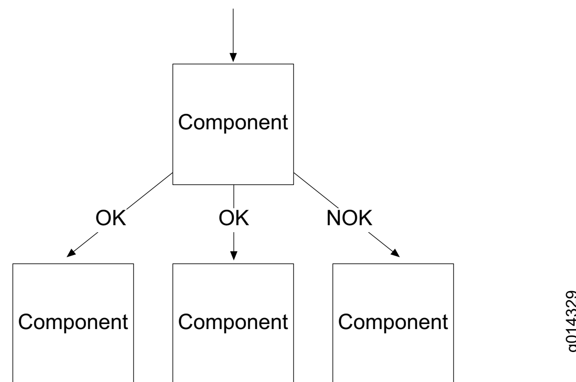
During the entire process, a workflow can be paused if it is not progressing, swapped to disk, and resumed later to try again. This process is done automatically without the knowledge of the workflow. This gives the illusion that all active workflows, the ones that have been started but not yet completed, are running in the main memory. The specifics vary depending on the kind of component, which is explained later.

Workflow Language

The workflow language is a visual language that specifies the rules for connecting components to produce workflows. Each component has an input and two outputs. The input receives tokens from a predecessor component; the outputs send tokens to successor components. Receipt of a token from a predecessor triggers the component to perform its task. When the task is complete, that component generates a token at each output, along with a result from the task. Successor components perform their task depending on the value of the result and the output from which they receive the token. Any number of components can be connected to each output.

The two outputs, named OK and NOK (not OK), send generated tokens with complementary results (Figure 5). For example, if the task was completed successfully, a token with a proceed result is generated at the OK output and another token with a not proceed result is generated at the NOK output. If the task was not completed successfully, the OK output sends a not proceed token and the NOK output sends a proceed token. A component that receives a token with a proceed result performs its task. A component that receives a token with a not proceed result does not perform its task and forwards a token with a not proceed result to both of its outputs, thus preventing all successors from performing their tasks.

Figure 5: Workflow fragment



A special kind of component, the synchronization work item, allows more than one predecessor. This type of component does not have an associated task. It waits until all predecessors are complete, and then performs a logical operation with the results to create the output token, such as the following:

> logical AND – generates a token with a proceed result only if all input tokens have a proceed result.
>
> logical OR – generates a token with a proceed result if at least one of the input tokens has a proceed result.

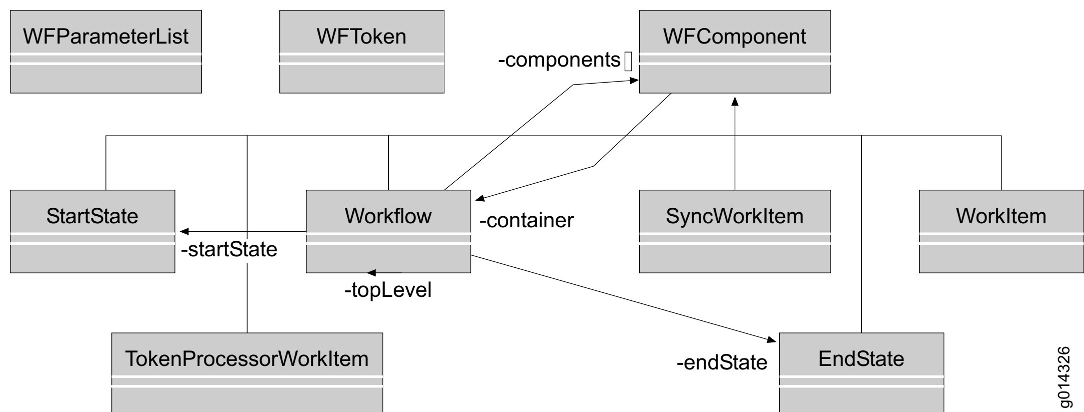
The Start State component is always the first component in a workflow. It has no predecessor component. The End State component is the last component in a workflow. It has no successor components. A workflow has only a single instance of each of these components.

The workflow token carries other data besides the task result, but this data does not influence whether tasks are performed. Thus, there is no distinction between the data path and the control path for a workflow. The data in the token is stored as a list of parameters and values. Parameters can have any name and any value. It is the responsibility of the components that use them to agree in their names and formats. A translation table lists a set of predefined parameter names that correspond to attributes of the objects that triggered the execution of the workflow. The meaning and valid values for each attribute are defined in the directory schema. See *Chapter 8, Workflow Translation Table*.

Workflow Framework Classes and Types of Work Items

All workflows are JavaBean components that extend a common base class, net.juniper.smgt.workflow.Workflow. All the classes mentioned in this section are in the same package as the Workflow class. The Workflow class implements all the basic services needed by a workflow to interface with the workflow engine. The Workflow class also serves as a container for other components. All components are specializations of the WFComponent abstract class, including the Workflow class itself. Thus, a workflow can contain other workflows. The class diagram of the framework is shown in Figure 6.

Figure 6: Workflow framework class hierarchy (UML notation)



The Workflow application employs the following types of work items:

Regular work item type –Most work items belong to the regular work item type. It extends the WorkItem class, which provides the basic infrastructure for executing a task in its own thread. The work item logs the information used by the workflow probe, which collects statistics about the progress of workflows.

Synchronization work item type – Extends the SyncWorkItem class and is the only type that accepts more than one predecessor. The synchronization work item enables the synchronization of concurrent paths and provides a basic interface to the more specialized work items that implement logical operations with the input tokens. These operations are the only factors that determine the contents of the output token. Because the synchronization work item does not have an associated task, it pauses and resumes instantly.

Token processor work item type – Very similar to the synchronization work item type, but does not allow multiple predecessors. It extends the TokenProcessorWorkItem class. The processing operations can alter the token in any way, as long as they do not take a long time to complete. What distinguishes a long-time task from a common task is that the latter does not use external resources that might not be available, and the computation is not complex enough to take more than a few seconds.

Start state work item type – Realized by the StartState class. The Start State type does not have an input, because it is the entry point for a token coming from outside the workflow.

End state work item type – Realized by the EndState class. The End State type does not have an output, because it is the exit point for a token to return to outside the workflow.

See Table 32 in *Chapter 6, Building Workflows* for a list of work items categorized by type.

The WFTokentoken class implements the functionality of a token. It provides methods for accessing the properties and the embedded result. Another support class is the WFParameterList, which implements a parameter list. All parameter lists must extend it and obey the naming convention.

LDAP Model for Workflow

Workflows can be stored in the UMC directory in bytecode format. Workflows are from the type *umcWorkflow* and are stored in the *o=Workflows,o=umc* tree. The OSM takes care of the execution of the workflow requests in a transactional behavior and that they are performed in the proper order according to the object life cycle definition.

The OSM has the knowledge about state machines and their representation in the directory and performs transactions on target objects. The state machine objects are stored as *umcStateMachine* objects in the *o=Workflow,o=umc* tree. Each target, this might be any object from the type *umcuser*, *umcEnterprise*, *umcSite*, *umcRetailer*, *sspServiceProfile*, *umcRadiusPerson*, *umcOutsourceServiceProfile* and *umcAccessServiceProfile*, of a transaction must have the auxiliary class *transactionalObjectAuxClass* attached to it. Those transactional objects can be locked by the OSM, that is indicated by the presence of a *umcTransactionalLock* object in the *o=Lock,o=umc* tree. This object has a copy of the target as direct subordinate within the directory tree. The OSM performs the directory operations for updating the states, creating and deleting the copy of the target.

The creation of state machines, workflows and transaction locks are performed using the SDX Admin that is able to trigger workflows.

For detailed information about the SDX LDAP schema and graphics of the object models, see the documentation on the SDX software CD in the folder /SDK/doc/ldap.

Persistent Storage

All workflow components are persistent objects; that is, they implement the java.io.Serializable interface through the WFComponent class. Persistent storage is responsible for storing the state of workflows, which are indexed by their IDs. Every workflow is assigned an ID in instantiation time. The workflow engine decides—based on the progress of the workflow—when a workflow should be sent to persistent storage and when it should be retrieved.

The progress of the workflow is determined by the progress of its component work items. The regular work item is the type of work item that may take an arbitrary time to complete; thus it is the one that reports the progress. Although this mechanism is hidden inside the WorkItem class, you should be aware of its existence to understand what causes a workflow to be paused.

The workflow engine sends a pause message to any workflow that takes longer than a specified time to report its progress. The time can be configured through the sleep threshold parameter in the control tab of the local configuration tool. When a workflow is paused, it is safe to store the state of the workflow. The workflow will be resumed after an interval specified by the activation threshold parameter in the control tab of the local configuration tool. After its state is restored, the workflow engine sends a resume message to the workflow so that it can resume execution. The arrival of an external event can also trigger the resumption of a paused workflow. If a workflow is sleeping because all ongoing work items are waiting for an event, only the arrival of an event can cause it to resume.

Work Item Life Cycle

Internally, every component can be in one of four states: nonstarted, processing, complete, and cancelled. Only the complete state is visible outside a component.

The nonstarted state indicates that the component has not yet received a token and thus has not yet started task execution.

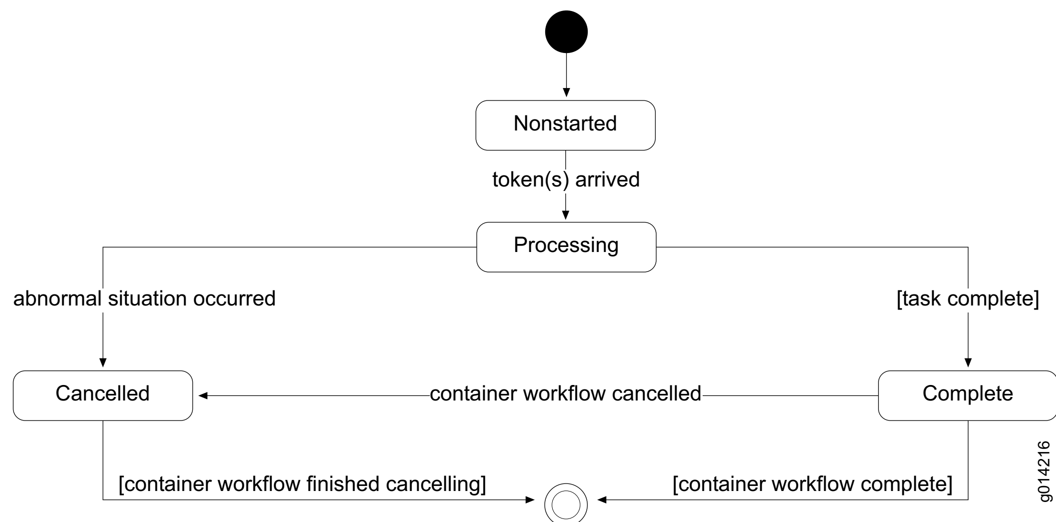
The processing state means that the component task is in progress.

The complete state indicates that the component has completed its task and sent the output token to its successors.

The cancelled state is an abnormal state that a component enters if it is unable to complete its task or has been ordered by its container to abort the task. The latter occurs when some other component inside a workflow has been cancelled or the workflow engine received a message to do so.

A component completes its task when it terminates the task in a normal condition; the meaning of normal depends on the component. A terminated task could have succeeded in performing its job or not, but in any case the component can be cancelled. A workflow cannot be cancelled after it has completed the task, because it ceases to exist when it communicates its completion to the workflow engine. Figure 7 summarizes the mechanism represented as a state machine in Unified Modeling Language (UML) notation.

Figure 7: Work item life cycle represented as a state machine in UML notation



When a component cancels, it has to undo anything that may affect the external world, such as directory operations or e-mails. Cancellation ensures that the whole workflow will be cancelled and that all work items are cancelled in the reverse order of their execution.