Nova Bonita documentation

BonitaTeam ()

- September 2008 -

Table of Contents

Introduction

This documentation is intended for Bonita administrators, architects and developers. It introduce the Bonita v4 architecture, presents some of the main concepts in Bonita and also provides some useful installation and configuration instructions. If you are already familiar with previous Bonita versions you will find in the last chapter a change history between those versions and Bonita v4.

Chapter 1 , General information describes the new version Bonita v4 called Nova Bonita

Chapter 2 , Prerequisites focus on the hardware and software prerequisites

Chapter 3, Concepts describes main workflow concepts

Chapter 4 ,Configuration and Services describes main configuration features and default services

Chapter 5, Installation guide guides you on installing the Bonita v4.0

Chapter 6, Developer's Guide guides you through the descovery of Nova Bonita functionalities.

Chapter 7, Change history between Bonita v3 and Bonita v4

Chapter 1. General information

1.1. Nova Bonita introduction

Nova Bonita is the name of new version of Bonita v4.

“Nova” technology is based on the “Process Virtual Machine” conceptual model for processes. The Process Virtual Machine defines a generic process engine enabling support for multiple process languages (such BPEL, XPDL…).

On top of that, it leads to a pluggable and embeddable design of process engines that gives modelling freedom to the business analyst. Additionally, it enables the developer to leverage process technology embedded in a Java application.

For more information about the Process Virtual Machine, check Nova Bonita FAQs [http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/FAQ] on the Bonita web site [http://bonita.objectweb.org].

1.2. Feature list

Nova Bonita (aka Bonita v4) is a lightweight workflow/BPM solution that provide XPDL 1.0 support. Nova Bonita V4.0 comes with an enhanced XPDL extension module, a rich workflow API, support for iterations and deadlines, multiple variables types support, activities multi instantiation, processes versioning and a set services such configurable journal, history and timers services.

This 4.0 version adds minor features enhancements to the runtime (deploy of single xpdl files, activities multi-instantiation clean up, log levels harmonization, iterations and subprocesses refactoring, versioning at both package and process levels...) and adds major improvements and new features to both Designer and Console applications. This version also benefit from the QA activity: code cleaning, validation, stress tests, unit tests coverage and performance improvements. Together with that some new examples (WebSale, Approval Workflow web application and Multi Instantiation of activities samples) has been added and synchronized with graphical tools (Designer and Console) allowing to deploy and execute them via a java application or through the Console.

This final version also includes an improved documentation: in addition to the reference and development guides, this version comes with two new documents: the Quick Start and Console guides.

*Note: Nova Bonita source code project structure is now using "maven". This move has no impact into the Bonita distribution as this one continue to be based on "ant". However, this will allow "maven" users to easily integrate Nova Bonita distribution in their projects as a maven dependency.

Hereafter you can find the list of features included in this 4.0 version (also listed features available in intermediate releases):

- Powerful workflow API covering deployment, definition, runtime and history workflow data
- QuereyAPsI vs RuntimeAPIs for advanced resources (hooks, mappers and performer assignments)
- Standard (J2SE) vs Enterprise (J2EE) deployment
- JEE deployment includes support for both 1.4 and 1.5 standards
- Support for XPDL 1.0 activities : Join, Split, Activity (Route, implementation no and subFlow) in both automatic and manual execution modes

- Support of main XPDL 1.0 elements : Datafield, DataType, Participant, Transition, RedefinableHeader, Transition Restriction, Package...
- Support of advanced entities/resources: Hooks, mappers, performer assignments and activities multi-instantiators (via XPDL extended attributes)
- Persistent execution (through a configurable persistence service, hibernate by default)
- Subprocesses support
- Activities multi-instantiation support
- Iterations/cycles support
- Activities deadlines support through the Process Virtual Machine generic and configurable Timer service
- Configurable journal and history workflow modules: history db vs history xml implementations
- Advanced process deployment capabilities including ".bar" file deployment and local vs global resources (hooks, mappers, performer assignments and instantiators)
- Processes and package versioning
- Standard security service based on JAAS LoginModules: Test, standard and J2EE login modules are included in the package
- Tasks (aka manual activities)assign and re-assign capabilities
- Unified life cycle for workflow activities (XPDL activities types) execution handling synchronization with Tasks, also known as manual activities, life cycle.
- Task Management module handling init, ready, executing, finished, dead, suspend and resume states
- Transitions conditions advanced support based on BeanShell scripting language and multiple variables types
- Workflow data: both process and activity level variables support
- Integer, String, Float, Boolean, Date and Enumerated types are supported as variable types
- Default mapper implementation: Initiator Mapper
- Process Virtual Machine technology based
- BPM Designer (ProEd):
  - Eclipse and Desktop versions
  - "Easy workflow project" creation wizard available in Eclipse version
  - Graphical support for advanced Nova Bonita entities: hooks, mappers, performers and instantiators
  - Support for multiple variables types
  - "Smart" conditions editor: graphical definition of complex conditions based on multiple operators and variables types
  - Automatic generation of start and end BPM steps

- BPM Console
  - Web 2.0 console supporting both desktop and traditional portal layout modes
  - Monitoring vs Worklist applications (portlets)
  - Internal user repository handling access rights to applications
  - Automatic generation of forms vs customized forms
  - Console customization capabilities: on the fly page creation, add/remove applications and widgets, look and feel...
  - Applications (portlets) and widgets support

1.3. Restrictions

Nova Bonita V4.0 comes out with an innovative architecture based on a generic and extensible engine, called "The Process Virtual Machine" and a powerful injection technology allowing services pluggability.

Nova Bonita V4.0 includes support for elements defined in the XPDL 1.0 standard. Next versions will add support for XPDL 2.0 standard coverage as well as the following new features: asynchronous activities execution, process changes (instance modifications), native clustering... Check the roadmap [http://wiki.bonita.objectweb.org/xwiki/bin/view/Main/Roadmap] for more information.

This release does not support the following features available in Bonita v3:

- Block activities
- Process definition and process modifications via Java APIs (process changes on the fly)
- Hooks: processes hook (onInstantiate) as well as activity onCancelled hook are not yet supported

Chapter 2. Prerequisites

2.1. Hardware

A 1GHz processor is recommended, with a minimum of 512 Mb of RAM. Windows users can avoid swap file adjustments and get improved performance by using 1Gb or more of RAM

2.2. Software

- Nova Bonita has been successully tested in the following environments (should work in others but those ones are part of Nova Bonita continuous integration infrastructure):
  - Operating Systems
    - Solaris-10 (SunOS 5.10) x86
    - GNU/Linux kernel 2.6.25-2 x86 Debian
    - Windows XP
  - Java Virtual Machines (jdk 1.5 and 1.6):
    - Sun-jdk1.5.0_13
    - Jrockit-R27.1.0-jdk1.5.0_08
    - Ibm-java2-i386-50
    - Jrockit-R27.2.0-jdk1.6.0
    - Sun-jdk1.6.0_06
  - Relational Databases:
    - Mysql-server 5.0.51a-6
    - Postgresql 8.3.3-1
    - Oracle 11.1.0
    - H2 1.0.76
    - HSQL 1.8.0.7
  - Application Servers:
    - Tomcat 5.5.26
    - JOnAS 4.8.6
    - Jboss-4.2.2.GA
    - JOnAS 5.0.5 snapshot
    - Jboss-5.0.0.CR2

    - Easybeans 1.0.1

- Nova Bonita requires Apache Ant 1.6.5 or higher. Apache ant will allow users to deal with configuration and administration operations

  It can be downloaded from http://ant.apache.org

Chapter 3. Concepts

3.1. Terminology

Bonita is an XPDL compliant workflow solution, so most of the concepts presented in this section are the ones included in the XPDL specification. Please, refer to this specification for more details. In the following lines we will briefly introduce those concepts and explain in details the way in which they are leveraged in Nova Bonita:

- **Package** issued from XPDL acts as a container for main workflow objects that can be shared by multiple workflow processes.
- **Process** (called Workflow Process into XPDL) contains the elements that make up a workflow: activities, data fields, participants, transitions, ...
- **Activity** is the base workflow entity to build a process. It contains others sub entities that will determine the behavior of the activity (the implementaion : no or subflow, the start mode: manual/automatic, the performer, the performer assignment, the transition restrictions : Split or Join).
- **Task** is a runtime object created as a specific activity type which is also called manual activity. Workflow tasks could be managed by an independant module receiving tasks from other applications.
- **Participant** is an actor in a workflow process. The following types are supported: SYSTEM, HUMAN, ROLE. Participants are associated to tasks.
- **Transition** is a dependency expressing an order constraint between two activities. The notion of loop (also called iteration) is also represented via a transition.
- **Variable** (aka Workflow Relevant Data in XPDL) is a workflow unit of data. Variables can be local to an activity or global to the process or a package. Nova Bonita support the following data types: Enumeration, String, Float, Integer, Boolean, Datetime, Performer.
- **Hook** is user defined logic adding automatic or specific behavior to activities and workflow processes
- **Mapper** is a unit of work allowing dynamic role resolution each time an activity with human task behavior is created (instantiated).
- **Performer assignment** is a unit of work adding additional activity assignment rules at run time.

Each one of those entities are leveraged by both BPM definition and runtime environments. Definition data, runtime recorded data and archived data are so automatically managed by the engine. To easily play with those three aspects that characterizes workflow entities, Bonita has introduced UUID's (Universally Unique Identifier). Each entity has its own typed UUID that can be used when doing operations at both definition and runtime sides through the Bonita facade API's.

3.2. Package

Package element in an XPDL definition file contains: processes, participants, datafields..... The idea is to put together multiple processes definition in one single XPDL file that will be deployed once in the engine. These processes can share participants and datafields.

Process deployment implies to deploy at least a package (ie. the XPDL file that contains the Package element). Package is the minimal unit of deployment.

The notion of package concerns also the scope of deployed java classes/artifacts (i.e hooks, mappers and performer assignments). If java classes are deployed within a package, they will be visible for all processes included in this package (XPDL file). Undeployment operation of a package will undeploy all these classes as well. Java classes can also be deployed globally (meaning at workflow server level) and so be acceded by any process/activity of any package.

QueryDefinitionAPI provides access to deployment and undeployement package related data.

3.2.1. LifeCycle

A package has its own life cycle:

- **Deployed:** When deployment operation is successfully executed (through the Management API), state of the package is **deployed**.
- **Undeployed**: when undeploy() operation is successfully performed, the state of the package becomes **undeployed**. An 'undeployed-package-handler' (refer to section Configuration and Services for more info) is then called. This handler is responsible for storing undeployment related data into the archive/ history repository through the default environment configuration.

3.2.2. Versioning

Packaging versionning is fully supported in this version. The Bonita API provides operations allowing to deploy and undeploy different package versions as well as to retrieve useful data from those packages.

The only contraint on regards to the version for the deployment is the following:

- It's forbidden to deploy two times in sequence a package with the same package Id if the version is the same in both packages (XPDL constraint)

3.3. Process

Processes are defined within a package and deployed into the engine by deploying the package.

3.3.1. Process Basics

- **Process definition**: contains the workflow definition logic (elements that makes up a workflow). A process definition is instantiated.
- **Process Instance**: represents a specific execution of a workflow process. It may run as an implementation of an activity of type subflow.

3.3.2. Life Cycles (process/instance)

A process has the following life cycle:

- **Deployed**: when the package containing the process is successfully deployed, the process is created into the engine and its state is **deployed**.
- **Undeployed**: when the process is successfully undeployed via the undeployment of the package containing the process its state become **undeployed**.

A process instance has the following life cycle:

- **Initial**: once the process instance has been created, state is set to initial.

- **Started**: when instantiateProcess() method of the RuntimeAPI is called, firstly the instance is created (**Initial** state) and secondly the execution is automatically started which causes the state of the instance to become **started**.
- **Finished: when the execution has reached the "bonitaEnd" activity (last activity in a workflow process), the instance state is set to finished.**

3.3.3. Process definition & process Instances

Common BPM scenarios are focused on the re-use of a process definitions; in these scenarios, a long-time is spent when defining a generic process model that instantiates in the same way many times. These processes are called business processes (aka process models in Bonita).

A process is a specific definition of a process that may be instantiated multiple times. These processes are based on a process-instance workflow paradigm. Processes are created into the the engine via the **ManagmentAPI** (deployment operations giving an XPDL file). J**ava DefinitionAPI** allowing the creation of a process via a java API is not yet supported for this release.. When the process is created, the workflow users are able to instantiate the workflow process via the **RuntimeAPI** to create process instance(s), execute assigned tasks... Once the process instance(s) are created, workflow participants can access the **QueryRuntimeAPI** to accomplish the following: obtain their "ToDo", "done", "suspended" lists, or access the **QueryDefinitionAPI** to get definition (as a complement of runtime data) informations about process, activities, tasks, instances, participants ....

A process keeps track of all its instances. That is, all instances of this process are retrievable through the **QueryRuntimeAPI** operations.

Bonita instantiation mechanism:

At process instantiation time a new process object issued from a deployed process definition is created. This object is initialized with definition elements and specific parameters such as variables. A root execution object pointing to the process object is created and started. It causes the execution to point to (and enter into) the first activity of the process. The root execution references a ProcessInstance object representing the runtime data being recorded in the journal all along the life of the process instance.

3.3.4. Versioning

Versionning of processes is fully supported. The only contraints on regards to the version for the deployment is:

- It's forbidden to deploy two processes with the same process id in the same package even if the versions are not the same (XPDL constraint)

3.3.5. Concept of Hooks

Hooks are user-defined logic that can be triggered at some defined point in the process life cycle. Process Hooks **are not supported in this version**.

Process hooks types are:

- OnInstantiate hook is called when a workflow instance is created. The OnInstantiate hook is not considered to be in the same transaction as the process instantiation action.
- OnFinish hook is called automatically after workflow instance termination ends.
- OnCancel hook is called automatically when a user or a workflow administrator decide to cancel a workflow instance

3.3.6. SubProcesses

Sometimes, an independently existing business process can take part in another more sophisticated process. Instead of redefining the activities, edges, properties, and hooks in the parent process, the independent process may run as an implementation of an activity of type subflow. As the execution logic is inside the subProcess, the subProcess activities are started and finished automatically by the Workflow engine according to the subProcess state. **Creating a SubProcess Activity:** When a subProcess activity is defined in the process, a specific activity with subflow behavior is created (process id of the process, local variables, in/out/in-out parameters of the sub...).. **Instantiating a Process with a SubProcess Activity:** At runtime, the execution enter into the subflow type activity, the following operations are done:

- An instance of the process referenced by the subflow activity is created.
- A new root execution is created into this instance and is automatically started (then execution enters in the first activity of the subflow).
- Local variables of the subflow activity (defined through extended attributes in XPDL 1.0) are created as global variables of the instance in the subflow (this is the default way to pass variables to a subflow when processes are defined using the **ProEd editor**). At the end of the subflow execution, global variables are automatically propagated to the parent process as local variables of the subprocess activity.
- If both **formal parameters** into the subflow process and **actual parameters** into the subflow activity have been defined, the list of actual parameters are mapped to the formal parameters (these XPDL definitions are supported by Bonita engine but not yet by ProEd editor).

3.4. Activities

Activity has no state (only started and finished dates are managed). Notion of state at this level depends only on the activity types: Task, Subflow, Route, Automatic (detailed below). When the execution enters into the activity it executes the logic (behavior/type) of this activity.

3.4.1. Activity Basics

The activity is the basic unit of work within a process.

Bonita engine is supporting all kinds of activities specified within XPDL 1.0. Those activity types are the following:

- **Manual activity** (startMode = manual, Implementation = No): When the execution enters into a manual activity a **task** object (aka human task or user task) is created. QueryRuntimeAPI allows to get access to the task according to the task state. RuntimeAPI allows to manage the task state (start, suspend, resume and finish operations). In further releases tasks could be managed by an external and pluggable task module that will allow tasks creation comming from a workflow engine as well as from any other applications such a forum, an online manager....
- **Automatic activity** (startMode = automatic, Implementation = No) : the activity is automatically executed by the engine.
- **Route activity** (Route element): Route are specialized to express Transition Restriction (e.g. SPLIT is automatically executed by the engine.
- **Subflow activity** (implementation = Subflow startMode = automatic): the activity is automatically executed by the engine.
- **BlockActivity** (BlockActivity element): the activity references an ActivitySet (set of activities) ans is automatically executed by the engine.

In addition to information determining the activity type, additional informations can be added to the activity definition depending on its type (this data can be accessed via the QueryDefinitionAPI):

- Name and Id: Id is unique within the process.
- Transition Restriction: logic of control for incoming or/and outgoing transtions (e.g. Split and Join). This attribuite applies to any activity type. Routes behaviour only relates to Transition restriction.
- Performer: actor defined in charge of the activity (Human or Role)
- Deadline: duedate for a particular activity
- Advanced Bonita features defined using extended attributes: local variables, role mapper, performer assignment
- Description, documentation, icon

Runtime (recorded) data concerning activities is divided in a common part and a body. the common part is represented by an object called "ActivityInstance". This object is returned by operations included in the queryRuntimeAPI. The "Body" relates to the specific behaviour of each activity (TaskInstance, SubflowBody, RouteBody and AutomaticBody are the types supported)

Route and subflow activity types can't execute hooks. Only Tasks and Automatic activities types are able to execute them. Refer to section Hooks here below for more details about hooks usage.

3.4.2. Life cycle for tasks (aka manual activity)

Only tasks has its a own life cycle. this life cycle is composed by the following states:

- **Ready**: This is the state of an activity ready to be started. There are two possible situations for this state to occur:
  - Activity for which there is no ingoing transitions
  - Ingoing connected activities of a particular activity are successfully finished and transition conditions were evaluated to true
- **Initial**: This is the default state of tasks that are not yet ready to be executed
- **Executing**: An activity under execution.
- **Suspended**: An activity having that was either in Ready or Executing state that has been suspended. Resume operation put back the activity with its initial state.
- **Finished**: An activity that has successfully finished.

3.4.3. Transition between Activities

Most of the usual transition patterns can be achieved through Nova Bonita Workflow. There is no special activities to achieve these patterns; however, any activity can behave as a routing node (obviously the case for Route activity). The transition pattern depends on the Transition Restrictions (e.g. Join, Split) and transition conditions defined. Transition Restrictions can be one or both of the followings:

- Join: describing the semantics of an activity with multiple (>= 1) **incoming** transitions;
- Split: describing the semantics of an activity with multiple (>= 1) **outcoming** transitions.

For **Join**, possibles types are:

- **AND** (also known as "synchronize join"): the activity is not initiated until the transition conditions on all incoming routes evaluate to true.
- **XOR** (also known as "asynchronous join"): No synchronisation is required. the activity is executed when the first execution enter the activity.

For **Split**, allowed types are:

- **AND**: the number of child executions that will be created depends on the number of outgoing transitions and the conditions (evaluated to true) associated with each transition.
- **XOR**: A single transition route is selected. when evaluation the conditions on the outgoing transitions the first one evaluated to true is taken.

> **Note**
>
> The current version of Nova Bonita Designer, only generates Join types . If more than one ougoing transitions is set (without Split within the Transitiion Restriction), an implicit Split And is constructed by the engine.

The transition patterns can be refined by defining conditions in edges between activities. A condition operates on the value of one or more variables of activities, and is expressed in Java. Any java expression statement is valid. Assuming that the variable “Var” is defined for a given activity, any of the following constructs is a valid condition: (str1.compareTo("initial value") == 0) && (enum1.compareTo("yes") == 0) && (float1.compareTo("1.0") == 0) && (int1.compareTo("123") == 0) && (boolean1.compareTo("true") == 0) && (date1.compareTo("2008-09-25T13:14:58") == 0)

3.4.4. Iterating Activities

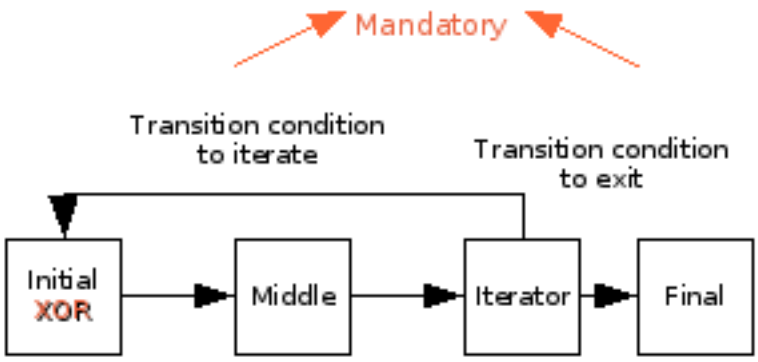
Bonita supports both structured (one entry and exit point in the cycle) and arbitrary cycles/iterations. Nova bonita iteration feature is natively supported through the use of XPDL Transition element: a loop may be represented via a transition that returns to an activity that was on a path that led to the transition.

The following Guidelines explain how to design iterations in Nova Bonita:

Premise: It is not possible to continue execution inside iterations and exit at the same time.

1. Only a single iteration is allowed between two connecting nodes.
2. All transitions exiting from a node starting the iteration must meet a condition. If there is more than one transition for exiting from that node, all transitions must meet a condition.

Following figure illustrates a typical design that matches our model:



To guarantee the premise, the iteration condition and edge condition must be exclusive. This means that when one is true the other is false. Only iterations from Iterator to Initial are possible. Conditions can be a group of conditions like: (((…) && (…)) || (…)). (Remember: only a single iteration between nodes is allowed.) There could be another iteration starting in the Iterator activity going to Middle or to Iterator itself. Iterations from Iterator to Final activity are not allowed because a cycle does not exist. An edge condition from Iterator to Final activity is strictly necessary and must be the opposite of the iteration condition. If there are multiple edges outgoing from Iterator to other activities, all of them must meet a condition not equal to the iteration condition (this is necessary to accomplish the above premise)..

Note that:

- the initial node must be XOR type to allow to validate its entry condtion either at the first entry and then to continue into the cycle. At each time only one execution can enters into the node.
- If the Iterator point has not Split Xor transition restriction a Warning is produced when deploying the process. Only a Split Xor setting can avoid threads of execution goes both outside the cycle and inside the cycle which is not authorized.
- Join XOR inside iterations (meaning that multiple branches have been created that may include tasks) do not cancel/delete non selected execution path (meaning that the tasks created during the past iterations may still be acceeded by end users).
- At each cycle of the iteration, for activities inside the iteration path, new runtime records (ActivitiyInstance interface) are created/recorded (with distincts iteration id). If iterations have been performed the parameter: iteration Id must be specified into some methods of the QueryRuntimeAPI.

3.4.5. Activity multi-instantiation

Multi-instantiation of activities is a new porwerful feature introduced by Nova Bonita.. This feature covers all types of activities previously described in section 4.1. The idea is to determine at runtime the number of instances to create for a particular activity. This feature is really useful in situations in which, at definition time, process designers do not know in advance the number of occurences of a particular activity to be created.

The principle is based on the execution of an "Multi-Instantiator" class (added to the activity definition) that returns an object containing:

- a list of values which size is determining the number of instances to be created and so executed. This list of values is used to set for each created activity instance a dedicated activity variable (this activity variable is also added to the definition of the activity);
- the number of finished instances expected to take the transition (called joinNumber). This number must be **greater than 0** and **lesser than or equal to** the number of created instances.

Caution

This version has a restriction on the joinNumber. Only: **equal to** criteria must be taken in account to match the behavior of the engine.

The condition on the outgoing transition must not contain local variables.

Here after an example of multi-instantiation definition within activity definition.

```
<Activity Id="Approval" Name="Approval">
   -.../...
  <ExtendedAttribute Name="MultiInstantiation">
```

```
              <Variable>performer</Variable>
              <MultiInstantiator>org.ow2.bonita.example.aw.instantiator.ApprovalInstantiator</
MultiInstantiator>
  </ExtendedAttribute>
  <ExtendedAttribute Name="property" Value="performer" -/>
   -.../...
</Activity>
```

The class given into the defintion must implement the interface: MultiInstantiator. Refer to the javadoc API for MultiInstantiator interface (org.ow2.bonita.definition package) as well as the developpementGuide for more details.

3.4.6. Concepts of hooks/connectors

Hooks in Nova Bonita Workflow context are external java classes performing user-defined operations. Hooks may be called at different moments in the activity lifetime. Only two types of activity allows setting of hooks: manual (task) and automatic activity. Hooks are prefixed by the type of activity for which they are associated.

In activities of type task, hooks may be called at different moments of the taks lifecycle:

- task:onReady: is called when the task becomes available.
- Task:onStart: is called as soon as the task is started.
- Task:onFinish: is called as soon as the task is finished.
- task:onSuspend: is called when the task is suspended by a user.
- task:onResume: is called when the task is resumed from a suspended state.

In automatic activities, hooks can only be called at one moment (when the activity is executed) :

- automatic:onEnter

Refer to the developpementGuide to get more details on hooks (writing, compiling, deploying hooks as well as samples).

Note

For deadline feature, the name of the class that implements the hook interface is specified within XPDL Deadline element. The event name (aka Hook type is called ON_DEADLINE.

3.4.7. Activity/Hooks and Transactions

Hooks are always executed within the transaction involved in the last activity change state (i.e. instantiateProcess, startTask, finishTask, suspendTask...). Transaction can involve more than one activity in synchronous executions (typically a task connected to one or more automatic activities). If the execution of the hook raises an exception, it will abort or not the transaction depending on the implemented interface (Hook vs TxHook):

- **Hook** interface, if an exception occurs it is catched by the engine and no rollback is performed. Hook interface is intended to execute methods of APIs that are allowed to perform read/query operations : QueryDefinitionAPI and QueryRuntimeAPI.
- **TxHook** interface, if an exception occurs it is raised by the engine and the transaction is not commited (rollback). TxHook interface is intended for executing Bonita APIs operations that are related to write/

set operations: RuntimeAPI, ManagementAPI, DefinitionAPI (not yet supported), CommandAPI. This interface should also be used to call business logic in which transactions are involved.

Refer to the javadoc on API for Hook/TxHook interfaces (org.ow2.bonita.definition package) as well as the developpementGuide for more details on Fault Management and on Activity/Hooks and Transactions features.

3.5. Role Mappers Feature

3.5.1. Overview

This feature can be added in the XPDL definition as an extended attribute inside the **"Participant" element of type Role**.

```
        <Participant Id="manager" Name="manager">
          <ParticipantType Type="ROLE" -/>
          <ExtendedAttributes>
            <ExtendedAttribute Name="Mapper" Value="Custom" -/>
            <ExtendedAttribute Name="MapperClassName"
Value="org.ow2.bonita.tests.functionnal.mappers.AdminRoleMapper" -/>
          </ExtendedAttributes>
        </Participant>

       -..../....

       <Activity Id="myTask" Name="myTask">
          <Implementation>
            <No -/>
          </Implementation>
          <Performer>manager</Performer>
          <StartMode>
            <Manual -/>
          </StartMode>
       -..../....
```

This feature is dedicated to manual activities (human tasks) where a "Performer" (element of XPDL Activity) referencing such type of participant has been defined. When the task runtime is created the mapper feature is executed allowing the dynamic resolution of the participant that will be assigned to the activity. A **list of candidates users** for the task is filled-in with the returned values of the role mapper. In the case that the mapper is an 'Initiator Instance' mapper (default implementation) the user that creates the instance is assigned to the task (this default implementation is useful for testing purposes).

Two types of mappers are available, depending on the method employed to retrieve users in the system.

- Calling a java class to request a users base (**Custom** mapper).
- Getting the initiator of the process instance (**Instance Initiator** mapper)

Role Mappers can be defined the ProEd editor application. The Bonita API allows to retrieve related data of performers defined in a XPDL file. The QueryDefinitionAPI allows to get the role mapper definition (getProcessParticipant operation).

Note

- The execution of a mapper for a particular participant is performed each time a task is created (only for tasks having this performer assigned)
- If there is no role mapper defined for participant of type role, no assignment of task is done (meaning, the taks is not yet assigned, but it could be done afterwards through the API)

3.5.2. Custom Mappers

This mapper type allows to perfeclty match with the users-roles mappings and constraints available in organizations. When this type of mapper is selected, a call to a java class is performed. This java class must implement the "RoleMapper" interface. In particular the "searchMembers" method of this interface must be implemented and will return the collection of expected users (see the javadoc of interface RoleMapper). The class name is specified into the extended attribute with Name MapperClassname. Refer to the section called "Mappers"4.2 in the developpementGuide to get more details on the practical steps to follow to define and deploy mapper classes into Nova Bonita (developping, compiling and deploying steps)..

3.5.3. Instance Initiator

This type of mapper fills in the candidates list of a task with the user that created the workflow instance (based on the authenticated user that initiates the instance). This user be able to perform operations to this the task.

3.6. Performer Assignment

3.6.1. Overview

This feature extends first assignment rules for tasks that was done through mappers. Mappers resolutions can assign a task to a list of possible candidates users (those ones are able to see and perform operations over the task). This list of candidates can be refined for each particular activity through the use of performers assignments elements. Depending on the type of performer assignment the following functionnalities can be added:

- Assign the activity (only tasks) to a user by calling a java class in charge to perform a user selection from the list of candidates (**Custom** performer assignment).
- Dynamically assign the activity to a user by using the value of a variable that has been previously set with the selected user Id (**Variable** performer assignment).

Once the performer assignment has been performed, the task is assigned to the selected user. This feature can be added though the ProEd editor application. QueryDefinitionAPI allows read access to the definition of the performer assignment.

3.6.2. Custom Performer Assignment

The java class must implement the "PerformerAssign" interface. In particular the "selectUser" method of this interface must be implemented. The return value of this method is the name of the selected user (see the javadoc of interface PerformerAssign). The class name is specified in an XPDL "extended attribute" called Name PerformerAssign. Refer to the section Performer Assignments in the developpementGuide to get details on the practical steps to deploy and define performerAssign classes into Nova Bonita (developping, compiling and deploying steps). Notice that the candidates list is passed to the "selectUser" method to simplify the user selection. Of course other strategy could be considered.

3.6.3. Variable Performer Assignment

With this type, assignment depends on the value of a variable previously set.

Chapter 4. Configuration and Services

This chapter introduces the services configuration infrastructure provided by Nova Bonita as well as main services included in this 4.0 version.

4.1. Services Container

The Process Virtual Machine technology includes a services container allowing the injection of services and objets that will be required during workflow definition and execution. Objects and services used by the Bonita engine are defined through a XML file. A dedicated parser and a wiring framework are in charge of creating those objects. Security, identity, persistence, notifications, human task and timers are examples of pluggable services.

This services container (aka IoC container) can be configured through a configuration file. A default configuration file is included in the package under the /conf directory (environment.xml):

```
<environment-definition>

    <environment-factory>
      <hibernate-configuration name='hibernate-configuration:core' >
        <properties resource='hibernate-core.properties' -/>
        <mappings resource='bonita.mappings.hbm.xml' -/>
        <cache-configuration resource='bonita.cache.xml' usage='read-write' -/>
      </hibernate-configuration>
      <hibernate-session-factory name='hibernate-session-factory:core' configuration='hibernate-
configuration:core' -/>
      <variable-types resource='bonita.type.resolver.xml' -/>
      <job-executor threads='1' auto-start='true' -/>
      <command-service>
        <retry-interceptor -/>
        <environment-interceptor -/>
        <standard-transaction-interceptor -/>
      </command-service>
      <api  type='AutoDetect' -/>
      <chainer name='finished-instance-handler'>
        <object class='org.ow2.bonita.services.handlers.impl.DeleteFinishedInstanceHandler' -/>
        <object class='org.ow2.bonita.services.handlers.impl.ArchiveFinishedInstanceHandler' -/>
      </chainer>
      <chainer name='undeployed-package-handler'>
        <object class='org.ow2.bonita.services.handlers.impl.ArchiveUndeployedPackageHandler' -/
>
      </chainer>
    </environment-factory>

    <environment>
      <journal name='journal' class='org.ow2.bonita.persistence.db.DbJournal'>
        <arg><string value='persistence-service:core' -/></arg>
      </journal>
      <history name='history' class='org.ow2.bonita.persistence.xml.XMLHistory'>
      </history>
      <!-- DbJournal cannot be shared by several environments.
      It contains a session cache that needs to be recreated for each environment --->
      <chainer name='recorder'>
        <recorder class='org.ow2.bonita.persistence.log.LoggerRecorder' -/>
        <ref object='journal' -/>
      </chainer>
      <chainer name='archiver'>
        <archiver class='org.ow2.bonita.persistence.log.LoggerArchiver' -/>
        <ref object='history' -/>
      </chainer>
      <!-- Query Api has an object reference to the journal,
      so it cannot be shared by multiple environments --->
      <queryApi name='queryList'>
        <ref object='journal' -/>
        <ref object='history' -/>
      </queryApi>
      <!-- DbRepository cannot be shared by several environments.
```

```
      It contains a session cache that needs to be recreated for each environment --->
      <repository class='org.ow2.bonita.persistence.db.DbRepository'>
        <arg><string value='persistence-service:core' -/></arg>
      </repository>
      <timer-session -/>
      <transaction -/>
      <job-db-session session='hibernate-session:core' -/>
      <hibernate-xpdl-persistence-service name='persistence-service:core' session='hibernate-
session:core'/>
      <hibernate-session name='hibernate-session:core' factory='hibernate-session-
factory:core' -/>
    </environment>

  </environment-definition>
```

Currently, following objects implementations can be injected in the environment:

- **repository:** data repository storing workflow processes, instances, activities... Db persistence (class org.ow2.bonita.repository.db.DbRepository) implementation is included in this version.
- **recorder**: object responsible of workflow execution logs. Default implementation handles workflow logs in the command line console (org.ow2.bonita.persistence.log.LoggerRecorder). Recorder and Journal (see next) objects can be chained (new ones can be added as well on top of the recorder chainer). This give you a powerful mechanism to handle workflow execution data
- **journal**: object responsible for storing or retrieving workflow execution data. Db persistence (class org.ow2.bonita.persistence.db.DbJournal) implementation is provided by default.
- **archiver**: object intended for workflow logs archiving. Default implementation handles logs on workflow data archiving through the default implementation (class org.ow2.bonita.persistence.log.LoggerArchiver). Archiver and History (see next) objects can be chained (new ones can be added as well on top of the archiver chainer). This give you a powerful mechanism to handle workflow archived data
- **history**: object intended for storing or retrieving workflow archieved data. Default implementation is provided and available in the following class: org.ow2.bonita.persistence.xml.XMLHistory. This class will store workflow history in the file system as XML files
- **queryList**: object intended to configure how the QueryRuntimeAPI will retrieve the workflow execution data. This retrieval could be configured to chain with the expected order into the journal and the history.
- **finished-instance-handler**: action to perform when a workflow instance is finished. This object could chain two distinct actions: for a given workflow instance, deleting the runtime object including its tasks from the repository and then store data in the archive and remove data from journal. Default implementations are proposed for both chained actions.
- **undeployed-package-handler** action to perform when a workflow package is undeployed. Default implementation is proposed allowing to store undeployment related data into the archive
- **api-type**: defines the execution context for Bonita: JSE vs JEE (1.4 or 1.5). Possible values are "AutoDetect", "Standard", "EJB2" and "EJB3". "AutoDetect" meaning that the runtime is able to detect the deployment envirobment (i.e if Nova Bonita is running on a JEE application server, EJB2 will be consider as the default execution context). Execution context is a key concept as it give to Nova Bonita the right information about security (JSE vs JEE) and APIs types (POJO vs Session Beans).

* Note 1: As explained before persistence objects are provided as default implementations in the environment. Notice that in a persistence configuration additional resources are required, i.e for hibernate persistence you can specify mapings, cache configuration...

* Note 2: The environment is divided in two different contexts: application and block. Objects declared inside the application context are created once and reused while objects declared inside the block context are created for each operation.

Hereafter you will find a second example of environment.xml configuration in which the history service is configured to archive the data into a relational database rather than use the default XML implementation:

```
<environment-definition>

    <environment-factory>
      <hibernate-configuration name='hibernate-configuration:core' >
        <properties resource='hibernate-core.properties' -/>
        <mappings resource='bonita.mappings.hbm.xml' -/>
        <cache-configuration resource='bonita.cache.xml' usage='read-write' -/>
      </hibernate-configuration>
      <hibernate-session-factory name='hibernate-session-factory:core' configuration='hibernate-
configuration:core' -/>
      <variable-types resource='bonita.type.resolver.xml' -/>
      <hibernate-configuration name='hibernate-configuration:history' >
        <properties resource='hibernate-history.properties' -/>
        <mappings resource='bonita.mappings.hbm.xml' -/>
        <cache-configuration resource='bonita.cache.xml' usage='read-write' -/>
      </hibernate-configuration>
      <job-executor threads='1' auto-start='true' -/>
      <hibernate-session-factory name='hibernate-session-factory:history'
configuration='hibernate-configuration:history' -/>
      <command-service>
        <retry-interceptor -/>
        <environment-interceptor -/>
        <standard-transaction-interceptor -/>
      </command-service>
      <api  type='AutoDetect' -/>
      <chainer name='finished-instance-handler'>
        <object class='org.ow2.bonita.services.handlers.impl.DeleteFinishedInstanceHandler' -/>
        <object class='org.ow2.bonita.services.handlers.impl.ArchiveFinishedInstanceHandler' -/>
      </chainer>
      <chainer name='undeployed-package-handler'>
        <object class='org.ow2.bonita.services.handlers.impl.ArchiveUndeployedPackageHandler' -/
>
      </chainer>
    </environment-factory>

    <environment>
      <journal name='journal' class='org.ow2.bonita.persistence.db.DbJournal'>
        <arg><string value='persistence-service:core' -/></arg>
      </journal>
      <history name='history' class='org.ow2.bonita.persistence.db.DbHistory'>
        <arg><string value='persistence-service:history' -/></arg>
      </history>
      <!-- DbJournal cannot be shared by several environments.
      It contains a session cache that needs to be recreated for each environment --->
      <chainer name='recorder'>
        <recorder class='org.ow2.bonita.persistence.log.LoggerRecorder' -/>
        <ref object='journal' -/>
      </chainer>
      <chainer name='archiver'>
        <archiver class='org.ow2.bonita.persistence.log.LoggerArchiver' -/>
        <ref object='history' -/>
      </chainer>
      <!-- Query Api has an object reference to the journal,
      so it cannot be shared by multiple environments --->
      <queryApi name='queryList'>
        <ref object='journal' -/>
        <ref object='history' -/>
      </queryApi>
      <!-- DbRepository cannot be shared by several environments.
      It contains a session cache that needs to be recreated for each environment --->
      <repository class='org.ow2.bonita.persistence.db.DbRepository'>
        <arg><string value='persistence-service:core' -/></arg>
      </repository>
      <timer-session -/>
      <transaction -/>
      <job-db-session session='hibernate-session:core' -/>
```

```
      <hibernate-xpdl-persistence-service name='persistence-service:history' session='hibernate-
session:history' -/>
      <hibernate-session name='hibernate-session:history' factory='hibernate-session-
factory:history' -/>
      <hibernate-xpdl-persistence-service name='persistence-service:core' session='hibernate-
session:core'/>
      <hibernate-session name='hibernate-session:core' factory='hibernate-session-
factory:core' -/>
    </environment>

  </environment-definition>
```

This environment file, available under the /conf directory (environment-db-history.xml file), replace the default XMLHistory implemntation of history service by the DBHistory one. A new hibernate configuration is also added to manage history DB persistence. This allow to easily configure journal and history data to be stored in two different databases if needed.

4.2. Services

Services in Nova Bonita is all about pluggability. Standard (StandAlone Java based) and Enterprise (JEE Server based) versions of Nova Bonita can be easily configured thanks to the services container. To allow that, each workflow related service has been thought in terms of an interface with different possible implementations. In the following lines you will find a description of main services supported in Nova Bonita:

4.2.1. Persistence

Persistence is one of key technical services injected into the services container. This service, as well as other major services in Nova Bonita, is based on a service interface. That means that multiple persistence implementations can be plugged on top.

The Persistence service interface (called DbSession) is responsible to save and load objects from a relational database. By default, a persistence implementation based on the Hibernate ORM framework (called HibernateDbSession) is provided (JPA and JCR would be other examples of persistence implementations).

The Process Virtual Machine core definition and execution elements (processes, nodes, transitions, events, actions, variables and executions) as well as the XPDL extension ones (join, split, manual and automatic activities, conditions, variables...) are persisted through this service. Process Virtual Machine core elements are also cached by leveraging the default persistence service implementation (Hibernate based). Workflow packages, processes, instances, tasks and advanced classes (such hooks or mappers) are are stored through this persistence service. Workflow repository is the term used in Nova Bonita to store those entities.

4.2.2. Identity

Identity service main objective is to give freedom to system administrators to leverage their favorite organization user repository. Traditional user repositories such LDAP, ActiveDirectory as well as any other user repository (database or API) can be plugged as implementations of this service.

By default, some user repositories implementations are provided for testing purposes: in memory, basic FileSystem based persistence, and basic database persistence (based on a predefined database schema). Those implementations can also be used in production if there is no other user repository available.

The Identity service is so an extensible interface (known as IdentityServiceOp) build around three main concepts: Users, Groups and Memberships:

- User: a particular user inside an users repository. Users can be created, modified, removed and queried (some of those operations could be not allowed for some repositories (i.e LDAP) through the IdentityService API).
- Group: a group of users in a particular users repository. A group could contain either users security restrictions or hierarchical information. As for users, groupes can also be created, removed, modified and queried.
- Membership: a membership represents a user position in a particular group. An user could have two different membership in two different groups. Membership related operations concern set, remove or updates on users position inside groups.

Both Security and Human Task services will use the Identity one by checking user login/password and user rights (Security) and by resolving workflow logical roles with users and so to assign manual activities to users based on some hierarchical information (Tasks Management)

By default, Nova Bonita is packaged with a test based identity module based on a properties file. This file contains the user/login allowed to reach Nova Bonita APIs. This properties file is in fact a Test Login Module (see security module description below), meaning that the same properties file is used for security and identity configuration.

4.2.3. Security

The security service is based on JAAS standard. Main purpose of this service is to provide both authentication and authorization capabilities to the workflow engine. As security directly relates to users permissions, this service also relates to the identity one (commonly security is configured on top of the identity service).

As for other services, the Nova Bonita team is concerned on let you the freedom to choose and plug your favorite security implementation. At the same time we also want to provide one ore more default implementations that allow users to quickly set up and start playing with Nova Bonita.

For testing purposes Nova Bonita includes a default JAAS login module checking user/password values stored in a file. This easily allow to start playing with Nova Bonita in a testing security environment in which the login module acts as a lightweight users repository. This login module (org.ow2.novabpm.identity.auth.PlainLoginModule) is the one provided in Nova Bonita examples directory.

The current implementation of the security service allows you directly work with the default identity service to handle users authentication. Users must login before start calling the Bonita APIs.

The Security service is composed by two different JAAS LoginModules. The first one (called PlainLoginModule) is responsible to handle security authentication and authorization. This one could just be replaced by you favorite JAAS Login Module. The second one (StorageLoginModule) is responsible to keep data of authenticated users (basically for security context initialization). Those login modules can be configured in both standard and enterprise environments (note that most of JEE servers already provides a Storage Login Module so you could just replace the one proposed by Nova Bonita by the one leveraged by you app server). Some examples of security configuration files for both standard and enterprise environments are included in the Bonita distribution (under the /conf directory).

4.2.4. Task Management

Task management is all about providing the right information to the right people at the right time !. This is one of the most important services that must be provided by a workflow solution.

As human task management can be re-used in other domains (not only by workflow solutions but by any Java based application) we wanted those features to be a service rather than an internal workflow module. As a result, this service is generic and extensible Task Management service that can be either used in Nova Bonita extension to handle manual task assignments and executions or either by any Java application or Domain Specific Language (i.e BPEL4People extension for instance).

Traditional features such users - roles/group mapping, delegation, scalation, task deadlines handling or manual activities execution life cycle are in scope of this service. Advanced features such configurable activity life cycle, interactions with other task managements system, services or collaborative applications and integration with organizational rules are also part of the main responsabilities of this service.

The current implementation focus on support of manual tasks (also known as manual activities) in Nova Bonita. Basic features such Bonita RoleMappers and Performer Assignments entities allowing users - roles mapping are already supported. Together with the identity and security service, users can login into the system, get their tasks todo list and execute them. As other service in Nova Bonita this module is executed in a persistent environment.

4.2.5. Journal and History

This module concerns the way in which the workflow data is stored during the workflow execution and archived when the execution is completed. This is indeed a crucial module in a workflow solution.

While in Bonita v3 journal data (aka execution workflow data) and history data (aka archived data) where handled by different mechanism, in Nova Bonita we decided to unify them as the underlying essence of both is to handle workflow data. For that to be done, we created the concept of workflow record. A record is a minimal set of attributes describing a workflow entity execution. That means that each workflow entity related to the execution has its own associated record: instance record, task record, hook record...

Those records are recorded during the workflow execution and stored depending on the persistence service implementation (db, xml...). The Nova Bonita API will retrieve record data from the records storage and sent them back to the users (meaning that records also acts as value objects in Nova Bonita APIs).

As soon as a workflow instance is finished, a typical scenario would be (by default) to move instance related workflow data from the production environment to a history one. While the physical device and the data structure could changed from one workflow engine deployment to another (XML, BI database...), the internal format could remain the same (records). This is exactly what is happening in Nova Bonita, when archiving data the engine just move execution records from the production to the history environment without data transformation inbetween.

4.2.6. Timers

To handle activities deadlines, a timer service is required that can schedule timers to be executed in the future. Timers must have the ability to contain some contextual information and reference the program logic that needs to be executed when the timer expires. A typical scenario would be a manual activity (task) that needs to be monitored with a timer. For example, if this task is not completed within 2 days, notify the manager.

This service, as well as any other asynchrous service in Nova Bonita is based on the Process Virtual Machine Job executor framework. Job executor framework is responsible for handling jobs. A job could be a timer scheduling or an asynchronous message for instance. When a job is created and stored in the database, the job executor starts a new transaction, fetch the job from the database and perform the instructions contained in the message.

Chapter 5. Installation guide

5.1. Installation

Nova Bonita V4.0 adds support for both standard and enterprise deployments. After unzipping this release you could easily use Nova Bonita "as a library" inside your web or rich client application or to deploy it into you favorite application server and use it remotely.

So, first of all you should start by unzipping the Bonita distribution package:

```
>unzip bonita-4.0.zip
```

A new directory `bonita-4.0` will be created with the following structure:

```
README
build.xml
build.properties
License.txt
release_notes.txt
conf/
doc/
javadoc/
examples/
ear/
lib/
```

Let's describe those items :

- README

  This file gives the basic information related to Nova Bonita

- build.xml

  This file is an "ant" file (aka makefile) that provides tasks to deal with Nova Bonita administration operations (detailled commands instructions are given in following sections).

- build.properties

  This file contains the J2EE properties required to deploy and to use Nova Bonita APIs deployed in a remote J2EE server (JOnAS, Jboss and EasyBeans properties are provided by default allowing to execute Nova Bonita samples remotely).

- License.txt

  The license of Nova Bonita. Bonita is released under the LGPL license v2.1.

- conf/

  This directory contains default configuration files for Nova Bonita. That includes "environment" xml files (including services and objects used as default by the engine), login modules configurations (JAAS compliant login modules samples) and hibernate persistence configuration (as a default implementation to handle Nova Bonita persistence). Standard (JSE) and Enterprise (JEE) versions are provided for JBoss and JOnAS application servers as well as with Easybeans EJB3 container

- doc/

  This directory contains Nova Bonita documentation:

- referenceGuide.pdf

  This is the document that you are actually reading. The bonita reference guide gives a detailed overview of features, architecture as well as installations and configurations instructions.

- developmentGuide.pdf

  This document describes the design and development process in Nova Bonita Workflow. The designer (ProEd) application is covered in this document. You will also find useful information about how to develop BPM connectors (aka, hooks, mappers, performer assignments and instantiators)

- quickStartGuide.pdf

  This guide covers main features of Nova Bonita, and will help you to get started right away. This document is intended for users looking to get started quickly on Nova Bonita runtime and graphical tools.

- javadoc/

  This directory contains developer's javadoc documentation of Nova Bonita. This javadoc describes in details Nova Bonita APIs

- examples/

  This directory contains workflow examples provided with Nova Bonita package. Those samples applications illustrates how to use Nova Bonita APIs from within a client application. That includes the process definition (XPDL) files, java related workflow artifacts (Hooks, mappers and performer assigns) and client applications which illustrates how to deploy, execute and query workflow processes through Nova Bonita APIs in both JSE and J2EE environments.

  - Approval Workflow sample

    This is a generic Approval Workflow process. Two versions of this process exists. One with a single approval performed by the manager (which decides whether he accepts or rejects an hypothetical request) and an other involving the multi-instantiation for the approval step followed by a "CheckDecision" step (to decide what should be the decision). Both sample applications show how hooks entities can be used by both manual and automatic activities. The version with multi-instantiation illustrates additional features like activities multi-instances coupled with a performer assignment.

    An advanced version of this sample is also included. In this version Nova Bonita engine is packaged in a .war file together with the approval workflow sample and a simple web application. This sample illustrates how simple it is to embeed Nova Bonita in a web application running in a web container (i.e Tomcat). See "Java Properties" chapter below for more details about Tomcat configuration for Bonita

  - Carpool sample

    This example is a carpool simulation in which requesters and publishers are put in relationship to each other. This process illustrates the way in which deadlines and asynchrous services can be leveraged in Nova Bonita.

  - WebSales sample

    This example is a web sale simulation process in which a customer and an online shop agent/employee are involved in a purchase request process. In this sample iterations/loops as well as multiple types of variables are illustrated.

- lib/

  This directory contains the libraries used in Nova Bonita V4.0. Nova Bonita can be integrated in your application/IS in different ways (integrated in a web application, inside a rich client application, remotly deployed in a JEE application server...). Depending on your integration environment only some of those libraries will be required.

5.2. Standard vs Enterprise installation

Find hereafter some instructions about how to deploy and to reach Nova Bonita in both Standard and Enterprise environments:

5.2.1. Standard installation

To integrate Nova Bonita in your application (i.e web or desktop java application) you only need to add a couple of libraries to your environment.

The following ones are the Nova Bonita main dependeces:

```
bonita-api.jar
bonita-core.jar
pvm.jar
bsh.jar
novaBpmIdentity.jar
novaBpmUtil.jar
xstream.jar
```

To leverage the Nova Bonita default persistence service based on hibernate, you should also need those ones:

```
hibernate3.jar
dom4j.jar
commons-logging.jar
commons-collections.jar
hsqldb.jar
cglib.jar
antlr.jar
asm.jar
asm-attrs.jar
geronimo-jta.jar
```

5.2.2. Enterprise installation

Move to the Nova Bonita installation directory and:

- call "ant ear.ejb2", "ant ear.ejb3" respectively tasks to generate the bonita.ear file corresponding to either JEE 1.4 or 1.5 specification.
- deploy this ear into your favorite JEE 1.4 or 1.5 application server.
- *Note: If you are using Jboss or JOnAS application servers or the EasyBeans EJB3 container you can directly deploy and start using Nova Bonita examples. Specific descriptors and classpath configurations for those servers are included in this distribution. In case you are using another JEE 1.4 application server (Weblogic, Websphere, Oracle, Geronimo, Glassfish...) just add specific descriptors for those application servers into the bonita.ear file and configure your client side to reach the Bonita APIs (take as example existing configurations for JOnAS and Jboss).

  bonita.ear file generated through ant "ear.ejb3" task can be deployed in any EJB3 compliant application server. in this version of the specification, standard descriptiors should work in any EJB3 environment.

5.2.2.1. Jboss 4.x and 5.x installation and deployment

Find hereafter required steps to deploy and run Nova Bonita in Jboss apps:

- Download Jboss 4.x or 5.x from Jboss web site: http://www.jboss.org and follows jboss application server installation instructions
- Edit build.properties file and set your Jboss configuration settings: URL provider (localhost with port 1099 by default) and set "jboss.home" and "jboss.lib" properties values (those properties must be initialized with the path corresponding to the directory in which Jboss was installed and the path in which jboss client libraries are available).
- Type "ant ear.ejb2" or "ant ear.ejb3" (depending on the jboss version you want to use) under your Bonita installation directory to generate the bonita.ear file
- Copy the bonita.ear file generated into JBOSS_HOME/server/default/deploy directory (default configuration)
- Start the Jboss application server by executing run.bat or run.sh under JBOSS_HOME/bin directory. Bonita should be deployed at that time
- Bonita is now up and running under Jboss

5.2.2.2. JOnAS 4.x and 5.x installation and deployment

Find hereafter required steps to deploy and run Nova Bonita in JOnAS apps:

- Download JOnAS 4.x or 5.x from JOnAS web site: http://jonas.ow2.org and follows JOnAS application server installation instructions
- Edit build.properties file and set your JOnAS configuration settings: URL provider (localhost with port 1099 by default) and set "jonas.root" and "jonas.lib" properties values (those properties must be initialized with the path corresponding to the directory in which JOnAS was installed and the path in which JOnAS client libraries are available).
- Type "ant ear.ejb2" or "ant ear.ejb3" (depending on the JOnAS version you want to use) under your Bonita installation directory to generate the bonita.ear file
- Copy the bonita.ear file generated into JONAS_ROOT/apps/autoload directory (default configuration)
- Start the JOnAS application server by executing "jonas start.bat" or "jonas start.sh" under JONAS_ROOT/bin directory. Bonita should be deployed at that time
- Bonita is now up and running under JOnAS

5.2.2.3. EasyBeans EJB3 Installation and deployment

Hereafter you will find steps required to deploy Nova Bonita (JEE version) in EasyBeans EJB3 container:

- Download easybeans up to v1.0 at http://www.easybeans.net/xwiki/bin/view/Main/Downloads (standalone version). EasyBeans is using a directory called 'easybeans-deploy' in the basedir to deploy new archives
- Create a directory with this name in the folder from where you will start easybeans container and copy the bonita.ear file in to the created 'easybeans-deploy' directory

- Be sure to have all security permissions in your java.policy file: permission java.security.AllPermission
- Add novaBpmIdentity.jar in your CLASSPATh environment variable. This jar is available under /lib directory of Nova Bonita distribution
- Then start easybeans :

  java $JAVA_OPTS -Djava.security.manager -Djava.security.policy="path to your java.policy" -jar "easybeans jar file"

Chapter 6. Developer's guide

This chapter describes how to start playing with Nova Bonita V4.0. More precisely it describes main steps to define and deploy workflow processes and how to start running them in Nova Bonita:

- How to create a process definition
- How to deploy a process definition
- How to develop a simple application by leveraging Bonita APIs

For end users, there is a dedicated document: quickStartGuide.pdf that illustrates the steps described above using a 100% graphical environment.

6.1. Designing a xpdl process with ProEd

Like in previous Bonita versions, processes could be created either through a java api or through a graphical editor : Nova Bonita designer (aka ProEd). The java api to build Bonita v4 processes is not yet developped, so processes should be deployed as .xpdl files. That can easily be done under ProEd and then imported as xpdl files. For that, you can use either the Desktop or the Eclipse based versions that are available to download on the Bonita download [http://forge.objectweb.org/project/showfiles.php?group_id=56&release_id=302/] page.

Please, refer to the Nova Bonita development guide (under /doc directory) to learn more about how to use ProEd.

Once a process definition is completed, ProEd saves the description of the designed process as an xpdl file that should be imported as described in the following section.

6.2. Nova Bonita APIs

If you are already familiar with previous Bonita versions and you have already developped your own applications on top of Bonita, we want to minimize your effort when migrating to Bonita v4. Compatibility from Bonita v3 to v4 is one of our main concerns.

At the same time, we took the chance to review and to improve Bonita v3 APIs in this new major version so basically the Bonita v3 API spirit is still there but we applied some improvements in Nova Bonita to simplify some operations and to add added value features.

Nova Bonita APIs has been refactored each intermediate release until become stable in this final version.

6.2.1. Getting started with Bonita APIs

Nova Bonita APIs are divided into 5 different areas:

- **DefinitionAPI**: to create/modify major process elements into the engine (packages, processes, activities, role mappers, variables by calling java methods instead of importing xpdl files. It will allows also to modify the execution of runtime elements such as tasks and instances.
- **QueryDefinitionAPI**: to get useful workflow definition data from a particualr process definition: packages, processes, activities, role mappers, ...
- **RuntimeAPI**: to manage process, instance and task life cycle operations as well as variables set/updates

- **QueryRuntimeAPI**: to get recorded/runtime informations for packages, processes, instances, activities, tasks (support for dynamic queries will be added in the future). It allows also to get tasks (aka manual activities) per state for a particular user and as well to get/list workflow instances.variables.

- **ManagementAPI**: to deploy workflow processes into the engine. XPDL files and advanced entities such hooks, mappers and performer assignments can be deployed individually or in one shot

There's also a generic API that allow to execute specific commands (queries) that could be needed (and not available in the APIs) in the context of a particular workflow based application:

- **CommandAPI**: to allow developpers to write and execute its own commands (mostly queries)

6.2.2.1. Nova Bonita APIs, playing with !

Nova Bonita V4.0 is an extensible and embeddable workflow solution that can be easily deployed in both standard (JSE) and Enterprise (JEE) environments.

- Nova Bonita can be easily integrated in your application as a workflow library.

- Nova Bonita can also be deployed in a JEE application server and so reached remotely by external applications

6.2.2.2. Nova Bonita local vs remote applications !

An utility class has been provided to unify access to Bonita APIs from both local and remote environments and so to avoid the use of lookups in JEE deployments: org.ow2.bonita.util.AccessorUtil. Through this class, Nova Bonita APIs can be reached in a unified way in both local and remote applications. From a developer point of view Nova Bonita APIs are just Java POJOs.

The AccessorUtil objet will also be leveraged inside BPM artifacts such hooks, mappers or performer assignments to deal with Nova Bonita APIs.

By configuration you can specify the way in which the Bonita APIs will be reached, the system property called "org.ow2.bonita.api-type" must be defined at client side to specify whether the APIs will be reached locally or remotely (possible values are "Standard","AutoDetect",'EJB2" and "EJB3").

There are two available methods in the AccessorUtil class (under the org.ow2.bonita.util package):

- getQueryAPIAccessor(): returning the helper class QueryAPIAccessor giving access to QueryRuntimeAPI and QueryDefinitionAPI interfaces.

- getAPIAccessor(): returning the helper class APIAccessor giving access to QueryRuntimeAPI, QueryDefinitionAPI, RuntimeAPI, ManagementAPI, DefinitionAPI, CommandAPI interfaces.

Hereafter you will find an example on how to use the AccessorUtil class from your client application:

```
        RuntimeAPI runtimeAPI = AccessorUtil.getAPIAccessor().getRuntimeAPI();
        QueryRuntimeAPI queryRuntimenAPI =
AccessorUtil.getQueryAPIAccessor().getQueryRuntimeAPI();
```

You will find some samples applications leveraging this API under the /examples directory. For a detailed insight on Nova Bonita APIs, please take a look to the Nova Bonita javadoc apis (available under /javadoc directory)

6.3. Running the examples

The Nova Bonita package contains some complete workflow examples. Those examples can be easily compiled, deployed and executed (in both JSE and JEE modes), thanks to a set of "ant" tasks available on examples directory. Hereafter you will find some information about how to play with one of those examples (ApprovalWorkflow sample). For others (carpool and websales samples), you can proceed the same way by replacing the references to approval workflow by the one corresponding to your sample name:

- The Bonita basic Approval Workflow: a simple Approval Workflow application illustrating the workflow definition, workflow deployment and execution phases.
- The Bonita Multi-instance Approval Workflow: a version of the Approval Workflow application in which the "approval" activity is dynamically assigned to a set of users at runtime. This example illustrates activities multi-instantiation paradigm.

The build.xml in the root directory (BONITA_HOME/examples/approvalWorkflow directory) contains required targets to compile and launch the example in both standard (JSE) and enterprise (JEE) environments:

The Approval Workflow sample is configured to run together with the default hibernate persistence service. Main Process Virtual Machine entities, XPDL extension as well as execution related data will be persisted in a relational database. By default Nova Bonita embeeds HSQL database and uses it as a default database. You can easily change this default configuration and use your favorite database by modifiying the hibernate-core.properties file located under "conf" directory.

```
>ant aw-standalone
```

This sample application leverages the Security and Identity services so you must provide a right user login/passwd to run the sample. The default identity module (based on a properties file) is provided with three users ("john", "jack" and "james" logins with "bpm", "bpm" and "bpm" as password). All three can login into to system but only john and jack are able to play in the Approval Workflow sample. This behaviour is due to the users - role mapping defined in the previous workflow sample.

The java example simply deploy the xpdl file as well as advanced java entities such hooks and mappers in the engine (deployBar method in ManagementAPI), then creates a workflow instance and calls getTasksList, startTask and finishTask methods from the RuntimeAPI

This sample application can be launched at both standard and enterprise modes. In the enterprise mode Nova Bonita APIs are available as Stateless Session Beans. Enterprise sample version can easily be executed with the following ant tasks:

```
>ant aw-jonas4 (for a deployment in JOnAS application server version 4)
```

```
>ant aw-jboss4 (for a deployment in JBoss AS version 4)
```

```
>ant aw-eb (for a deployment in EasyBeans EJB3 container)
```

*Note that the enterprise version requires Nova Bonita (bonita.ear file) to be deployed in an application server (see Enterprise Instalaltion chapter of this Guide to know more about how to deploy Nova Bonita in a JEE application server).

As for basic Approval Workflow sample, you can easily launch the multi-instantiated version as follows:

```
>ant aw-multiInst-standalone
```

Multi-instantiation version of Approval Workflow can also be launched remotely. Please type "ant -p" to get an overview of remote execution tasks.

An advanced version of this sample is also provided. This version package the approval workflow sample as well as the Nova Bonita engine (standalone mode) in a war file together with a simple web application. This war file can be deployed in a web container (i.e Tomcat) as an standalone workflow application. This web application allows to deploy, undeploy and create instances of a deployed workflow sample to illustrates how to reach Nova Bonita APIs from a web application (API calls will directly leverage the Bonita POJO based API).

In order to generate this war file, just type:

```
>ant war (for aw.war file generation)
```

once this file is generated, just deploy it in your favorite web container (i.e Tomcat). The web application will be available by default at http://localhost:8080/aw if you are using Tomcat.

6.4. Java Properties

There are couple of considerations that must be taken into account when using Nova Bonita in both JSE and JEE versions. Security and workflow environment (aka environment.xml related files) configurations should be set at both server and client sides. Those JAVA properties are concerned:

- org.ow2.bonita.environment = URL to the xml file containing the workflow environment configuration or filepath to this file or resource path

  Nova Bonita engine environment lookup mechanism will check for the environment location following this order: first URL, second, filepath and third resource

- java.security.auth.login.config = PATH where the login configuration is available (default configurations are available under /conf directory)

Those properties should be set at server side when running Nova Bonita in a JEE environment as well as at client side (i.e java desktop application). JAVA_OPTS environment variable can be used for this purpose.

In aIf those properties are not defined, for instance when starting JOnAS or Jboss servers, Nova Bonita will take as default the environment.xml file include in the bonita.ear.

6.5. Administration operations

The Nova Bonita distribution provides a set of administration utilities/commands that can be executed through ant tasks. Hereafter you will find main commands availiable:

```
deployBarDb    Deploy the bar file in standalone mode
deployBarEb    Deploy the bar file in enterprise mode
ear.ejb2       Generate an ear which can be use in an ejb2 environment (Jonas 4 or Jboss 4
application server)
ear.ejb3       Generate an ear which can be use in an ejb3 environment (easybeans, jonas 5,
jboss 5)
init-db        Generate database schema from the environment configuration
test-eb        Run the test suite in easybeans and put the results in target/testresults
test-jboss4    Run the test suite in jboss 4 and put the results in target/testresults
test-jboss5    Run the test suite in jboss 5 and put the results in target/testresults
test-jonas4    Run the test suite in jonas 4 and put the results in target/testresults
test-jonas5    Run the test suite in jonas 5 and put the results in target/testresults
tests          runs the test suite in database and put the results in target/testresults
undeployBarDb  Undeploy the process in standalone mode
undeployBarEb  Undeploy the process in enterprise mode
```

```
usageDeploy    The usage to successfully deploy  a bar file
usageUndeploy  The usage to successfully undeploy a bar file
```

Through those commands, workflow processes can be deployed , undeployed, J2E enterprise version can be generated, database schema can be initialized or still th test suite can be launched.

6.6. Database configuration

Nova Bonita distribution is configured by default to run with an embedded database (HSQL database). This allows you to start playing with Bonita without configuring your own database. The default database will be created in your filesystem and soon as you execute one of the samples provided in the distribution (default location would be the "java.io.tmpdir" system variable: i.e /tmp in Linux Systems or Local Settings/tmp in windows platforms).

Database configuration for HSQL is available under the /conf directory of the distribution (hibernate-core.properties file).

```
hibernate.dialect                      org.hibernate.dialect.HSQLDialect
hibernate.connection.driver_class      org.hsqldb.jdbcDriver
hibernate.connection.url               jdbc:hsqldb:file:${java.io.tmpdir}/bonita-db/
bonita_core.db;shutdown=true
hibernate.connection.username          sa
hibernate.connection.password
hibernate.hbm2ddl.auto                 update
hibernate.cache.use_second_level_cache true
hibernate.cache.use_query_cache        true
hibernate.cache.provider_class         org.hibernate.cache.EhCacheProvider
hibernate.show_sql                     false
hibernate.format_sql                   false
hibernate.use_sql_comments             false
```

This configuration (core) concerns the Nova Bonita definition and runtime information. As you can see in the /conf directory, there is also another file called hibernate-history.properties which contains the database configuration for the Nova Bonita history database. In fact process history data can be stored either in XML files or in a database (please check Configuration and Services chapter) and so this file concerns the choice of a database to store history data.

6.6.1. Changing the default database configuration

Hereafter the instructions to update the default database configuration in Nova Bonita to use your favorite releational database:

- Copy your database driver into the Nova Bonita distribution /lib directory (i.e mysql.jar, oracle.jar)
- Configure hibernate.properties file for your favorite database (i.e examples for MySQL, Oracle and Postgres are provided in both core and history proerties files). Note that you can use different databases or instances for "core" and "history" configurations. By default, if you only update hibernate-core.properties file and you keep the default history service to use the XML service (see the environment.xml file) no history data will be stored in the database.
- Open a command line and go to the Nova Bonita distribution main directory and type "ant init-db":

```
init-db:
    [input]
    [input]       Which hibernate configuration to use to generate database -?
    [input]       Default value for bonita engine database is:  hibernate-configuration:core
    [input]       Default value for bonita DbHistory database is: hibernate-
configuration:history
    [input]      []
```

This command allows Nova Bonita administrators to initialize a particular database instance with the Nova Bonita database schema (including FK and indexes). Just select the database you want to initialize (core vs history) by copying either "hibernate-configuration:core" or "hibernate-configuration:history" chains.

Note: Previous values names are correlated to the corresponding environment files (environment.xml and environment-db-history.xml) so if you change the values of those files you should take care to use the sames during the DB initialization.

Chapter 7. Change history between Bonita v3 and Nova Bonita

Main concepts and features that made the friendly usage and the Bonita v3 brand have been kept: hooks, role mapper, performer assignments, local/global variables, rich and powerful API. Most of these features have been revisited in order to become even more efficient thanks to the PVM execution environment. Aim was to be the most compatible with the last version but of course some changes are required.

Goal of this chapter is to list/focus all these differences.

7.1. Concept of package

The concept of package has been introduced by XPDL specification from the WfMC in order to be a container for main workflow objects that can be shared by multiple workflow processes that can support an overall business application. Amoung these elements are: participants, datafields, others process workflows/sublows.

This concept has been natively taken in account by Nova Bonita engine. According the requirements and needs of our customers this concept should be enforced.

7.1.1. Package life cycle

States for package: UNDEPLOYED, DEPLOYED

7.2. Processes, instances, activities and tasks life cycles

One major change concerns the adding of task entity. If the activity is manual (ie. startMode=manual) when the execution enters the graph node of the activity a task is created. This task has its own life cycle with some synchronisation with the activity entity. Within this version task is still managed by the engine but in the future, it will be possible to plug an external task module to manage the tasks.

7.2.1. Process life cycle

States for process: UNDEPLOYED, DEPLOYED

Deployment of processes implies deployment of a package. Same thing for the undeployment. Package can be deployed and undeployed several times in order to make modifications onto its contained elements (process, participants, activities, ...). This is the way to maintain processes before the introduction of versionning in next version.

See the developpement guide for more details.

7.2.2. Instance life cycle

States for process instance: INITIAL, STARTED, FINISHED

No difference with bonita v3.

CANCELED state is not yet supported.

7.2.3. Activity life cycle

Activity has no state (oly started and finshed date). Notion of state at this level depends only on the type of behavior defined within the activity. A specific body of the activity is created according the type of the activity (Task, Subflow, Route, Automatic). The state is implemented by the body. At now only task has a life cycle.

7.2.4. Task life cycles

States for task: INITIAL, READY, EXECUTING, SUSPENDED, FINISHED, CANCELED

State SUSPENDED has been introduced. This state can be reached either from READY or from EXECUTING. transition is also reciproque.

Tasks are particular types of activities (such subflow, route...) associated to human actors.

7.3. APIs

Bonita v3 APIs were divided into 5 different areas and can be compared to bonita v4 API (see Chapter 4)

- ProjectSessionBean: is covered by both DefinitionAPI (for set/add methods) and QueryDefinitionAPI (for get methods)
- UserSessionBean: is covered by both RuntimeAPI and QueryRuntimeAPI
- AdminSessionBean: has fonctions that could be found into QueryDefinitionAPI and QueryRuntimeAPI according on the type of information (runtime or definition information). At now there's no check for admin role
- UsersRegistrationBrean: is not relevant for bonita v4 because user base is not managed by the engine.
- historyAPI: is covered by QueryRuntimeAPI.

Bonita v3 can only be acceeded as a remote workflow server. Bonita v4 supports both java workflow library and remote workflow server (see Chapter 4).

A new API has also be added for improving workflow processes deployment as well as advanced entities deployment: hooks, mappers and performers assignments. This API is called ManagementAPI. No need anymore to deploy xpdl first and the compile and copy by hand advanced entities in a particular server directory. Any deployment/undeployment operation can be performed through the ManagementAPI.

Furthermore Nova Bonita v4 provides extensibility to the APIs by the addition of the commandAPI. Developper is now free to write and execute its own commands and consequently can extends the proposed API. This is a service oriented feature and it also should avoid to provide a querier language for complexe request (involving requests with multiple criteria).

7.4. Hooks

Pieces of end-users java code that are executed at particular moments of either a process instance or a task or an "automatic activity" (route and subflow type activity cannot have hook).

7.4.1. for task

Xpdl definition of hooks has changed in order to extend rollbacking capabilities to all hook types making by the way the usage of hook simpler. An example of the new one is given here after

```
<ExtendedAttribute Name="hook" Value="org.ow2.bonita.integration.hook.OnReadyHook">
  <HookEventName>task:onReady</HookEventName>
  <rollback>false</rollback>
</ExtendedAttribute>
```

The element rollback has been introduced to indicate if the hook will be or not rollbacked.

Hook events have also been adapted to match the constraints of task life cycle.

- task:onReady
- task:onStart
- task:onFinish
- task:onSuspend
- task:onResume

Main change is the suppression of before/after for terminate and before/after start types because of the introduction of the rollback parameter. An other change is the introduction of new events due to the new state: SUSPENDED.

Note: if using proEd, the designer can select for each hook:

- either rollback=true (case1)
- or rollback=false (case2)

Hooks are always executed into a transaction. In case1, if an exception has occured the exception is raised by the engine and the transaction is rollbacked. In case2, the occuring exception is catched by the engine.

To implement a hook class the developper has the choice between two interfaces. Look at the javadoc of these interfaces for more details:

- org.bonita.ow2.definition.TxHook
- org.bonita.ow2.definition.Hook

If rollback=false has been previously defined, only Hook interface can be implemented otherwise an exception is raised at runtime. Then it prevents the use of TxHook interface. These hooks are intended to execute not critical operations for the workflow. Only query API are proposed to be acceeded into the parameters of the execute() method of the interface.

If modification on hook class is required it can be hot deployed to replace the previous one (see the ManagementAPI). It can be also deployed within the bar archive or independendtly. It can be also undeployed if the class is not required by a deployed process.

7.4.2. for automatic activity

One type of event can be defined.

- automatic:onEnter

7.4.3. for process

ON_INSTANTIATE hook (set within the process element of XPDL definition) is not yet supported.

ON_CANCELLED hook is not yet supported.

7.4.4. Interactive hook

Interactive hooks (also called Bean Shell) are not yet supported for activity and process. Those hooks will be implemented soon adding support for others scripting languages such Groovy

7.5. Deadlines

Deadline feature within Bonita v4 is the same as for Bonita v3. org.ow2.bonita.definition.TxHook or org.ow2.bonita.definition.Hook interface must be implemented in case of deadline hook (ON_DEADLINE event). See javadoc for more details.

7.6. Mappers

org.ow2.bonita.definition.RoleMapper interface must be implemented (see javadoc for more details).

Main difference concerns the moment in which the searchMembers() method is executed. In Bonita v3 it was executed at process instantiation since in Bonita v4 it is at the creation of the task from which the activity has been defined with a role mapper. It has the advantage to take in account modification of the groups within the external user base

7.7. Performer assignments

org.ow2.bonita.definition.PerformerAssign interface must be implemented (see javadoc for more details).

7.8. Variables

Properties entity in Bonita v3 has been renamed to Variables in Bonita v4. This seems a more natural way to work with workflow relevant data.

Variables support and flexibility in Bonita v3 was too limited. Only String and enumerated types were supported. In Bonita v4 support for common variables types as well as as advanced ones (including own Java based ones) will be added in next releases (currently, the V4.0 version support same types than v3 plus: Float, Integer, Boolean, Datetime, Performer).

Getting and Setting variables operations directly handles Java Objects, meaning that a get operation returns an Object so the developer only needs to use the instanceOf operator to determine the type of a particular variable.

7.9. Iterations

Iterations support in Nova Bonita follows the innovative mechanism included in Bonita v3, meaning supporting complex and advanced uses cases: unstructured iterations or arbitrary cycles.

Main difference between Bonita v3 and v4 related to iterations is that in v4 there is no need anymore for a dedicated entity called iteration. Transitions can be used in Bonita v4 to create a cycle in a workflow processes.

For compatibility reasons iterations entities defined as XPDL extended attributes (Bonita v3) are still supported.

The current implementation has some restrictions:

- A cycle must have at least one XOR entry point

- Split activities as exit points are only supported in case of XOR

- Join XOR inside iterations do not cancel/delete non selected execution path

Those restrictions will be fixed in the next release with the addition of a new behaviour for XOR activities in which non selected execution path will be automatically deleted/removed