BONITA Workflow Cooperative System Application Programming Interface

(Version 2.7)

Christophe Loridan
Miguel Valdés Faura
Anne Géron

BULL R&D

CHANGES TRACK		
REFERENCES	DATE	CHANGE
1.1		Document creation
2.0	30/06/04	User Registration API documentation Integration of all API methods
2.1	16/07/04	Explanations & schema Samples integration
2.2	11/08/04	Sub Process – Sub Process activities – roles
2.3	19/08/04	Added a paragraph about transitions in activity general description Documented syntax of conditions in the description of the API Project Session Access Control paragraph was updated.
2.4	27/08/04	Process terminology was updated. Processes examples were reviewed. Projects attributes and new methods were added.
2.5	19/10/04	Added explanations about iterations in chapter 1
2.6	5/11/04	Added explanations about hooks, mappers and performers in chapters 1 and 5.
2.7	21/01/05	New chapter User Management was included

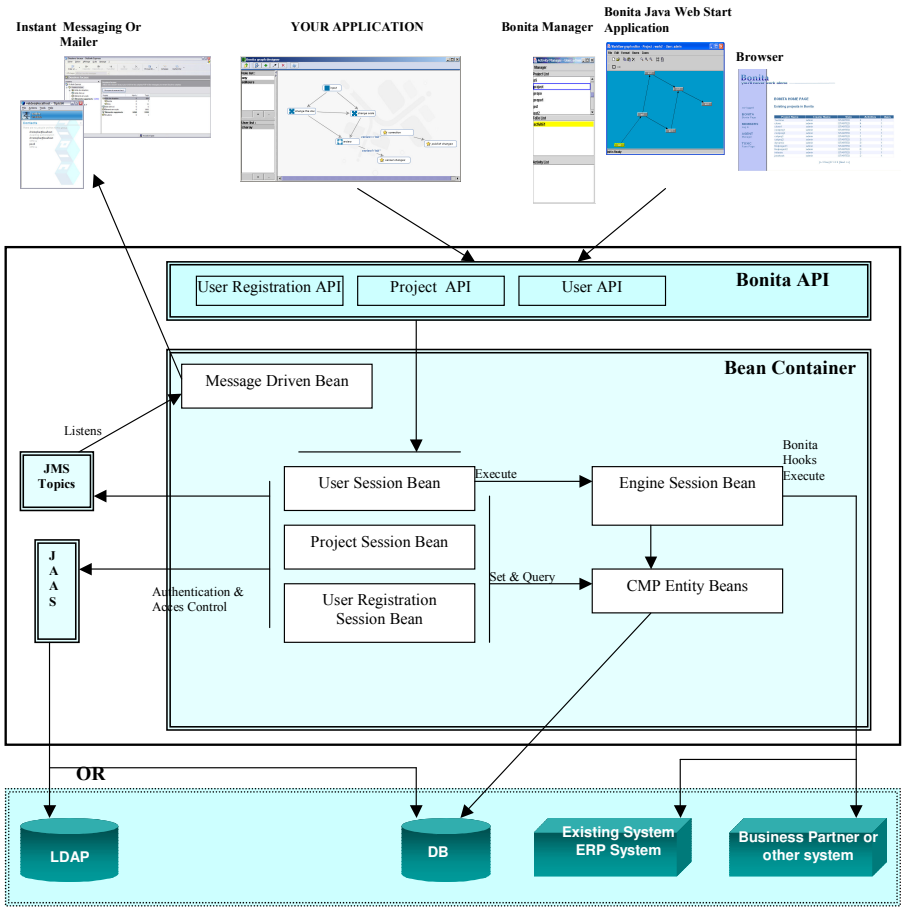
INDEX

INTRODUCTION

BONITA is a workflow system featuring a lot of innovative features like activities that can start in anticipation, awareness infrastructure allowing users to be notified of any events occurring during the execution in a given process , or automatic activation of user's code according to a defined activity life cycle. Traditional workflow features like dynamic user/roles resolution, activity performer and sequential execution are also included in Bonita to support both cooperative and administrative workflow processes.

BONITA is a fully conformant J2EE application, taking advantage of the power and robustness of the J2EE platform. The BONITA API is accessible either thru EJB's.

Processes are created using a graphical definition tool or by using the Project interface API. A process is defined as a set of activities and an associated execution model. The enactment engine takes care of scheduling the activities according to the defined execution model. The User API provides full control over the execution of the process, for example allowing starting or stopping an activity. BONITA supports also dynamic modification of an existing process, that is the Project interface can be used against a running process.



- The **User Registration Session bean** provide the interface for :

  - User creation and management
  - Group creation
- The **Project Session Bean** provides the interface for :
  - Creation of the process
  - Definition of nodes and edges
  - Modifications of properties
- The **User Session Bean** implements commands and queries related to
  - Projects of a user
  - Todo List
  - Executing activities
  - Start/terminate/Cancel commands
- The **Engine Bean** is a special session bean that implements the state machine and controls Process execution. It is not part of the API.
- Each method call in the Bonita API involving a state modification of the workflow system is registered into a JMS Topic. Depending on user preferences (defined while user creation), the **Message Driven Bean** notifies the user either using Instant Messaging services, either Traditional Mailer.

Bonita Hooks can access existing systems in the SI (Erp or whatever else), or Business partner systems using JCA or Web services.

Both User and project APIs are available either as Session Bean, or as web services.

1 CONCEPTS

1.1 Terminology

- A **process** is a set of activities. In BONITA, the term project is also used.
- An **activity** is an atomic unit of work. In BONITA, activities are also termed **Nodes.**
- **A transition** is a dependency expressing an order constraint between two activities. In BONITA, transitions are also termed **Edge**s.
- A **property** is a workflow unit of data, commonly known as workflow relevant data.
- A **hook** is a user defined logic adding automatic behaviour to activities/nodes.
- A **mapper** is a unit of work allowing dynamically roles resolution at workflow instantiation time.
- A **performer assignment** is a unit of work adding additional activity assignment rules at run time.

1.2 Process

1.2.1 Process basics

Bonita supports both cooperative and administrative workflows processes. These processes are mapped on three Bonita types:

- **Cooperative**: flexible workflow process allowing definition and execution operations just after the process is created.
- **Model**: workflow process which contains the workflow definition logic. These projects can be instantiated by users.
- **Instance:** workflow process representing a specific execution of a workflow model.

The status of a workflow process can be controlled at definition or runtime by the workflow process administrator/s. Two possible statuses are allowed for a workflow process:

- **Active**: the workflow process could be modified or executed. This is the default status for a cooperative, model or instance process.
- **Hidden**: the process is not yet available. Process definition operations are not allowed for models, cooperative and instances processes. Even more, the execution of cooperatives and instances processes are not allowed to.

1.2.2 Life Cycle

BONITA has a very simple process life cycle

- A process is **initial** once it has been created. As soon as the process is in this state, it can be controlled using both User API & Project API. The User API allows monitoring the execution of the process. Whenever the first activity has been started using the User API, the process goes to **started** state. The execution of the process is performed by the BONITA enactment engine, under control of applications thru the use of the User API.

- A process is **started** as soon as the first activity has started. While being executing, the process definition still can be modified using the Project API. When all activities are terminated, the process stays in state **started**. It still can be modified, for example new activities can be added.
- A process is **terminated** once it has been explicitly terminated by an application thru the User API. In **terminated** state, the process definition cannot be modified any more.

1.2.3 Cooperative processes

Bonita has always had a quite simple view of cooperative process enactment: once a process is defined, it is enacted ! For example, just after the creation of a process with a single activity using the Project API, you would be able to run it by using the User API and still be able to add new activities to the process definition. This brings lot of flexibility to workflow participants, and is particularly convenient for so-called cooperative (we also call them ad'hoc) processes.
You would typically set up a specific process in order to perform a given job between several colleagues very easily. To allow some level of reuse of process definition, we introduced the concept of **process clone** (see 2.2.4 clone processes).

1.2.4 Models & Instances

Obviously, there are usage scenarios where the reuse of process definition is of key importance; in these scenarios, one spends a long-time to define carefully a generic process model that will be instantiated in the same way a lot of time. We call those processes administrative processes (we also say process models).

A process model is a specific definition of a process that can be instantiated many times. They are based on model-instance workflow paradigm. In this kind of workflows, the Project API must be used to define the workflow model. When the process definition is done, the workflow users are enabling to instantiate the previous workflow model via Project API. Once model instance are created, workflow participants can access to the User API to obtain their todolist, to execute assigned activities, and so on.
A process model keeps track of all its instances, that is all instances of such a process can be retrieved thru the User API.

So, either cooperative or administrative workflows use the same component definition API that is Project API. Depending of the type of the process with which we want to create/initialize, this API must be initialized for.

There are also some differences between those workflow types concerning processes execution. Cooperative workflows are ready to be executed and modified from the creation. In the other hand, administrative workflows need to be instantiated before. The term process model is used to talk about Bonita projects defined on the context of administrative workflow use case.

In future releases of BONITA , the concept of Process model will be further extended, with the implementation of a Process Model Repository allowing import of process definition in a variety of format.

Bonita Instantiation mechanism:

Previous versions of Bonita workflow engine were "duplicating" in a new process instance the whole process model (activities, properties, edges, hooks …) as a kind of new clone of the project. This took quite a long time even for medium workflow processes, and it was a problem for users at instantiation time.
The new versions (since 1.4) have been reviewed to improve the performances. Only the Ready state activities (properties, roles and users if exist) are copied at the creation of the new instance. Once activity is started, hooks are executed from the Model Hooks (they are not copied anymore). Then, after activity termination, edges and Ready or Executable following activities are copied as well.

Note :

- A process model which has been instantiated can still be modified, but be aware that the modifications can bring errors in the instance execution,
- An instance can still be enriched, but be aware that the modifications can be in conflict with the model definition that will be applied at execution time.
- To avoid those cases you could actually set the status of the project to hidden (no modifications are allowed).

1.2.5 Clone processes

A process clone is a duplicate of an existing process. Once the cloning operation is completed, the two processes will execute completely independently.

Just after the cloning operation:

- The process instance has the same set of activities than the process model, each activity allocated to the same role than in the model. All activities are in initial state, and have the same properties than in the model, with the associated value. All activities have the same hooks and the same transition conditions than the ones in the original process.
- The process properties are the same than the process one, with the same initial value.
- The users associated to the process are the same than in the first process, and has the same associated roles.
- The process instance can be controlled without any restriction thru the User and Project APIs.
- Iterations between activities are the same than in the process model.

Process clone is available for both cooperative and administrative workflow processes.

1.2.6 SubProcesses

Sometimes, a business process which can exist independently can take part in another more sophisticated process. Instead of defining again the activities, edges, properties, hooks … in the parent process, it can be included in it as a "SubProcess" by the way of a specific kind of node.

Creating a subProcess activity:

- In this case, when the SubProcess activity is added to the process, the sub process is automatically cloned by Bonita in a new process which is given the name of the subProcess activity in the parent process, and links are maintained between the sub process and the parent Process.

Instantiating a Process with a SubProcess activity :

- While instantiating a Process with a subProcess activity, new instances of the two processes are created (Parent process and Sub process). Bonita engine assumes SubProcess node and equivalent Sub process instance to have the same name, so it automatically give the Sub process instance name to the subProcess activity in the parent process.

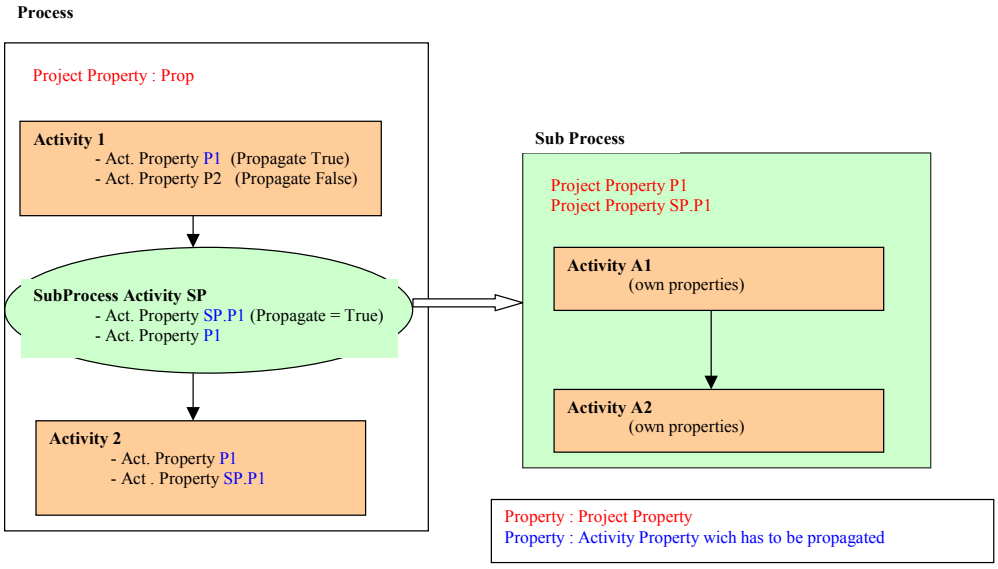
As any other activity, an SubProcess activity can be iterated as well.

Constraints :

- As in a normal process, activities, properties and hooks in the sub-process must not have the same name than an activity existing in the whole process.

Properties propagation

The properties of the Sub Process Activity in the global Process are propagated as Process properties in the Sub Process, as it is shown underneath:



1.2.7 Relationship to Users

A process has an associated set of Users. Such an user has access to the corresponding process, meaning

- He knows about the existence of the process.
- He can take over roles that exist in the scope of the process.
- He can be notified of various events occurring in the process.
- He can control the execution of the process.

Users assuming the Admin role can modify the definition of the process. The role Admin is specific to each process, that's means the role Admin for the "process1" is different than the role Admin for the "process2".

The User on behalf of whom the project has been created is automatically assigned the Admin role, he is then responsible for the creation of other users in the process, and to allocation of role to those users (including the Admin role that can be allocated to several users).

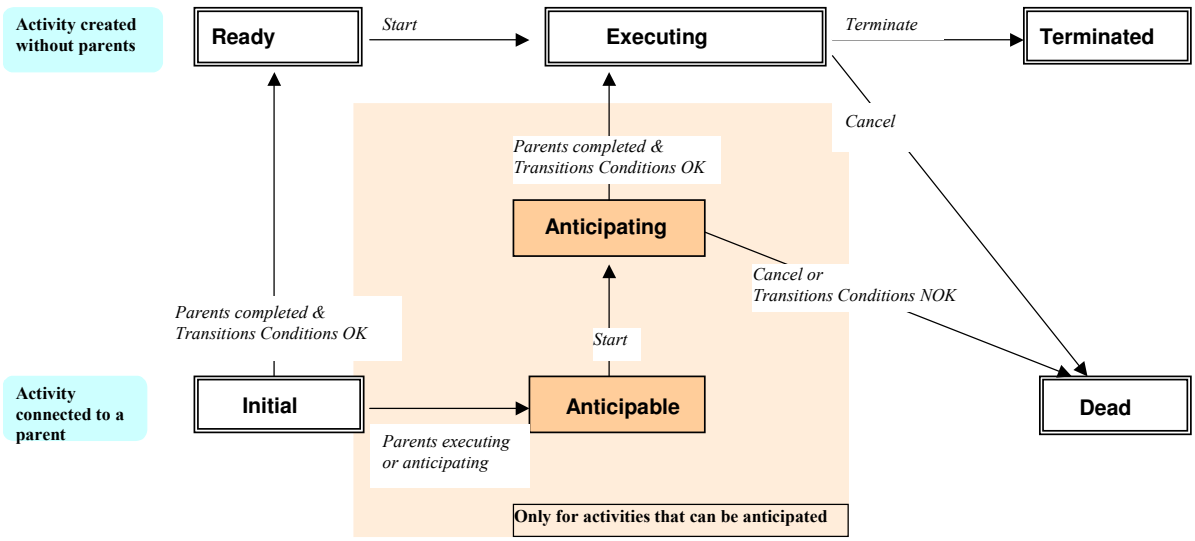
1.3 Activities

1.3.1 Activity basics

The activity is the basic unit of work within a process.
Execution of an activity can either be **automatic**, or **manual** :

- **Automatic** : in this case the BONITA enactment engine will start it as soon as the applicable transitions from preceding activities are successfully evaluated.
- **Manual** : the BONITA enactment engine will not start a manual activity until some application has explicitly started it thru the User API.

The life cycle of an activity is then as follows :

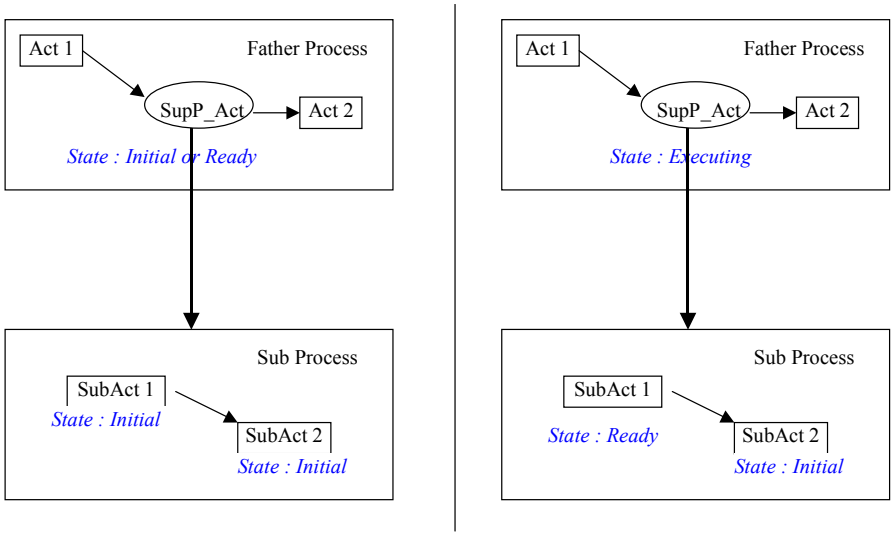


- **Ready :** This is the state of an activity ready to be started. There are two possible situations when this occurs. In the first one, an activity has no parent activity (so is the first activity of the workflow process). In the second one, a normal activity has parent activities that have all terminated successfully, and whose transition condition to the activity has been successfully evaluated.
- **Initial :** This the state of an activity waiting for some processing to complete before being ready to run. In case of normal activities, at least one of the parent activities is still executing. In case of activity that can be anticipated, at least one of the parent activities has not started.
- **Anticipable:** This is the state of an activity that can be started, without waiting for its parent's activity to complete. All the parents' activities must be started however.
- **Anticipating:** A previously anticipable activity that has been started. Automatic activities are automatically transitioned from anticipable to anticipating, manual activities must be explicitly started. An anticipating activity cannot be terminated until all its parent activity has themselves terminated, and the transition conditions have been successfully evaluated.
- **Executing:** An activity being executed.
- **dead:** An activity that has been cancelled. All the depending activities will be automatically cancelled. Cancellation occurs in two cases: explicit cancellation, or unsuccessful evaluation of one of the inwards transition condition.
- **Terminated**: An activity that has been successfully terminated.

For **automatic activities**, BONITA will **automatically**

- (*Non anticipable activities*) Transition the state from ready to executing ,
- (*Anticipable activities*) Transition the state from anticipable to anticipating ,
- (*Anticipable activities*) Transition the state from anticipating to executing whenever all the parents complete
- Launch the executing hook
- Terminate the activity whenever the executing hook has complete

In case of activities involved in a sub process, the life cycle is as described below :



Any **activity** is associated with a **role**. All the users being allocated that role in the scope of the process have the possibility to take over the activity.

Any **activity** is enclosed in a **Transaction**, and every call to a method of the Bonita API which change the state of an activity is considered as part of a transaction (that means every method but those beginning with “getxxx” which only retrieve information).

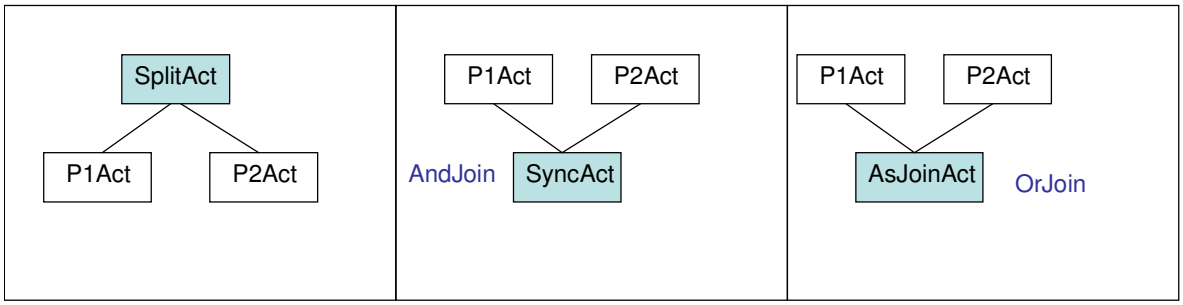
1.3.2 Transition between activities

Most of the usual transition patterns can be achieved with BONITA. There is no special node to achieve these patterns; rather any activity can act as a routing node.

The transition pattern will be determined according to the type of the activity, which can be one of AND-JOIN (also known as "synchronize"), or OR-JOIN (also known as "asynchronous join").

The transition pattern will also be determined from and the number of outgoing edges from an activity; that is the SPLIT construct, (which allows to have several activities executing in parallel) is not a specific type of activities; if there are several outgoing nodes from a given activity, then it is a SPLIT construct.

The most usual patterns are summed up below, where the activity controlling the pattern is figured in blue, with the type of the activity shown beside.



The SplitAct activity allows to start two parallel activities. This is achieved by having simply two outgoing edges, one to P1Act activity, and one to P2Act Activity.

The SyncAct activity is of type AND-JOIN. It will be executed only where both P1Act and P2Act are in state terminated. If one of those activities is cancelled, then SyncAct will be cancelled.

The AsJoinAct is of type OR-JOIN. It will be executed whenever either P1Act or P2Act are terminated. If both of these activities are cancelled, then AsJoinAct will be cancelled as well.

The transition patterns can be refined by defining conditions on edges between activities. A condition operates on the value of a property of the activities, and is expressed in Java. Any string that can be the operand of an *if* statement is valid. Assuming that the property Prop is defined for a given activity, any of the following constructs is a valid condition:

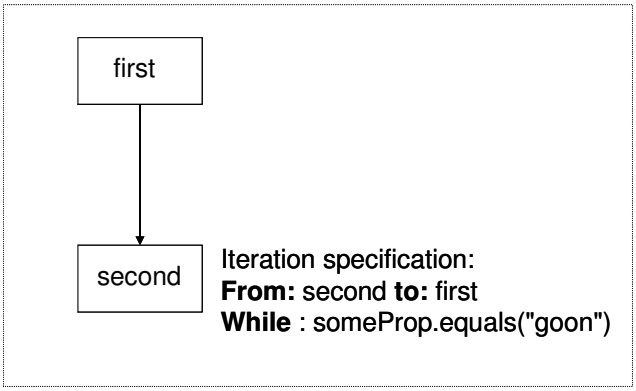
Prop.equals (“SomeString”)

(Prop.indexOf (“SomePart”) == 2)
(Prop.lenght() == 9)
orderType.equals("PO") && newInteger(Qte).intValue>100

1.3.3 Iterating activities

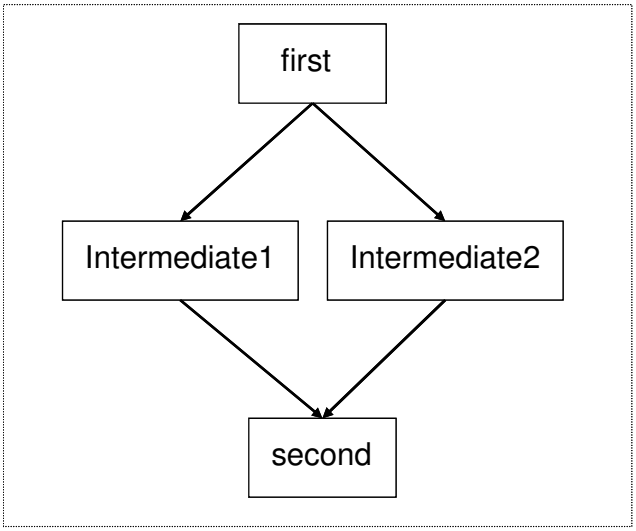
Bonita supports arbitrary cycles within a process. For this purpose, one attaches one iteration to the last activity of the cycle. This iteration bears the name of the first activity of the cycle, and the loop condition: while the condition evaluates to true, the Bonita execution engine will loop to the first activity while executing the termination algorithm for the last activity.

Therefore, here is how the simplest loop ever looks like :



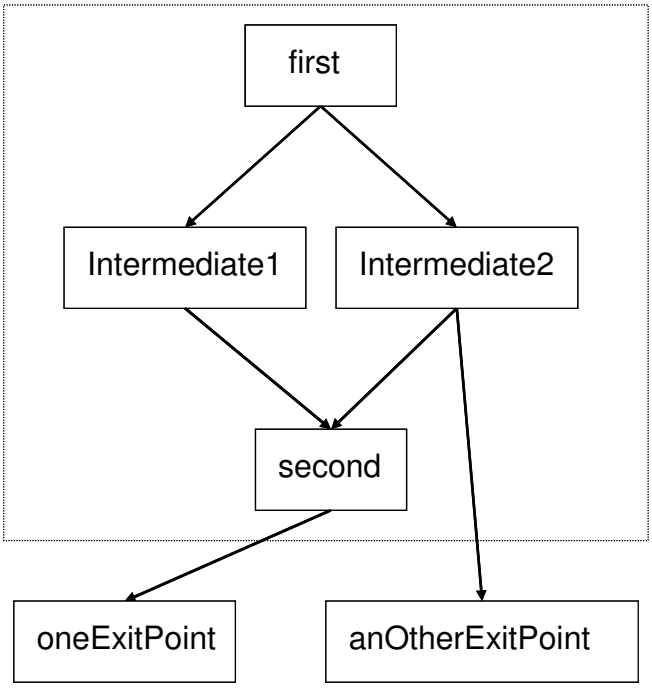
The condition is related to the value of the property someProp. This property is bound to the activity *second*, either directly (it is an activity property), or because it has been defined at the level of the process (it is project property)

The following example is a bit more complex :



Note that all the execution paths allowing to go from activity *first* to activity *second* will be included in the cycle, just like in the example beside, where *intermediate1* and *intermediate2* will be iterated several times.

It is possible to have several exit points from an iteration, like in the example below:

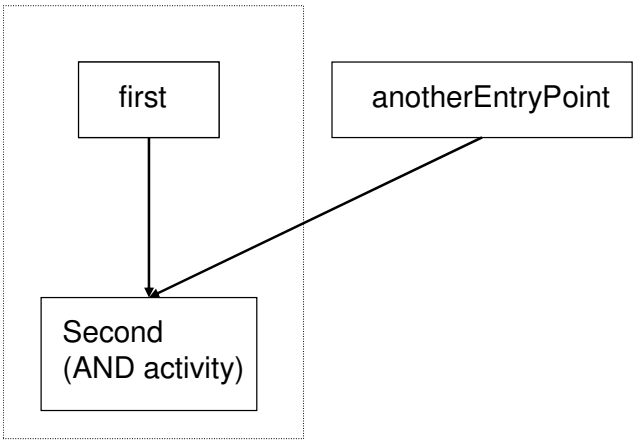


Assuming that an iteration has been set up between *second* and *first*, just like in the examples above:

When the iteration is entered, the outgoing transitions from *second* to *oneExitPoint* and from *Intermediate2* to *anOtherExitPoint* are both frozen, meaning they are not evaluated during the course of the iteration.

When the iteration condition set up on *second* evaluates to false, the iteration terminates, and all outgoing transitions are reevaluated. If the corresponding conditions are valid, then both *oneExitPoint* and *anOtherExitPoint* activities will be started.

It is possible to have several entry points into an iteration, like in the example below:



Assuming that an iteration has been declared between *second* and *first* just like in the example below:

Because *second* is an AND activity, it starts only when *anotherEntryPoint* has terminated. This is true only for the first occurrence of *second* : for the following iterations, the incoming transitions from *anotherEntryPoint* will be ignored.

1.3.4 Concept of Hooks

Hooks are user defined logic that can be triggered at some defined points in the life of the activity. Those defined points are :

- **Before Start hook** is called just before the activity starts. The Before Start hook is not considered to be in the same transaction than the activity. The Before Start hook is not triggered for automatic activities that cannot be anticipated.
- **After Start hook** is called just after the activity has started. It is considered to be in the same transaction than the activity. The After Start hook is not triggered for automatic activities that cannot be anticipated.
- **Cancel hook** is called before cancelling an activity and it's considered to be in the same transaction than the activity.
- **Before Terminate hook** is called just before the activity terminates. The Before Terminate hook is considered to be in the same transaction than the activity.
- **After Terminate hook** is called just after the activity has terminated. It is not considered to be in the same transaction than the activity.
- **Anticipating hook** is called when an automatic activity is started, only if the activity is anticipable. It is considered to be in the same transaction than the activity.

Fault management

If an exception occurs during the execution of a hook, it will be propagated to the application having triggered the execution of the hook.

Consider the following simple scenario:

- an application calls the **terminate Activity** statement on "Activity1"; this triggers the execution of a **before Terminate hook** which raises an exception; the exception will be caught by the application.

Things may be a little bit trickier if you use automatic activities :

- Imagine that the terminate Activity statement on "Activity 1" completes normally, and that "Activity 1" has an outgoing edge towards an automatic activity "Activity 2".
- "Activity 2" will be started and terminated automatically in the context of the first call related to "Activity 1".
- Therefore if "Activity 2" has a Before Terminate Activity hook that raises an exception, it will interrupt the call related to "Activity 1".
- That's means, "Activity1" does not terminate (the activity stay at executing state) and the system throws an exception due to "Activity2" execution error.

Examples above show you two error scenarios related to transactional hooks execution. **You should be aware that Hooks can be executed in a transactional or in a non-transactional context, depending on their types (before start, after start, …)**

Transactional hooks are executed in the same transactional context than the activity in which they are executed. Available transactional hooks in Bonita are: After Start, Before Terminate, Anticipate and On Cancel hooks (see also activities and transaction below)..

- Any changes performed on a transactional resource will be included in this existing transactional context.
- Any exception raised by the Hook will abort the existing transaction, so the activity will be re-executed later on. Furthermore, all operations executed by the hook before the exception was raised will be roll-backed.

Bonita also brings you the possibility to create hooks which can be executed without a transactional context. In this case, Before Start and After Terminate hooks are executed outside the activity transactional context.

- **We extremely recommend to do not use these types of hooks to access either Bonita APIs or other transactional APIs.**
- **If one of these hooks fails, during its execution the system will throw an exception but the activity starts/terminates without roll-backing any operation.**

Consider the last sample scenario described above and change Before Terminate hook by After Terminate hook. Let's over the execution:

- Imagine that the terminate Activity statement on "Activity 1" completes normally, and that "Activity 1" has an outgoing edge towards an automatic activity "Activity 2".
- "Activity 2" will be started and terminated automatically in the context of the first call related to "Activity 1".
- Therefore if "Activity 2" has an After Terminate Activity hook that raises an exception, the hook does not interrupt the call related to "Activity 1".
- That's means, "Activity1" terminates without problems, but the system throws an exception due to "Activity2" execution error.

1.3.5 Activity/hooks and transactions

Any change of state (that is startActivity, terminateActivity, cancelActivity statements) performed against an activity is part of a transaction.

Such a transaction will typically involve more than one activity: for example, a terminate Activity statement performed on a father activity will trigger a change of state in all daughter activities. BONITA therefore keeps transactional consistency across activities.

BONITA will abort the transaction in two cases:

- A failure at system level (e.g. impossibility to access the BONITA database)
- An exception not catched by a transactional Hook.

When Hook are executed in a transactional context :

- Any changes performed on a transactional resource will be included in this existing transactional context.
- Any exception raised by the Hook will abort the existing transaction.

1.3.6 Practical steps to use hooks

Hooks loading.

Hooks code can be stored in the Bonita database as *beanshell* programs. We call this kind of hook Interactive Hooks, or "InterHook" To use hooks in this fashion, you should just store programs in the Bonita database, either thru the graphical tool grapheditor (just right click on an activity, select add Hook, and use the editor to enter beanshell code), or thru the project API (see addInterHook, setInterHookValue, setNodeInterHookValue entry points).At execution time, the Bonita executive will take care of reading the code from the Bonita database, so you don't need to care about code loading.

Hooks code can also be stored on the file system as standard java classes. In that case, you need to load the code that you have written into the application server. The way to do this is as follows:

- Create your source .java file, say *MyHook.java*. It must be within the package *hero.hook*.
- Copy your java source file in the directory *$BONITA_HOME/src/resources/hooks/hero/hook*
- Goto *$BONITA_HOME/src* directory and type : *ant deployHook -DhookClass=<name of you java source file>*. In the example below, that's : *ant deployHook –DhookClass=MyHook*

Hooks Interface

All hooks must implement the hook interface. This interface is quite simple, with a single method having two parameters: an object EngineBean which is a session bean allowing to access the Bonita executive, and a BnNodeLocal object, which is a local interface to the entity bean representing the activity whose execution has triggered the execution of the Hook.

- We do not recommend making direct use of the EngineBean object.
- The BnNodeLocal object can be used to retrieve information about the currently executing activity.

1.4 User

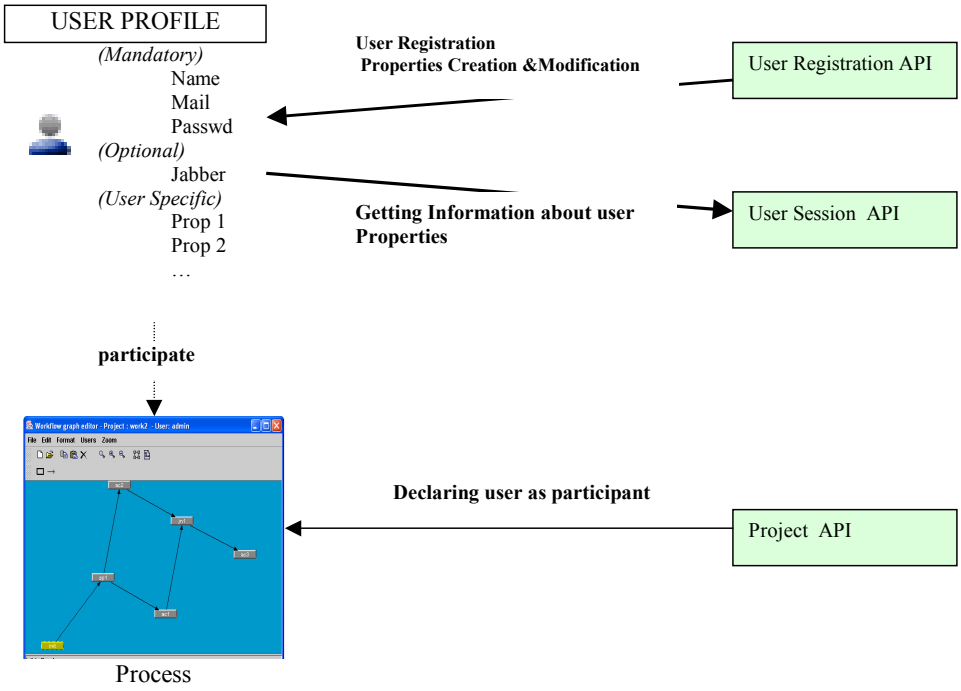
BONITA make the distinction between Users and Participants :

- **Users** are people who will make use of the workflow system (whatever process they will be part of).
- **Participants** are all the users that are allowed to play some role in a given process.

First, the **user** has to be registered in Bonita System, for authentication need (using **Bonita User Registration API**). Then, he has to be declared as **participant** in each project he is involved in (using **Bonita Project API**). He is then able to take part in the process.

Users are managed in BONITA specific base (or thru a LDAP repository as well). This base allows to store properties (also called preferences) for a given user. Properties are (key, value) pairs where both key and values are String variables. The application can set and retrieve

properties using the User interface. BONITA makes use of specific user properties in order to store the User preferences.



1.4.1 Relationship to processes

So, users have to be explicitly associated to processes in order to participate and to have visibility of events occurring in those processes.

Two scenarios allow to associate a User with a process (that is make a User a Participant of this process)

- Whenever a process is created, it is created on behalf of the User that initiated the Project Interface. This user is automatically associated to the newly created process, and can from now on assume the Admin role in the scope of the process.

- The users assuming the admin role for a given process has the right to associate new users to the process, and to allocate any role to them.

1.4.2 Authentication scenario

BONITA performs the User Authentication using either its specific database (mySql, Postgres, …) or a Ldap repository.. Here is an example of code running the authentication of the admin user. It uses the "TestClient" login context implemented in Bonita.

All other users have to be authenticated the same way.

Code sample:

```
import javax.security.auth.login.LoginContext;
import hero.client.test.SimpleCallbackHandler;


...

public class MyWorkFlowClass {

        static public void main(String[] args) throws Exception{
                // User Admin authentication
          char[] password={'t','o','t','o'};
          SimpleCallbackHandler handler = new SimpleCallbackHandler("admin",password);
          LoginContext lc = new LoginContext("TestClient", handler);
          lc.login();

          ...
}
```

1.5 Roles

1.5.1 J2EE Roles

The User Registration Interface, which allows to create users in Bonita database, is accessible without role restriction, that means everybody can call its methods, with no need to be authenticated.

Other Bonita Java Beans deal with the following J2EE roles: “Admin” and “users”. After authentication, only users having these J2EE roles will be able to access the Project and User Session Interface.

While being created with the User Registration Interface, a Bonita user is automatically associated first the “Admin” J2EE role. Everybody can then access the User Registration interface and create Bonita users.
Once created and after J2EE authentication, each Bonita user can access the Project Interface and create a new process, or clone or instantiate an existing process.

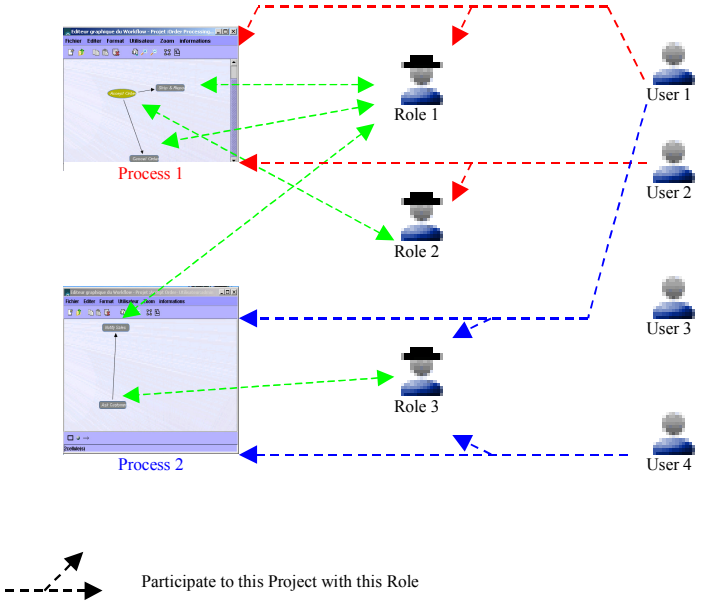
This J2EE security policy can be modified to enforce access control to Bonita Java beans methods, but in this case, be aware that Bonita beans source code has to be adapted to your own policy (especially if you modify role names). If you take this option you must have some headaches in order to migrate to the new Bonita versions.

In fact, we strongly recommend to let Bonita way of running as it stands, and to implement any user access restriction to Project or User Interface methods at an application level. See the *Application Access control* paragraph below for more details.

1.5.2 Bonita Roles

BONITA roles are related to activities access in processes. Each Process has its specific role management. This allows to have different semantics associated to the same role name in the scope of two different processes.

Activities are associated to roles, that is each activity can be taken over only by a user assuming a given role. There is a single role associated to each activity.
Users participate to a project, and in the scope of this project, a user can assume one or several roles.



User2 and User1 have to play Process1 alternatively; User1 can play every process2 activities due to its accreditation in the two roles involved in it.

Note:
Despite he has no role to play in any process, Bonita user3 could be able to clone, instantiate any of it but he don't able to modify it. He just has to know the name of a process to be able to call the Project interface methods to do this.
However:

- No Project or User session Interface methods will give him the name of an existing process, he is not involved in it,
- After instantiation, he won't be able to play any activity in it, because of the standard Bonita role access control.

Bonita Default roles :
Bonita handles two pre-existing roles : "admin" and "InitialRole". While created, an activity is automatically associated the "InitialRole". Then you have to modify it to suit your application functional requirements.
However, this InitialRole can be let for the first activity of the Workflow Process. This role can be then granted to one of the participant of the process, in charge of starting the workflow, independently of the other functional roles he can have in the process.

Additionally, this role could also be let for the automatic activities which do not need to be taken by users.

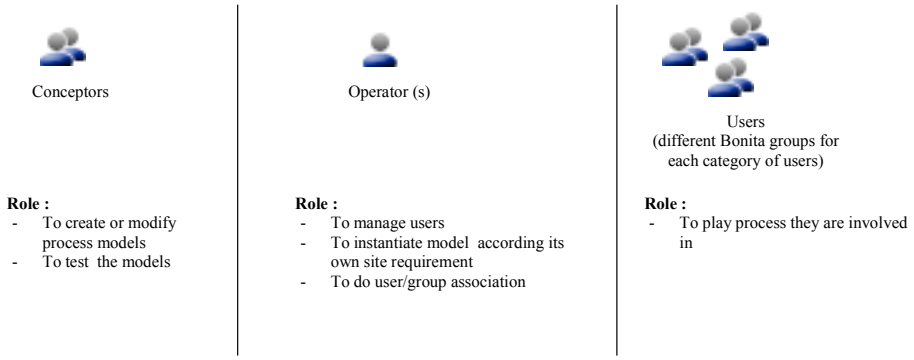
1.5.3 Application Access control

As mentioned above, standard Bonita access control is really open and allows you to adapt it to your organization needs.

The Bonita access control mechanism put in place a basic authentication scenario based on the workflow projects roles:

- User who has created the project becomes the admin of the project.
- Only this user can add other participants/users to this project.
- Only admin users can modify the project (set, add and delete entities).
- Users take part of the project are authorized to obtain some project information (get entities data).
- Project hooks and mappers could contain confidential information, so get data methods can only be used by admin users.
- Participants of the project can set/update properties of activities in which they are the corresponding role.

The Bonita Graph Editor application just follows these constraints: only the creator of a process and the users he has given the Bonita "admin" role can modify it. Even if he has a role to play in this process, another user can't add, delete or modify any node in it but he are able to visualize the evolution of it.
For example, a typical workflow application will distinguish three categories of users:



The application interface (graphical interface most of the time) will have to implement a kind of façade to restrain the users actions.

Applicative restrictions can concern stronger access control than Bonita does. But of course, you can't relax others Bonita standard access control based on the analysis of the points mentioned above:

- In this project, this node is associated to this role
- In this project, these users are participant
- In this project, this user can assume these roles

→ Can this user access this node ?

1.6 Mappers feature: automatic filling in of the bonita groups

1.6.1 Introduction

Mappers feature gives the possibility to fill in automatically the bonita roles defined into the project model when the project is instantiated.

Three filling in methods are available (3 types of mappers) depending on the way to retrieve the users in the information system

- by getting groups/roles in an LDAP server (*ldap mapper*)
- by calling a java class to request a database (*custum mapper*)
- by getting the initiator of the project instance (*properties mapper*)

Like others definitions of process elements, the access to this functionality is performed throw the bonita API (See the ProjectSessionBean API). It's also accessible within the *graphEditor* application.
This function is particularly interesting for process instantiation usage of Bonita workflow System. The fill in of the groups happens at the first instantiation of the project model (for both the project model and the 1st instance). Then, it happens at each instance creation.

1.6.2 LDAP, Custom and Properties Mappers

LDAP mapper:

This mapper uses your LDAP directory to retrieve users corresponding with a specific role defined in a Bonita Workflow project. Please refer to the documentation (Bonita LDAP configuration for JOnAS) to apply this type of mapper .

- LDAP mapper specificities:
  - The location of the LDAP groups depends on the attributes: *roleDN* and *roleNameAttribute* .
  - There is no mapping between roles/groups in the LDAP and roles in bonita database (same name for both bases).
  - The attribute name: *uid* has been used to realize the mapping between the actor identifier in the LDAP base and the userName in the bonita base.
  - If the goup does not exit an exception is thrown.
  - Users found in the groups must have been deployed before usage of the mapper function. Otherwise an exception is thrown.
  - The name of the mapper could be what you want

- Limitations of this version:
  - Groups cannot be recursive. Group's inclusions are ignored.
  - No checking that the distinguished names (dn) for the users found in the groups are compatible with the LDAP tree containing the users defined in the JOnAS LDAP realm configuration.

Custom mapper

It lets the process developer to request its own user's storage base. When this type of mapper has been added, a call to a java class is performed. The name of this mapper is the name of the called java class (ex.: *hero.mapper.CustomSeachGroup*) located under *BONITA_HOME\src\resources\mappers\hero\mapper*. After retrieving users these must be added to the project instance and also added to the targeted role. The Bonita workflow engine loads and executes these classes at runtime, so, if you would add your custom mapper, please follow the next steps:

- Take a look at sample class above and implements your custom mapper logic in a new java file.
- Create your source .java file, say *MyMapper.java*. It must be within the package *hero.mapper*.
- Copy your java source file in the directory *$BONITA_HOME/src/resources/mappers/hero/mapper*
- Goto *$BONITA_HOME/src* directory and type : *ant deployMapper -DmapperClass=<name of you java source file>*. In the example below, that's : *ant deploMapper –DmapperClass=MyMapper*

Properties mapper

At now, this type of mapper fills in the role with the user name of the creator of the instance (based on the authenticated user that initiates the instance). This mapper is very useful for administrative workflow processes in order to assign the role specified in the property to the user which has instantiated the process.

Example of Mapper code are available under *$BONITA_HOME/src/resources/mappers/hero/mapper.*

1.7 Performer Assignment

This feature propose to increase the possibility of Bonita by giving a mean to modify the standard assignment rules for activities

1.7.1 Introduction

This new feature allows getting additional assignment rules than in the standard bonita model. In the std model (oriented cooperative workflow), all the users defined into the group associated to the activity can see and can execute (***toDo List***) this one.
By adding the new functionality, we can:

- **assign the activity to a user of a group** by calling a java class in charge to do the user selection into the user group (*callback performer assignment*)
- **assign dynamically the activity to a user** by using an *activity property* (*properties performer assignment*)

When this functionality is added, the user is notified (mail notification) that the activity is ready to be started.
The users of the groups (role in Bonita) associated to the activity can see the activity but cannot start and terminate it.

This functionality is accessible within the Bonita API (see ***ProjectSessionBean API***) and inside the Bonita ***graphEditor*** application.
Furthermore, we can assign an activity to the initiator of the instance. It needs only the use of a properties mapper (as described above).

1.7.2 Description of these performer assignments

Callback performer assignment

It lets the process developer writing a request with its own algorithm of user selection. When this type of callback **performer assignment** has been added, a call to a java class is performed.

The name of this callback performer assignment is the name of the called java class (ex.: *hero.performerAssign.CallbackSelectActors*) located under *BONITA_HOME\src\resources\performerAssigns\hero\performerAssign*. As mappers, your callbacks are loaded and executed by Bonita workflow engine. If you would add your own callback, please follow the next steps:

- Take a look at sample class above and implements your performer assignment logic in a new java file.
- Create your source .java file, say *MyPerformer.java*. It must be within the package *hero.performer*.

- Copy your java source file in the directory *$BONITA_HOME/src/resources/performers/hero/performer*
- Goto *$BONITA_HOME/src* directory and type : *ant deployPerformer -DperformerClass=<name of you java source file>*. In the example below, that's : *ant deploPerformer –DperformerClass=MyPerformer*

Properties performer assignment

It allows the process developer to provide at the **properties performer assignment** creation the activity property that is used by the workflow engine to assign the activity. This activity property has to be defined either into a previously sequenced activity with the property propagation or into the targeted activity to be assign.

2 BONITA LDAP CONFIGURATION (FOR JONAS)

This chapter is an HowTo describing necessary steps for install and configures Bonita with your LDAP directory. After this operation, you will use your own LDAP in order to control user's access to Bonita Workflow System. This configuration is specifically for JOnAS Application Server 3.3.5 version.

For the moment, Bonita's LDAP module offers two functionalities:

- Bonita User Authorization Access via LDAP.
- Users import from your LDAP directory to Bonita user's database.

2.1.1 Installation

- The EJB implementing this function is compiled and deployed with others bonita files (under ejb/hero/session repository).
- Ldap module for JOnAS application server needs *javax77.jar* and *mejb.jar* files, so you have to invoke the ldap config task with the command: *ant configLdap*, in order to copy and deploy these files into JOnAS directory.

2.1.2 Configuration

- As importLdap bean uses JMX, an LDAP resource in the JOnAS-realm.xml file is required
- The configuration of this resource is detailed in the JONAS documentation (you can also use JonasAdmin graphical tool to add the new ldap realm; defaults are provided)
- Notice that you must have one and only one LDAP resources in JOnAS, otherwise the ldapImport method will fail.
- The ldap base could be also your authentication base for both JOnAS and bonita. For JOnAS, see the documentation. For bonita you have to set the ldap resourceName for bonita context in $JONAS_ROOT/conf/server.xml and in $JONAS_ROOT/conf/jaas.config for bonita entries.
- Configuration example:

```
<jonas-ldaprealm>
  <ldaprealm name="ldaprlm_1"
        baseDN="ou=fr,o=Bull, c=fr"
        initialContextFactory="com.sun.jndi.ldap.LdapCtxFactory"
        providerUrl="ldap://serveur_host:389"
        securityAuthentication="simple"
        authenticationMode="bind"
        userPasswordAttribute="userPassword"
        userRolesAttribute="memberOf"
        roleNameAttribute="cn"
        userDN="ou=frec_users,ou=fr,o=Bull, c=fr"
        userSearchFilter="uid={0}"
        roleDN="ou=frec_groups,ou=fr,o=Bull, c=fr"
        roleSearchFilter="uniqueMember={0}"
        referral="follow" />
</jonas-ldaprealm>
```

2.1.3 What do this ldap Import?

This function is intended to search users under the *userDN* subtree with the *userSearchFilter* filter and then for each found user get the attribute value specified into the userSearchFilter and also get the mail attribute value (assuming that the attribute name for email in the ldap is: *mail*, otherwise change it in the bean code). No mapping attribute name for email has been yet introduced.

If the user doesn't exists in the bonita base, it is created by calling the userCreate() API with parameters: name, password (filled in with the name value) , mail. If it already exists, the email is updated (the user identifier in bonita db is the column-name: name). Remember also that all users in bonita db that no more exist in the ldap subtree (and only if the user is not involved in workflow projects) are removed. Previous operation except the default users of Bonita Workflow System: “admin”, “admin2”, “nobody”.

2.1.4 How you can use it?

- This function can be integrated into your administration module. See example provided under *src\main\client\hero\client\importLdap* to call the bean method.
- You can also simply invoke the ldap import with the command: *ant importLdap* <uid> <password>

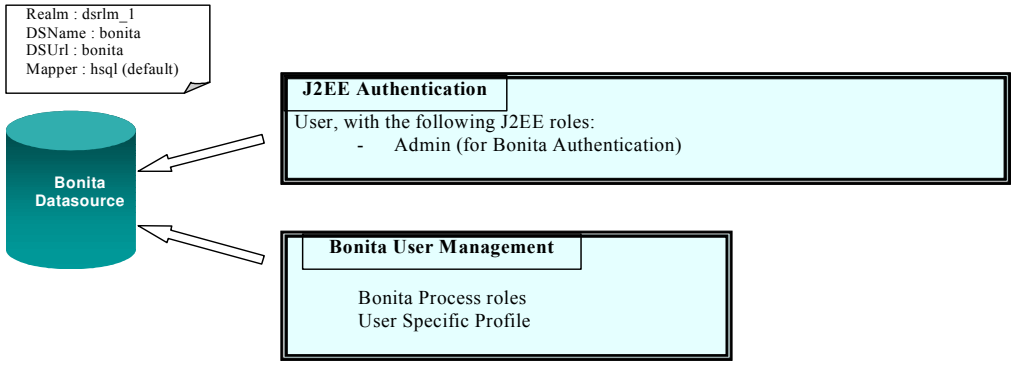
The user with the uid and password provided to the importLdap ant task has to be previously declared into the directory under the *userDN* subtree and has also to be member of the group Admin (under the *RoleDN* subtree)

Note: if the ldap server acts as your authentication base while the Bonita base is your user base, you have also to declare the user accessing the bonita admin console as member of the Admin group (under the *RoleDN* subtree)

3 USER MANAGEMENT

3.1.1 Bonita User Management basic configuration

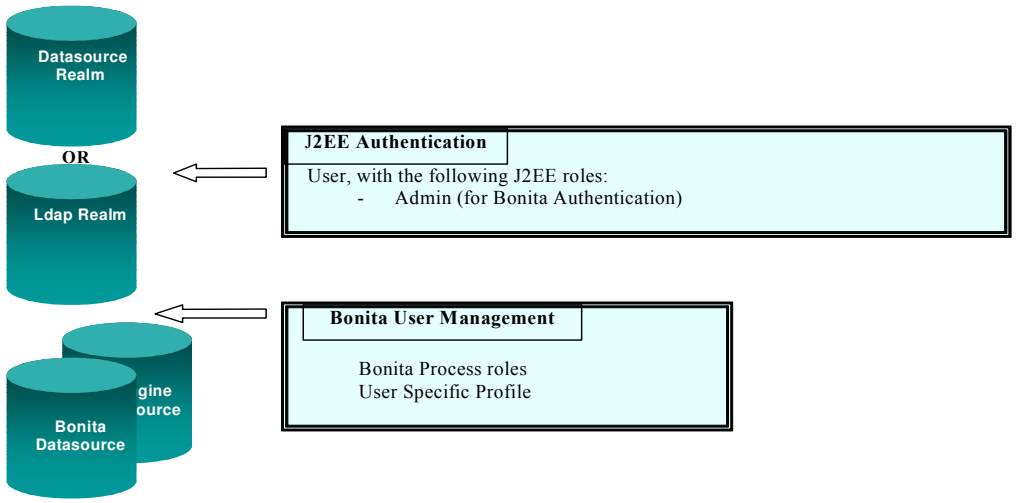
After Bonita installation and configuration, user specific data will be stored in the Bonita database you choose during the configuration phase. That is, basically, some tables created in the Bonita database allowing security control and user's management as it is shown below.



This basic configuration can be changed according to your preferences, for example to use your existing user database or your enterprise Ldap Directory.

3.1.2 How to change the basic configuration

User Management may move to the following schema, to make your application fully integrate your enterprise Information System. Bonita will then take advantage of User Management defined at an upper level than the only need of your workflow application.



3.1.2.1 J2EE Authentication

Bonita uses the security realm defined at the global context for Jonas (jonas-realm.xml file in $JONAS_BASE/conf directory). To change the basic configuration you will then have to:

In case of use of another Datasource Security Realm:

- Modify the existing datasource (called dsrlm_1) by your own user and roles queries.

In case of use of an Ldap Security Realm

- Uncomment the <jonas-ldaprealm> sample file and configured it. Take a look at http://jonas.objectweb.org/current/doc/Config.html#Config-Security (look for *Configure LDAP resource in the jonas-realm.xml file*)

3.1.2.2 Bonita User Management

By default, Bonita uses *hero.user.DefaultUserBase* user's implementation class to manage users. To add your own user's management class you will have to:

- Implements the *hero.user.UserBase* interface providing users information required to deal with your own user's management system (database, LDAP directory, User Interface…). This class must be under the *hero.user* package.

- Copy your java source file into *$BONITA_HOME/src/resources/users/hero/user* directory.

- Goto *$BONITA_HOME/src* directory and type: *ant deployUserBase -DhookClass=<name of you java source file>*. For instance: *ant deployUser –DuserClass=MyUserClass*

- Update the value of the *user.base* attribute with your class name implementation ($BONITA_HOME/.ant.properties file).

4 USER REGISTRATION INTERFACE

4.1 Principle

The User Registration interface provides access to J2EE users and roles definition.

In case of EJB Session access, the User Registration interface will automatically retrieve the identity of the calling user in the J2EE security context. Hence, calling the User Registration interface from an unidentified context will fail.

Then, without modifying the Bonita sources, only users with “Admin” or “users” J2EE roles will be able to access Project and User Session Interfaces.

*** Important Note: UserRegistration API should only be used in the case of your User Management configuration is the Bonita default configuration !! If you are using your own User Management implementation just ignore this API.**

4.2 Creating the UserRegistrationBean

The UserRegistrationBean can be seen as a handle to add new user or role in the J2EE Application Server security context. You first have to create the handle, and the call the UserRegistration interface methods.
This API is a stateless session bean.

Code sample :

```
import javax.security.auth.login.LoginContext;
import hero.client.test.SimpleCallbackHandler;

import hero.interfaces.ProjectSession;
import hero.interfaces.ProjectSessionHome;
import hero.interfaces.ProjectSessionUtil;

public class MyWorkFlowClass {

        static public void main(String[] args) throws Exception{
                // User Admin authentication
          char[] password={'t','o','t','o'};
          SimpleCallbackHandler handler = new SimpleCallbackHandler("admin",password);
          LoginContext lc = new LoginContext("TestClient", handler);
          lc.login();


          // User Registration Bean Creation using Remote Interface
          UserRegistrationHome userRHome= (UserRegistrationHome) UserRegistrationUtil.getHome();
          UserRegistration urSession = userRHome.create();

          ...
}
```

4.3 Managing Users

4.3.1 Creating Users

```
void userCreate(String name, String password, String email)
```

Creates user account. The user is automatically associated the “Admin” group.

`void` **`userCreate`**`(String name, String password, String email, String jabber)`

Creates user account with an instant messaging address. The user is automatically associated the "Admin" group.

4.3.2 Defining Users

`void` **`setUserProperty`**`(String userName, String key, String value)`

Set a new property to the user. User properties will be used to define user preferences.

`void` **`setUserRole`**`(String userName, String roleName)`

Set a new authorization role to the user.

4.3.3 Deleting Users

`void` **`deleteUser`**`(String userName)`

Detete a user from Bonita database. If the user is included in active projects this methods throws an exception.

4.4 Roles

4.4.1 Creating Roles

`void` **`roleCreate`**`(String name, String roleGroup)`

Creates a new authorization role to the system. This kind of role is used to control the user access to different APIs. **Remember that the User Registration API deals with J2EE identities.** These roles must not be mistaken with Bonita roles associated with projects.

This function can be useful to change the defaults roles of Bonita and therefore control more precisely the access rights.

4.5 Code sample

```
import javax.security.auth.login.LoginContext;
import hero.client.test.SimpleCallbackHandler;

import hero.interfaces.ProjectSession;
import hero.interfaces.ProjectSessionHome;
import hero.interfaces.ProjectSessionUtil;

public class MyWorkFlowClass {

        static public void main(String[] args) throws Exception{
                // User Admin authentication
          char[] password={'t','o','t','o'};
          SimpleCallbackHandler handler = new SimpleCallbackHandler("admin",password);
          LoginContext lc = new LoginContext("TestClient", handler);
          lc.login();


          // User Registration Bean Creation using Remote Interface
          UserRegistrationHome userRHome= (UserRegistrationHome) UserRegistrationUtil.getHome();
          UserRegistration urSession = userRHome.create();

          // User "jack" (customer) creation in Bonita database
          try{
            userReg.userCreate("jack","jack","miguel.valdes-faura@ext.bull.net");
          }catch(Exception e){System.out.println(e) ;} // Maybe user exists


          // User "john" (service customer) creation in Bonita database
          try{
            userReg.userCreate("john","john","miguel.valdes-faura@ext.bull.net");
          }catch(Exception e){System.out.println(e) ;} // Maybe user exists

          userReg.remove();
}
```

5 PROJECT INTERFACE

5.1 Principle

The Project interface provides access to functions allowing to modify the execution of a given process.

In case of EJB Session access, the Project interface will automatically retrieve the identity of the calling user in the J2EE security context. Hence, calling the Project interface from an unidentified context will fail. Therefore, the interface is initiated for a given user. Only the processes where the User are declared can be accessed

Once the Project interface has been created, it must be initiated. Initiating the Project interface allows to specify which project is going to be managed thru the Interface.

You will find below examples of code using this interface.
These are extracts of the SampleProjectApi.java example proposed in the “samples” directory of Bonita. You can run this example by using then ant tasks “sample-project-api” .
You can also refer to the sample1xxx classes, which implement the user guide workflow example (Order Processing and Customer Service).
You can run them by using then ant tasks “sample1-create-process-model” (Model creation), “sample1-admin-wf” (user administration and project instantiation), “sample1-running-session” (Process execution).

5.2 Creating the ProjectSessionBean

Think about the ProjetSessionBean as an handle to your connexion with the BONITA workflow System. You first have to create the handle, then to associate a given project to this handle in order to be able to modify it.

Code sample :

```
import javax.security.auth.login.LoginContext;
import hero.client.test.SimpleCallbackHandler;

import hero.interfaces.ProjectSessionHome;
import hero.interfaces.ProjectSession;
import hero.interfaces.ProjectSessionUtil;



import hero.interfaces.Constants;

import java.util.*;

public class SampleProjectApi {

          static public void main(String[] args) throws Exception{

        // User Admin login
        char[] password={'t','o','t','o'};
        SimpleCallbackHandler handler = new SimpleCallbackHandler("admin",password);
        LoginContext lc = new LoginContext("TestClient", handler);
        lc.login();

       // Project Session Bean Creation using Remote Interface
        ProjectSessionHome prjHome= (ProjectSessionHome) ProjectSessionUtil.getHome();
        ProjectSession prjSession = prjHome.create();
```

5.3 Initiating the ProjectSessionBean

5.3.1 Initiating the Session Bean (Cooperative projects & instances)

```
Void  initProject (String projectName)
```

Creates or initializes cooperatives workflow projects. This method could also be used to initializes workflow instances.

The Project interface is **initialized** with the given projectName.
If the corresponding *projectName* does not exist.
A new empty project is then created and given this name.
The user is then associated the Bonita "admin" role for this project.
There are no restrictions on the number of characters used to name process.

5.3.2 Initiating the Session Bean (Models)

```
Void  initModel (String modelName)
```

Creates or initializes workflow models.

The Project interface is **initialized** with the given modelName.
If the corresponding *modelName* does not exist.
A new empty model is then created and given this name.
The user is then associated the Bonita "admin" role for this project.
There are no restrictions on the number of characters used to name process.

5.3.3 Initiating with the clone project creation option

```
Void  initProject (String oldProject, String newProject)
```

The Project interface is initialised after oldProject was cloned. This interface is initialised with the given *newProject* project name.

After using the initProject method, all subsequent interface methods further called will then deal with the corresponding project.

5.3.4 Initiating with the instantiate project creation option

```
Void  instantiateProject (String modelName)
```

The Project interface is initialised after new project instance was created.
This interface is initialised with the new project instance name, automatically given by Bonita which derives the instance name from the model name as follows:

<instance-name> = <model-name>_instance<sequence-number>

All subsequent interface methods will deal with the corresponding project instance.
After this instantiation, users have to be added to the new instance if they were not defined in the process model. Also, they must be associated roles to start/stop activities in this new project.

Only workflow models can be instantiated. Cooperative projects are ready-to-define, ready-to-executed after creation.

5.3.5 Code sample

```
//*************************************************************/
//********* API Documentation - Sample 1 (adapted version) *************/
//*************************************************************/

//Process creation by user admin
prjSession.initProject("Original Process");
    // if "Original Process" does not exists, it is created.
    // Process definition  see following sections
    // adding activities, edges, ...
    //

//Process "Original Process" Cloning into "Clone Process"
try {
    prjSession.initModel("Original Process", "Clone Process");
} catch(Exception e) {System.out.println(e);} //Maybe project does not exists

// "Original Process" instantiation
try {
    prjSession.instantiateProject("Original Process");
} catch(Exception e) {System.out.println(e);} //Maybe project does not exists
// The new instance becomes the current project
```

5.4 Managing project

With BONITA, there is a single API to cope with projects. This API is used to control processes, no matter which kind of process they are:

- Processes can exist on their own, without having a relationship to a process model. In this category we find processes created from scratch, and processes cloned from parent processes.
- Process can be process model, from which process instances can be derived. At the moment, a process model can be executed as well, but this behaviour will be withdrawn in a near future.
- Process instances are specific runable processes whose definition are contained in a process model. At creation time, the specific context of this instance is taken into account in order to specialize the instance.

5.4.1 Project attributes

A project has a name, which is given at creation time thru the Project API.

Only the name of process instances is constrained, where BONITA automatically allocates a name in the following form : <*Project Model Name*>_instance<*Project Instance Number*>.
The <*Project Instance Number*> is automatically managed by BONITA.

A project has properties, which are simple (key, value) pairs.

A project records the name of the person which created it and the creation date.

The values associated with processes states constants are:

CONSTANT	VALUE
hero.interfaces.Constants.Pj.INITAL	0
hero.interfaces.Constants.Pj.STARTED	1
hero.interfaces.Constants.Pj.TERMINATED	2

The values associated with processes types constants are:

CONSTANT	VALUE
hero.interfaces.Constants.Pj.COOPERATIVE	Cooperative
hero.interfaces.Constants.Pj.MODEL	Model
hero.interfaces.Constants.Pj.INSTANCE	Instance

The values associated with processes status constants are:

CONSTANT	VALUE
hero.interfaces.Constants.Pj.ACTIVE	Active
hero.interfaces.Constants.Pj.HIDDEN	Hidden

5.4.2 Active/Hide a workflow process

public void **activeProcess**()

Set the process status to Active (model/cooperative/instance).
Workflow processes can only be executed and modified if they are active

public void **hideProcess**()

Set the process status to Hidden (model/cooperative/instance).
Hide this workflow process. Only get methods operations will still allowed.

5.4.3 Getting the name of a project or an instance

```
public String  getName();
```

Returns the name of the project that is being managed by the current instance of ProjectSessionBean interface.

```
public String   getProjectNameOfInstance(String instanceName)
```

Returns the project name of the instance instanceName.

5.4.4 Getting the name of the parent project

```
public String getParent()
```

If the current project is a subProcess, returns the name of its parent project.

5.4.5 Getting the name of a project's creator

```
String  getCreator();
```

Returns the name of the user who has been creating the Project. The creator name is automatically retrieved by BONITA executive when one creates a project thru the ProjectSessionBean Interface.

5.4.6 Properties

`void` **`setProperty`** `(String key, String value)`

Creates a new property , assigning it a value, or override the value of an existing property.

`Collection` **`getProperties`**`()` *`(BnProjectPropertyValue Collection)`*

Returns all the properties existing for this project.

`Collection` **`getPropertiesKey`**`()` *`(String Collection)`*

Get properties key of the project. A property is a pair key/value representing workflow relevant data.

`BnProjectPropertyValue` **`getProperty`**`(String key)`

Returns the property value of the project. A property is a pair key/value properties associated to this project.

`void` **`deleteProperty`**`(String key);`

Delete a property of an existing project.

5.4.7 Project details

`public BnProjectValue` **`getDetails`**`()`

Returns project information: project attributes, nodes, edges, hooks, properties...

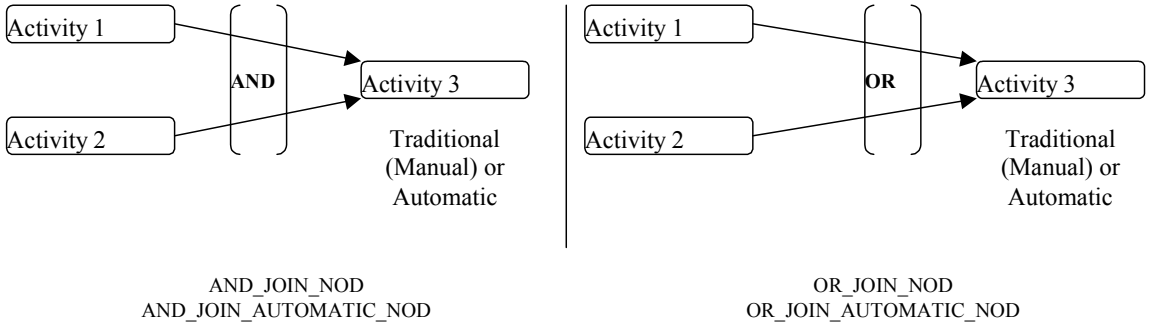
5.4.8 Code sample

```
 /*****************************************************************/
 /************** API Documentation - Sample 2 ******************/
 /*****************************************************************/

String processName = prjSession.getName() ;
System.out.println("Current Process : " + processName) ;

try {
      String parentName = prjSession.getParent();
       System.out.println("Parent Process : " + parentName) ;
 } catch(Exception e) {System.out.println(e);} //Maybe there is no parent


try {
      String creatorName = prjSession.getCreator();
       System.out.println("Process Creator : " + creatorName) ;
 } catch(Exception e) {System.out.println(e);} //Maybe there is a problem

try {
       prjSession.setProperty("userId","user1");
       prjSession.setProperty("recordId","1111");
       prjSession.setProperty("orderId","0001");
 } catch(Exception e) {System.out.println(e);} //Maybe there is a problem


 // First way to get properties values
 System.out.println("First way to access proprerty values : ");

 Collection properties = prjSession.getProperties() ;
 Iterator i = properties.iterator();
 while (i.hasNext())
 {
      hero.interfaces.BnProjectPropertyValue property = (hero.interfaces.BnProjectPropertyValue)i.next();
      try {
            String propertyKeyName = property.getTheKey();
            String propertyValue = (String)property.getTheValue();
            System.out.println("Property (Key, Value) : " + propertyKeyName + "/" + propertyValue);
      } catch(Exception e) {System.out.println(e);} //Maybe there is a problem
 }


 // Second way to get properties values
 System.out.println("Second  way to access proprerty values : ");
 properties = prjSession.getPropertiesKey() ;
 i = properties.iterator();
 while (i.hasNext())
 {
      String propertyKey = (String)i.next();
      try {
            hero.interfaces.BnProjectPropertyValue propertyValue = prjSession.getProperty(propertyKey);
            System.out.println("Property (Key, Value) : " + i + "/" + propertyValue);
      } catch(Exception e) {System.out.println(e);} //Maybe there is a problem
 }

//Deleting Property
try {
   prjSession.deleteProperty("orderId");
} catch(Exception e) {System.out.println(e);} //Maybe there is a problem

//Verification

 System.out.println("Properties after one deletion : ");
 Collection propertiesLeft = prjSession.getPropertiesKey() ;
 Iterator j = properties.iterator();
 while (j.hasNext())
 {
      String propertyLeftKey = (String)j.next();
      try {
            hero.interfaces.BnProjectPropertyValue propertyValue = prjSession.getProperty(propertyLeftKey);
            System.out.println("Property (Key, Value) : " + i + "/" + propertyValue);

      } catch(Exception e) {System.out.println(e);} //Maybe there is a problem
 }
```

5.5 Defining and Getting Informations about activities

5.5.1 Types of activities

Activity type can be either one of the following :



AND_JOIN_NOD
AND_JOIN_AUTOMATIC_NOD

OR_JOIN_NOD
OR_JOIN_AUTOMATIC_NOD

or SUB_PROCESS_NODE : this node is itself a complete process included in the current process as a sub-process.

Here are the values associated with the types constants :

CONSTANT	VALUE
hero.interfaces.Constants.Nd.AND_JOIN_	1
hero.interfaces.Constants.OR_JOIN_NODE	2
hero.interfaces.Constants.AND_JOIN_AUTOMATIC_NODE	3
hero.interfaces.Constants.OR_JOIN_AUTOMATIC_NODE	4
hero.interfaces.Constants.SUB_PROCESS_NODE	5

5.5.2 Activities states

See the "Activities basics " section of this document.

The values associated with the main activities states constants are :

CONSTANT	VALUE
hero.interfaces.Constants.Nd.INITIAL	0
hero.interfaces.Constants.Nd.READY	1
hero.interfaces.Constants.Nd.DEAD	2
hero.interfaces.Constants.Nd.ANTICIPABLE	3
hero.interfaces.Constants.Nd.ANTICIPATING	5
hero.interfaces.Constants.Nd.EXECUTING	6
hero.interfaces.Constants.Nd.TERMINATED	10

The following values can also be returned by the Bonita Engine :

CONSTANT	EXPLANATION	VALUE
EXPIRED	Deadline expired	4
EXECUTED		7
INERROR		8
FINISHED		9
CHECKEDOUT		11
ANT_SUSPENDED		12
EXEC_SUSPENDED		13
BAD_TRANSITION		14

5.5.3 Creating activity

```
void addNode(String name, int nodeType)
```

Add a node to the project. This method creates a node with the corresponding node type and assign it **InitialRole** role. This role is not assigned to any user at creation time, so this activity isn't eligible without using the setNodeRole method.

5.5.4 Creating SubProcess activity

```
void addNodeSubProcess(String name, String projectName)
```

Add a subProcess node to the project. This method creates the subProject from an existing project and creates the node associated to it. The type of created node is hero.interfaces.Constants.Nd.SUB_PROCESS_NODE

5.5.5 Configuring activity

```
void setEditNode(String node,String role, String description,
long deadline)
```

Set information on node changes (including role, description, deadline). This is especially useful for graphical client application

```
void setNodeAnticipable(String name)
```

Set the node in anticipable mode.

```
void setNodeAutomatic(String name)
```

Set the node in automatic mode. The responsible of the activity execution is now the engine.

```
void setNodeDeadline(String name, long date)
```

Set the node deadline. Activity deadline is the lastest date in which the activity must be finished.

```
void setNodeDescription(String name,String description)
```

Set the node description. Node description represents, explicity, execution related information of this task.

```
void setNodeProperty (String nodeName, String key, String value)
```

Set a property of a node. A property is a pair key/value representing workflow relevant data. With this method the property is propagated within others nodes.

```
void setNodeProperty(String nodeName, String key, String value,
boolean propagate)
```

Set a property of a node. A property is a pair key/value representing workflow relevant data. By using propagate argument we can specify if we want to propagate this property.

```
void setNodeTraditional(String name)
```

Set the node in traditional mode. When a node is traditional the anticipable attribute is false. This method must be used if you want to execute this activity in a traditional model.

```
void setNodeType(String name, int type)
```

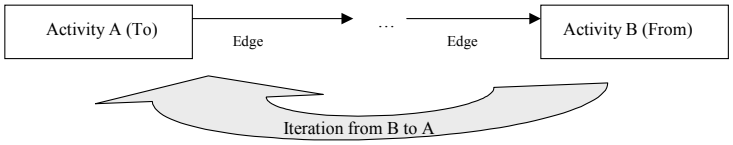
Set the node type. Change the current type of the node (if node is not executing).

5.5.6 iterating activities

```
void addIteration(String from, String to, String condition)
```

Add a new iteration between two nodes. This methods sets an iteration which is stopped when the condition is false. “from” means the name of the first node, “to”, the name of the last node.

Note that the iteration must be added on the node which executes the last; in the example below, activity A is executed, then some activities between A and B take place, and then B is executed. After the processing of B, control will go back to A if the iteration condition that has been set on B evaluates to true.



The condition can be something like “lastNodeProperty.equals(\”value\”)”, while the value of the property is positioned depending on the execution of the process.

Code sample : To see activity iteration, refer also to **Sample1CreateProcessModel** example in the Bonita samples directory. **Sample1RunningSession** will play the fisrt iteration of the process. To terminate it after the second iteration, you just have to modify the *once_more* value of the *Receive Order* activity in the *Order Processing* Instance you are running.

5.5.7 Getting information about nodes in the project

```
Object getNodes()
```

Returns project nodes data as an array of StrutsNodeValue. This is especially useful for Struts based IHM, but can be used also in any kind of application.

```
Collection getNodesNames()     (String Collection)
```

Returns all node names of the project .

5.5.8 Getting information about a specific node

`BnNodeValue` **`getNode`**`(String projectName,String nodeName)`

Get Node Value from a specific project

`String` **`getNodeDeadline`**`(String nodeName)`

Returns node deadline. Activity deadline is the lastest date in which the activity must be finished.

`String` **`getNodeDescription`**`(String name)`

Returns the node description. Node description represents, explicity, execution related information of this task.

`String` **`getNodeExecutor`**`(String name)`

Returns the node executor. Get the name of the user which is executing the activity.

`Collection` **`getNodeProperties`**`(String nodeName)` *`(BnNodePropertyValue Collection)`*

Returns Node properties as a list of pair key/value properties associated to the node.

`BnNodePropertyValue` **`getNodeProperty`**`(String nodeName, String key)`

Returns Node property value. Get a pair key/value properties associated to the node.

`BnNodePropertyValue` **`getNodeProperty`**`(String nodeName, String key)`

Returns Node property value. Get a pair key/value properties associated to the node.

`int` **`getNodeState`**`(String name)`

Returns the state of the node.

`int` **`getNodeType`**`(String name)`

Returns the type of the node.

`BnNodeValue` **`getNodeValue`**`(String name)`

Returns the node Value. Returns node information.

`boolean` **`getNodeAnticipable`**`(String name)`

Returns if the node is set to be executed in anticipated mode.

5.5.9 Deleting activity

`void` **`deleteNode`**`(String name)`

Delete a node from the project. If this node is in executing, terminated or cancelled state, the method throws an execption

`void` **`deleteNodeProperty`**`(String nodeName, String key)`

Delete a property of a node. Deletes the node property associated to this key

5.6 Managing Edges

5.6.1 Adding an edge to an activity

An edge is a way to establish a dependency between two activities.

Edges have unique name in the scope of the project. The name of the edge can be choosen by the application, or it can be automatically generated by BONITA.

```
String      addEdge(String in, String out);
```
The two activities named in and out will be connected by a new eddge. The method returns the name of the newly created edge.

```
String      addEdge(String name,String in, String out);
```
The two activities named in and out will be connected by a new edge. The newly created edge will be named according to the name passed as input parameter.

5.6.2 Deleting an edge

```
Void  deleteEdge(String name);
```
The edge named with the parameter name will be deleted.

5.6.3 Getting connected activities from an edge

```
String      getEdgeInNode(String edgeName) ;
```
Get back the name of the inbound node of the given edgeName.

```
String      getEdgeOutNode(String edgeName) ;
```
Get back the name of the outbound node of the given edgeName.

5.6.4 Setting a condition on an edge

```
Void  setEdgeCondition(String edge, String condition);
```

A condition operates on the value of a property of the activities, and is expressed in Java. Any string that can be the operand of an *if* statement is valid. Assuming that the property Prop is defined for a given activity, any of the following constructs is a valid condition:

Condition = “Prop.equals (\“SomeString\”)
Condition = “(Prop.indexOf (\“SomePart\”) == 2)”
Condition = “(Prop.lenght() == 9)”

5.6.5 Getting the condition on an edge

```
String      getEdgeCondition(String edge);
```

5.6.6 Getting all existing edges in a project

```
Collection  getEdgesNames()   (String Collection)
```
Returns all the existing edges in the project.

5.6.7 Getting all existing edges for an activity

```
Collection  getNodeInEdges()  (String Collection)
```
Returns all the existing edges inbound for a given node.

```
Collection  getNodeOutEdges() (String Collection)
```
Returns all the existing edges outbound for a given node.

5.6.8 Reading an edge as a Java Object

```
hero.interfaces.BnEdgeValue  getEdgeValue (String name);
```
Get the edge value.

5.6.9 Changing the state of an Edge

```
void  setEdgeState(hero.interfaces.BnEdgeLocal edge, int state);
```
Set the edge state

5.7 Managing Hooks

Hooks are piece of code that is executed at specific point during the activity life cycle.

Must document in a central place the different possible scripting strategies

Hooks can be coded in a scripting language, or as java library. Therefore, the hook interface is divided in two sets (Hooks and InterHooks).
Hooks can be defined at the project level. Such hooks will be activated for every activity contained in the project.

Hooks can also defined at the activity level, they will be activated only in the context of the related activity.

Interactive Hooks :
Script hooks are called interactive Hooks, hence all calls related to them will contain “Inter” in their name. Their type is hero.hook.Hook.BSINTERACTIVE

Hooks are executed on reception of one of the following events, and the appropriate method of the hook will be executed. If the hook does not include such a method, an exception is raised.

EVENT	VALUE	METHOD

hero.interfaces.Constants.Nd.BEFORESTART	"beforeStart"	beforeStart
hero.interfaces.Constants.Nd.AFTERSTART	"afterStart"	afterStart
hero.interfaces.Constants.Nd.BEFORETERMINATE	"beforeTerminate";	beforeTerminate
hero.interfaces.Constants.Nd.AFTERTERMINATE	"afterTerminate";	afterTerminate
hero.interfaces.Constants.Nd.ONCANCEL	"onCancel"	onCancel
hero.interfaces.Constants.Nd.ANTICIPATE	"anticipate";	anticipate

Different hooks types actually taken in count by the Bonita engine :

HOOK TYPE	VALUE
hero.interfaces.Constants.Nd.TCL	**4**
hero.interfaces.Constants.Nd.BEANSHELL	5
hero.interfaces.Constants.Nd.BSINTERACTIVE	6

5.7.1 Hook at the project level

Hooks associated at the project level are associated to all nodes of the project.

Creating

`Void` **`addHook`** `(String hookName, String eventName, int hookType)`

Add an existing hook file to the project. This hook type uses a Java or TCL file loaded at run time. The hookName represents the class java or tcl file to be loaded by the system at run time. These classes must be in the application server classpath to be correctly executed. Put your hooks classes in $BONITA_HOME\src\resources\hooks and then redeploy bonita.ear (ant task).

`Void` **`addInterHook`** `(String hookName, String eventName, int hookType, String value)`

The hook with name hookName will be added to a project. The hook activation will be triggered whenever the event eventName occurs in any activity of the project.

Deleting

`Void` **`deleteHook`** `(String hookName)`

`Void` **`deleteInterHook`** `(String hookName)`

The hook or interHook with name hookName will be deleted from all projects nodes.

Managing

`void` **`setInterHookValue`**`(String hook, String value)`

Set the value of the inter hook named « hook ». The value parameter contains the new inter hook script which will be associated to all project nodes

`void` **`setNodeInterHookValue`**`(String node, String hook, String value)`

Set the value of the inter hook named « hook » associated to the node named « node ». The value parameter contains the new inter hook script which will be associated to all project nodes

`Collection` **`getHooks`**`()` *`(ProjectHooksValue Collection)`*

Returns all the hooks of the project**.**

`Collection` **`getInterHooks`**`()` *`(ProjectInterHookValue Collection)`*

Returns the interactive hooks of the project.

`String` **`getInterHookValue`**`(String hook)`

Returns the inter hook value script.

5.7.2 Hooks associated to a specific activity

Creating

`Void` **`addNodeHook`** `(String nodeName, String hookName, String eventName, int hookType)`

Add hook to a node. Add an existing hook file to the node. This type of hook use a Java or TCL file loaded at run time. The hookName represents the class java or tcl file to be loaded by the system at run time. These classes must be in the application server classpath to be correctly executed. Put your hooks classes in $BONITA_HOME\src\resources\hooks and then redeploy bonita.ear (ant task).

`Void` **`addNodeInterHook`** `(String nodeName, String hookName, String eventName, int hookType, String value)`

The hook with name hookName will be added to the node. The hook activation will be triggered whenever the event eventName occurs for this activity.

Deleting

`Void` **`deleteNodeHook`** `(String hookName)`

Delete a node hook.

`Void` **`deleteNodeInterHook`** `(String hookName)`

Delete a node interHook. The hook or the interHook with name hookName will be deleted from the node.

Managing

`Collection` **`getNodeHooks`**`(String nodeName)` *`(NodeHookValue Collection)`*

Returns the Node hooks of the project.

`Collection` **`getNodeInterHooks`**`(String nodeName)` *`(NodeInterHookValue Collection)`*

Return all the Interactive Node hooks of the project.

`BnNodeInterHookValue` **`getNodeInterHook`**`(String nodeName, String interHook)`

Returns all the node inter hook data associated to the hook of name « interHook » at the node « nodeName).

`String` **`getNodeInterHookValue`**`(String node, String hook)`

This method returns the hook script associated to the hook with name « hook » of this node

5.7.3 Code sample

```
/*****************************************************************/
/************** API Documentation - Sample 3 ********************/
//************** Activities in Project       ********************/
/*****************************************************************/

System.out.println("Activities creation ... ");
try {
    prjSession.addNode("Activity 1",Constants.Nd.AND_JOIN_NODE);
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
try {
    prjSession.addNode("Activity 2",Constants.Nd.AND_JOIN_NODE);
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
try {
    prjSession.addNode("Activity 3",Constants.Nd.AND_JOIN_NODE);
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

System.out.println("Activity 3 definition ... ");
try {
    Date dateLim = new Date(2005,05,02) ;
    prjSession.setNodeDeadline("Activity 3",dateLim.getTime()) ;
    prjSession.setNodeDescription("Activity 3","Activity 3 Description") ;
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

System.out.println("Setting Activities types");
try {
    prjSession.setNodeTraditional("Activity 1");
    prjSession.setNodeAutomatic("Activity 2");
    prjSession.setNodeTraditional("Activity 3");
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

System.out.println("Setting node properties which will not be propagated to other nodes");
try {
    prjSession.setNodeProperty("Activity 1","color","blue",false);
    System.out.println("Setting node properties which will be propagated to other nodes");
    prjSession.setNodeProperty("Activity 1","price","expensive",true);
    prjSession.setNodeProperty("Activity 1","shape","square");
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

System.out.println("Adding edges between activities");
try {
    prjSession.addEdge("Activity 1","Activity 2");
    prjSession.addEdge("Activity 2","Activity 3");
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

System.out.println("Getting names of all the nodes in the project");
Collection nodesNames = prjSession.getNodesNames() ;
j = nodesNames.iterator();
while (j.hasNext())
{
    String nodeName = (String)j.next();
    System.out.println("Node :  " + nodeName + " (anticipable : " + prjSession.getNodeAnticipable(nodeName) + " )");
    Collection nodeProperties = prjSession.getNodeProperties(nodeName) ;
    Iterator k = nodeProperties.iterator() ;
    while (k.hasNext())
    {
            hero.interfaces.BnNodePropertyValue nodeProperty = (hero.interfaces.BnNodePropertyValue)k.next();
            try {
                    String nodePropertyKeyName = nodeProperty.getTheKey();
                    String nodePropertyValue = nodeProperty.getTheValue();
                    System.out.println(" --> Property (Key, Value) : " + nodePropertyKeyName + "/" + nodePropertyValue);
            } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

    }
```

```
System.out.println("Node deletion");
try {
    prjSession.deleteNode("Activity 3") ;
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

System.out.println("Node deletion verification");
try {
  nodesNames = prjSession.getNodesNames() ;
  j = nodesNames.iterator();
  while (j.hasNext())
  {
     String nodeName = (String)j.next();
     System.out.println("Node :  " + nodeName ); }
} catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
```

To see activity, edges and hooks definition, refer to **Sample1xxx** examples in the Bonita samples directory.

5.8 Managing users

5.8.1 Getting the list of all bonita registered users

`Collection` **`getAllUsers`**`()` *(String Collection)*

Return all registered users names registered in Bonita System.

5.8.2 Getting the list of users which are part of a project

`Collection` **`getUsers`**`()` *(String Collection)*

Returns all users of the project.

5.8.3 Adding a user to a project

`void` **`addUser`**`(String username);`

Add a user to this project (This user must exist at bonita database)

5.8.4 Checking whether a user is part of a project

`boolean` **`containsUser`**`(String username);`

Test if the project contains this user

5.8.5 Code sample

```
/*****************************************************************/
 /************** API Documentation - Sample 4 ********************/
 //************** Users in Project       *****************/
 /*****************************************************************/

     System.out.println("  Getting users names of the project ");
     try {
   Collection usersNames = prjSession.getUsers() ;
   j = usersNames.iterator();
   while (j.hasNext())
   {
     String userName = (String)j.next();
     System.out.println("User :  " + userName ); }
     } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

     System.out.println("  Adding John in the project ");
     try {
           prjSession.addUser("john") ;
     } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

processName = prjSession.getName() ;
System.out.println("Current Process : " + processName + " contains john :" + prjSession.containsUser("john") ) ;
```

5.9 Managing roles in a Project

Role is the mean by which User can be associated to activities. A role has a name and a string description.

Roles must be first declared in a project. Then role can be associated to Users and to Activities.

5.9.1 Declaring a new role in the project

```
void  addRole (String roleName, String description);
```

Add a role to the project. Creates a role within this project. The role is specific of this project.

5.9.2 Allocating a role to a User

Roles are allocated to users in the scope of given project. That is, a user may assume different roles in different project. Also, in the scope of a project, an user can assume several roles.

```
void  setUserRole (String userName, String roleName);

void  unsetUserRole (String userName, String roleName);
```

5.9.3 Getting the list of roles that an user can assume

`Collection` **`getUserRoles`** `(String userName)` *(BnRoleLOcal Collection)*

Returns all the roles available for this user (independently of any project).

5.9.4 Getting the list of roles that an user can assume in the scope of a project

`Collection` **`getRoles`**`()` *(BnRoleLocal Collection)*

Returns all roles of the current project. These roles have been associated with the nodes included in the project.

`Collection` **`getRolesNames`**`()` *(String Collection)*

Returns the names of all the roles of the current project as a collection of String objects.

`Collection` **`getUserRolesInProject`**`(String userName)` *(BnRoleValue Collection)*

Returns the roles of this user in the current project.

`Collection` **`getUserRolesInProjectNames`**`(String userName)` *(String Collection)*

Returns the role names of the user in the current project.

5.9.5 Associating an activity with a role

Only a single role can take over a given activity.

`String` **`getNodeRoleName`**`(String nodeName)`

Returns node role name. Obtains the role name of this node.

`void` **`setNodeRole`**`(String name, String role)`

Sets or changes if already exists the role of an activity.

5.10

5.10.1 Code sample

```
/*****************************************************************/
/************** API Documentation - Sample 5 ********************/
//************** Roles in Project       ******************/
/*****************************************************************/

    System.out.println("Adding a Custumer role for john in the current project ");
    try {
            prjSession.setUserRole("john","Customer") ;
    } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

    System.out.println("  Getting role names of the project ");
    try {
 Collection rolesNames = prjSession.getRolesNames() ;
 j = rolesNames.iterator();
 while (j.hasNext())
 {
    String roleName = (String)j.next();
    System.out.println("Role :  " + roleName ); }
    } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

    System.out.println("  Getting role names for john user in this project ");
    try {
 Collection johnRolesNames = prjSession.getRolesNames() ;
 j = johnRolesNames.iterator();
 while (j.hasNext())
 {
    String johnRoleName = (String)j.next();
    System.out.println("John role :  " + johnRoleName );
      }
    } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

  System.out.println("  Setting role names for an activites of this project ");
    try {
            System.out.println("  --> Getting the actuel role names for Activities ");
            try {
                    System.out.println(" --> Activity 1 role :  " + prjSession.getNodeRoleName("Activity 1"));
                    System.out.println(" --> Activity 2 role :  " + prjSession.getNodeRoleName("Activity 2"));
            } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

            System.out.println("  --> Setting activities new roles " );
            try {
                    prjSession.setNodeRole("Activity 1","admin") ;
                    prjSession.setNodeRole("Activity 2","Customer") ;
            } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

            System.out.println("  --> Getting the new role names for Activities ");
            try {
                    System.out.println("     Activity 1 role :  " + prjSession.getNodeRoleName("Activity 1"));
                    System.out.println("     Activity 2 role :  " + prjSession.getNodeRoleName("Activity 2"));
            } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong

    } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
```

5.11 Mappers

`void` **`addRoleMapper`**`(String roleName, String mapperName, int mapperType)`

Add an existing mapper to the role « roleName ». This type of mapper use a Java file loaded at run time.
mapperType can be one of the following :

- *Constants.Mapper.LDAP for a LDAP Mapper*
- *Constants.Mapper.PROPERTIES for a Properties Mapper*
- *Constants.Mapper.CUSTOM for a custom Mapper*

`void` **`deleteRoleMapper`**`(String roleName)`

Delete a role mapper.

`Collection` **`getRoleMappers`**`()` *(BnRoleMapperValue Collection)*

Returns all the role mappers of the project..

5.11.1 Code sample

```
      .../....
        ProjectSessionHome projectSessionh=ProjectSessionUtil.getHome();
        ProjectSession pss=projectSessionh.create();

         String role1="Admintoto";
         pss.addRole(role1, "role added for activity 1");
         String role2="Admintiti";
         pss.addRole(role2, "role added for activity 2");

         // NODE 1
         pss.addNode("h1",Constants.Nd.AND_JOIN_NODE);
         pss.setNodeRole("h1",role1);

         // NODE 2
         pss.addNode("h2",Constants.Nd.AND_JOIN_NODE);
         pss.setNodeRole("h2",role2);

         // add MAPPERS
         pss.addRoleMapper(role1,"hero.mapper.mapper1",Constants.Mapper.LDAP);
      pss.addRoleMapper(role2,"hero.mapper.mapper2",Constants.Mapper.PROPERTIES);

      // Custom mapper : Constants.Mapper.CUSTOM

   pss.instantiateProject(projectName);
....../....
```

Example of Mapper code are available under *$BONITA_HOME/src/resources/mappers/hero/mapper.*

5.12 Performer assignment

5.12.1 Addition of a performer assignment to a node

```
void addNodePerformerAssign(String nodeName,
String performerAssignName,int performerAssignType, String propertyName)
```

Add an existing performerAssign to the node. This type of performerAssign use a Java file loaded at run time.

PerformerAssignType can be one of the following :

- *Constants.Performer.CALLBACK for a Callback Performer Assignment*
- *Constants.Performer.PROPERTIES for a Properties Callback Assignment*

5.12.2 Code sample

```
....//....
   // NODE 1
   pss.addNode("h1",Constants.Nd.AND_JOIN_NODE);
   pss.setNodeRole("h1",role1);

   // NODE 2
   pss.addNode("h2",Constants.Nd.AND_JOIN_NODE);
   pss.setNodeRole("h2",role2);

   // NODE 3
   pss.addNode("h3",Constants.Nd.AND_JOIN_NODE);
   pss.setNodeRole("h3",role3);

...//....

 // activity property
 pss.setNodeProperty("h3","acteurH3","gaillarr");
....//....

 // PERFORMER ASSIGN
  pss.addNodePerformerAssign("h2",
"hero.performerAssign.CallbackSelectActors" ,
Constants.Performer.CALLBACK,"");
pss.addNodePerformerAssign("h3",
"hero.performerAssign.PropertySelectActors" ,
Constants.Performer.PROPERTIES ,"acteurH3");
```

6 USER SESSION INTERFACE

6.1 Principle

The User interface provides access to process execution control functions. The interface is initiated for a given user. Only the processes where the User are declared can be accessed.

In case of EJB Session access, the User interface will automatically retrieve the identity of the calling user in the J2EE security context. Hence, calling the User interface from an unidentified context will fail.

Much of the User interface methods are taking the Project name as parameter. This name may be known directly from the application logic. Alternatively, the application may retrieve the project name according to various search criteria. At the moment, the corresponding search methods are not implemented.

The User Session Bean, is an stateful session bean that provides the user API to get information on Todo list and started activities and to produce events on activities (start, terminate, cancel).
This Session Bean is based on Engine Session Bean: a recursive implementation that manage the previous execution operations and propagates the activity state changes to the activities that are connected to this one.

The User Session Bean API provides information about user projects and activites (project list, todo list and activity list) and also useful information about project instances or user preferences. With this API users can performs his task/activities by using start, terminate and cancel methods and also terminates workflow processes.

You will find below examples of code using this interface.
These are extracts of the SampleUserApi.java example proposed in the “samples” directory of Bonita. You can run this example by using then ant tasks “sample-user-api” .
You can also refer to the sample1xxx classes, which implement the user guide workflow example (Order Processing and Customer Service).
You can run them by using then ant tasks “sample1-create-process-model” (Model creation), “sample1-admin-wf” (user administration and project instantiation), “sample1-running-session” (Process execution).

6.2 Creating the UserSessionBean

The UserSessionBean can be seen as an handle to the connection with the BONITA workflow System. After an user authentication, this handle has to be created, and it is then created under his identity.
Every call to further User Session API function is related to this identity.

Code sample :

```
import javax.security.auth.login.LoginContext;
import hero.client.test.SimpleCallbackHandler;

import hero.interfaces.UserSession;
import hero.interfaces.UserSessionHome;
import hero.interfaces.UserSessionUtil;

import hero.interfaces.Constants;

import java.util.*;

public class SampleUserApi {

        static public void main(String[] args) throws Exception{

      // User Admin login
      char[] password={'t','o','t','o'};
      SimpleCallbackHandler handler = new SimpleCallbackHandler("admin",password);
      LoginContext lc = new LoginContext("TestClient", handler);
      lc.login();

     // User Session Bean Creation using Remote Interface
      UserSessionHome usrHome= (UserSessionHome) UserSessionUtil.getHome();
      UserSession usrSession = usrHome.create();
```

6.3 User Properties

6.3.1 Setting User Properties

`Void` **`setUserProperty`** `(String key, String value)`

Set the property whose name is *key* to the value *value*.

If the property already exists, the current value is overridden. If the properties does not exist, it is created and its value is set to *value*.

`void` **`setUserMail`**`(String userName, String mail)`

Set the mail of this user into Bonita database.

6.3.2 Getting User Information

`String` **`getUser`**`()`

Returns the name of the current authentified User.

`String` **`getUserPassword`**`()`

Returns the user password

`String` **`getUserMail`**`(String userName)`

Returns the mail of this user from Bonita database.

`Collection` **`getUserProperties`**`()` *`(BnUserPropertyValue Collection)`*

Returns the properties defined for the current authentified User.

6.4 User and Projects

6.4.1 Getting the list of projects for the User

`Collection` **`getProjectList`**`()` *`(BnProjectLightValue Collection)`*

Returns Workflow processes associated to this user.

`Collection` **`getProjectListNames`**`()` *`(String Collection)`*

Returns project list names for this user.

`Collection` **`getProjectsByProperty`**`(String key, String value)` *`(BnProjectValue Collection)`*

Returns Workflow projects from a property.

`Collection` ***`getProjectsByPropertyNames`***`(String key, String value)` *`(String Collection)`*

Returns Workflow projects from a property.

6.4.2 Getting the list of instances for the User

`Collection` **`getInstancesList`**`()` *`(BnProjectLightValue Collection)`*

Returns user instances list. This method is equivalent to getProjectList but it only returns the current instances of the user.

`Collection` **`getInstancesListNames`**`()` *`(String Collection)`*

Get instances list names for this user. This method is equivalent to getProjectListNames but it only returns the current instances of the user.

`Collection` **`getProjectInstances`**`(String projectName)` *`(BnProjectValue Collection)`*

Returns Workflow instances of this project.

`Collection` **`getProjectInstancesNames`**`(String projectName)` *`(String Collection)`*

Returns workflow instances names of this project.

`Collection` **`getInstancesByProperty`**`(String key, String value)` `(BnProjectValue Collection)`

Returns Workflow instances from a property.

`Collection` **`getInstancesByPropertyNames`**`(String key, String value)` *`(String Collection)`*

Returns a list of project instances from a property.

6.4.3 Managing the project for the User

`void` **`removeProject`**`(String projectName)`

Delete a Workflow project

Tries to terminate a project (only when all project activities are terminated)

`void` **`terminate`**`(String projectName)`

Tries to terminate a project (only when all project activities are terminated)

6.5 User and Activities

6.5.1 Getting the list of activities for the User

`Collection` **`getActivityList`**`(String projectName)` *`(String Collection)`*

Obtains all user activities for a specific project in executing and anticipating state. See also the **getToDoList** for activities in ready state.

`Collection` **`getActivityListAllInstances`**`()` *`(BnNodeValue Collection)`*

Obtains a list of executing user activities for all instances (ready and anticipable state).

`Collection` **`getActivityListByProperty`**`(String key, String value)` *`(BnNodeValue Collection)`*

Obtains executing user activities matching with property value (executing and anticipating state activities).

6.5.2 Getting Information on User activity

`BnNodeValue` **`getNode`**`(String projectName,String nodeName)`

Returns Node Value from a specific project .

6.5.3 Getting the ToDo list for the User

`Collection` **`getToDoList`**`(String projectName)` *`(String Collection)`*

Obtains all user activities from specific project (ready and anticipable state).

`Collection` **`getToDoListAllInstances`**`()` `(BnNodeValue Collection)`

Obtains the list of todo activities of the user for all instances (ready and anticipable state).

`Collection` **`getToDoListByProperty`**`(String key, String value)` *`(BnNodeValue Collection)`*

Obtains the list of todo activities for the user matching to property value (ready and anticipable state activities).

6.5.4 Managing activities for the User

`void` **`startActivity`**`(String projectName, String nodeName)`

Tries to start an activity (when activity state is ready or anticipable)

`void` **`terminateActivity`**`(String projectName, String nodeName)`

Tries to terminate an activity (when activity state is executing or anticipating)

`void` **`cancelActivity`**`(String projectName, String nodeName)`

Tries to cancel an activity (when activity is executing or anticipating)

6.6 Code sample

```
//*************************************************************/
//************** API Documentation - Sample 6   ***************/
//************** Users and Activities                 *****************/
//*************************************************************/
        System.out.println("Current User Name/Passwd : " + usrSession.getUser() + "/" + usrSession.getUserPassword());

        usrSession.setUserProperty("Language","Spanish");

        System.out.println("Getting Current User properties values" );
Collection properties = usrSession.getUserProperties() ;
Iterator i = properties.iterator();
while (i.hasNext())
{
    hero.interfaces.BnUserPropertyValue property = (hero.interfaces.BnUserPropertyValue)i.next();
    try {
          String propertyKeyName = property.getTheKey();
          String propertyValue = (String)property.getTheValue();
          System.out.println("Property (Key, Value) : " + propertyKeyName + "/" + propertyValue);
    } catch(Exception e) {System.out.println(e);} //Maybe there is a problem
}

        System.out.println("\n Getting project names for this user");
        try {
 Collection prjNames = usrSession.getProjectListNames() ;
 Iterator j = prjNames.iterator();
 while (j.hasNext())
 {
    String prjName = (String)j.next();
    System.out.println(" --> Project :  " + prjName ); }
      } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong


        System.out.println("\n Starting & terminating Activities available for this user");
        try {
 Collection instNames = usrSession.getInstancesListNames() ;
 Iterator j = instNames.iterator();
 while (j.hasNext())
 {
    String instName = (String)j.next();
    System.out.println("--> INSTANCE :  " + instName );

     System.out.println("Getting ToDo list for this instance");
  Collection activityNames = usrSession.getToDoList(instName) ;
  Iterator k = activityNames.iterator();
  while (k.hasNext())
  {
   String activityName = (String)k.next();
   System.out.println("   --> activity :  " + activityName );
                try {
                          usrSession.startActivity(instName,activityName) ;
    System.out.println("   --> activity started" );
                } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
        } // End ToDo list

         System.out.println("Getting the activity List (executing aor anticipating) for yhe user");
  activityNames = usrSession.getActivityList(instName) ;
  k = activityNames.iterator();
  while (k.hasNext())
  {
   String activityName = (String)k.next();
   System.out.println("   --> activity :  " + activityName );
                try {
                          usrSession.terminateActivity(instName,activityName) ;
    System.out.println("   --> activity terminated" );
                } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
        } // End ToDo list

       } // End Intances List

      } catch(Exception e) {System.out.println(" --> " + e);} //Maybe something is wrong
```

7 BONITA ENTITIES

A lot of entry points of the API allow to retrieve data about the process entities, such as the information relevant for a given activity. Although Bonita currently makes use of the Enterprise Java Beans entities to store data, the corresponding information has been made available at the API level as plain old java beans.

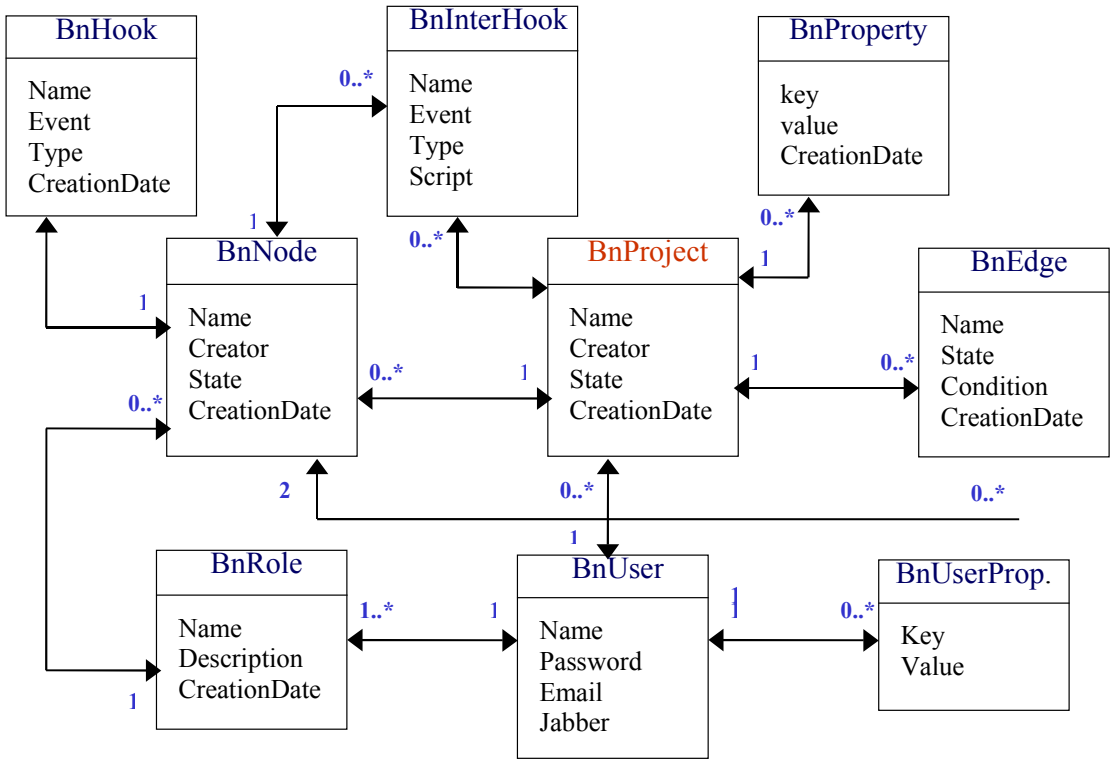
The following is a first level of description of those java beans. For further information, refer to the code in the bonita/build/generate/hero/interfaces directory.

The following naming convention applies for all the entities managed at the API level.

If *Entity* is the name of the internally used Enterprise Java Bean, *EntityValue* is the name of the corresponding plain old java object, *EntityLightValue* is the name of a simpler java object (very often, *EntityLightValue* will have only fields that have simple type).

Would you like to use directly the internal EJB thru their remote or local interfaces (This is a choice that we do not recommend) , each of these entities can be accessed using its name suffixed by hero.interfaces.

7.1 Diagram



7.2 Entities Attributes

7.2.1 BnAuthRoleValue

TYPE	ATTRIBUTE	MEANING
int	**id**	
boolean	**idHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
String	**bnRoleGroup;**	
boolean	**bnRoleGroupHasBeenSet**	
hero.interfaces.BnAuthRolePK	**pk;**	

7.2.2 BnEdgeValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
int	**state;**	
boolean	**stateHasBeenSet**	
String	**condition;**	
boolean	**conditionHasBeenSet**	
java.sql.Date	**creationDate;**	
boolean	**creationDateHasBeenSet**	
java.sql.Date	**modificationDate;**	
boolean	**modificationDateHasBeenSet**	
hero.interfaces.BnNodeValue	**InBnNode;**	
boolean	**InBnNodeHasBeenSet**	
hero.interfaces.BnNodeValue	**OutBnNode;**	
boolean	**OutBnNodeHasBeenSet**	
hero.interfaces.BnEdgePK	**pk;**	

7.2.3 BnInstanceValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
String	**creator;**	

boolean	**creatorHasBeenSet**	
String	**parent;**	
boolean	**parentHasBeenSet**	
int	**state;**	
boolean	**stateHasBeenSet**	
java.util.Date	**creationDate;**	
boolean	**creationDateHasBeenSet**	
java.util.Date	**modificationDate;**	
boolean	**modificationDateHasBeenSet**	
hero.interfaces.BnProjectValue	**javaTree;**	
boolean	**javaTreeHasBeenSet**	
Collection	**BnUsers**	
Collection	**BnRoles**	
Collection	**BnNodes**	
Collection	**BnProperties**	
hero.interfaces.BnInstancePK	**pk;**	

7.2.4 BnIterationValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
String	**fromNode;**	
boolean	**fromNodeHasBeenSet**	
String	**toNode;**	
boolean	**toNodeHasBeenSet**	
String	**condition;**	
boolean	**conditionHasBeenSet**	
hero.interfaces.BnIterationPK	**pk;**	

7.2.5 BnNodeHookValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
String	**event;**	
boolean	**eventHasBeenSet**	
int	**type;**	
boolean	**typeHasBeenSet**	
hero.interfaces.BnNodeHookPK	**pk**	

7.2.6 BnNodeInterHookValue

TYPE	ATTRIBUTE	MEANING
int	**id**	
boolean	**idHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
String	**event**	
boolean	**eventHasBeenSet**	
int	**type**	
boolean	**typeHasBeenSet**	
String	**script**	
boolean	**scriptHasBeenSet**	
hero.interfaces.BnNodeInterHookPK	**pk**	

7.2.7 BnNodePerformerAssignValue

TYPE	ATTRIBUTE	MEANING
int	**id**	
boolean	**idHasBeenSet**	
String	**name**	
boolean	**nameHasBeenSet**	
int	**type;**	
boolean	**typeHasBeenSet**	
String	**propertyName**	
boolean	**propertyNameHasBeenSet**	
hero.interfaces.BnNodePerformerAssignPK	**pk**	

7.2.8 BnNodePropertyValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
String	**theKey;**	
boolean	**theKeyHasBeenSet**	
String	**theValue;**	
boolean	**theValueHasBeenSet**	
boolean	**propagate;**	
boolean	**propagateHasBeenSet**	
hero.interfaces.BnNodePropertyPK	**pk;**	

7.2.9 BnNodeValue

TYPE	ATTRIBUTE	MEANING

int	**id;**	
boolean	**idHasBeenSet**	
int	**type;**	
boolean	**typeHasBeenSet**	
int	**state;**	
boolean	**stateHasBeenSet**	
boolean	**anticipable;**	
boolean	**anticipableHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
String	**description;**	
boolean	**descriptionHasBeenSet**	
String	**activityPerformer;**	
boolean	**activityPerformerHasBeenSet**	
hero.entity.NodeState	**transition;**	
boolean	**transitionHasBeenSet**	
hero.entity.EdgeState	**activation;**	
boolean	**activationHasBeenSet**	
java.util.Date	**startDate;**	
boolean	**startDateHasBeenSet**	
java.util.Date	**endDate;**	
boolean	**endDateHasBeenSet**	
java.util.Date	**deadline;**	
boolean	**deadlineHasBeenSet**	
java.util.Date	**creationDate;**	
boolean	**creationDateHasBeenSet**	
java.util.Date	**modificationDate;**	
boolean	**modificationDateHasBeenSet**	
hero.interfaces.BnUserLightValue	**Creator;**	
boolean	**CreatorHasBeenSet**	
hero.interfaces.BnUserLightValue	**Executor;**	
boolean	**ExecutorHasBeenSet**	
hero.interfaces.BnRoleValue	**BnRole;**	
boolean	**BnRoleHasBeenSet**	
hero.interfaces.BnNodePerformerAssignValue	**BnNodePerformerAssign;**	
boolean	**BnNodePerformerAssignHasBeenSet**	
hero.interfaces.BnProjectLightValue	**BnProject;**	
boolean	**BnProjectHasBeenSet**	
Collection	**BnProperties**	
Collection	**BnHooks**	
Collection	**BnInterHooks**	
hero.interfaces.BnNode	**pk;**	

PK		

7.2.10 BnProjectHookValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
String	**name;**	
boolean	**nameHasBeenSet**	
String	**event;**	
boolean	**eventHasBeenSet**	
int	**type;**	
boolean	**typeHasBeenSet**	
hero.interfaces.BnProjectHookPK	**pk;**	

7.2.11 BnProjectInterHookValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
String	name;	
boolean	nameHasBeenSet	
String	event;	
boolean	eventHasBeenSet	
int	type;	
boolean	typeHasBeenSet	
String	script;	
boolean	scriptHasBeenSet	
hero.interfaces.BnProjectInterHookPK	pk;	

7.2.12 BnProjectPropertyValue

TYPE	ATTRIBUTE	MEANING
Int	id;	
Boolean	idHasBeenSet	
String	theKey;	
Boolean	theKeyHasBeenSet	
String	theValue;	
Boolean	theValueHasBeenSet	
hero.interfaces.BnProjectPropertyPK	pk;	

7.2.13 BnProjectValue

TYPE	ATTRIBUTE	MEANING

Int	id;	
boolean	idHasBeenSet	
int	instanceNs;	
boolean	instanceNsHasBeenSet	
String	parent;	
boolean	parentHasBeenSet	
String	name;	
boolean	nameHasBeenSet	
String	creator;	
boolean	creatorHasBeenSet	
int	state;	
boolean	stateHasBeenSet	
java.util.Date	creationDate;	
boolean	creationDateHasBeenSet	
java.util.Date	modificationDate;	
boolean	modificationDateHasBeenSet	
Collection	BnUsers	
Collection	BnRoles	
Collection	BnInstances	
Collection	BnNodes	
Collection	BnEdges	
Collection	BnAgents	
Collection	BnAgentEdges	
Collection	BnProperties	
Collection	BnIterations	
Collection	BnHooks	
Collection	BnInterHooks	
hero.interfaces.BnProjectPK	pk;	

7.2.14 BnRoleMapperValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
String	name;	
boolean	nameHasBeenSet	
int	type;	
boolean	typeHasBeenSet	
hero.interfaces.BnRoleMapperPK	pk;	

7.2.15 BnRoleValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
String	description;	

boolean	descriptionHasBeenSet	
String	name;	
boolean	nameHasBeenSet	
hero.interfaces.BnRole MapperValue	BnRoleMapper;	
boolean	BnRoleMapperHasBeenSet	
hero.interfaces.BnRoleP K	pk;	

7.2.16 BnUserPropertyValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
String	theKey;	
boolean	theKeyHasBeenSet	
String	theValue;	
boolean	theValueHasBeenSet	
hero.interfaces.BnUserP ropertyPK	pk;	

7.2.17 BnUserValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
String	name;	
boolean	nameHasBeenSet	
String	password;	
boolean	passwordHasBeenSet	
String	email;	
boolean	emailHasBeenSet	
String	jabber;	
boolean	jabberHasBeenSet	
java.sql.Date	creationDate;	
boolean	creationDateHasBeenSet	
java.sql.Date	modificationDate;	
boolean	modificationDateHasBeenSet	
Collection	BnProjects	
Collection	BnInstances	
Collection	BnRoles	
Collection	BnAuthRoles	
hero.interfaces.BnUserP K	pk;	