BONITA Workflow Cooperative System
Application Programming Interface

(Version 3.2)

Christophe Loridan
Miguel Valdés Faura
Anne Géron

BULL R&D

CHANGES TRACK

RÉFÉRENCES	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
2.8	22/03/05	Hooks types were updated. Process status types modification
2.9	11/05/2005	Global document revision. User management paragraph was updated. Hide and Active status description was also updated. New features such Relative Deadlines was added.
3.0	23/06/2005	Project hooks chapter was updated.
3.1	18/12/2005	Hooks at project level was reviewed. Initiators mappers chapter was included.
3.2	22/05/06	Added figures, many comments and changes (R. Perey)

INDEX

Introduction

BONITA is a workflow system featuring innovative features like activities that can start in anticipation, awareness infrastructure allowing user notification 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 or Web Services calls.

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 of an activity. BONITA also supports dynamic modification of an existing process, that is, the Project interface API can be applied to a running process.

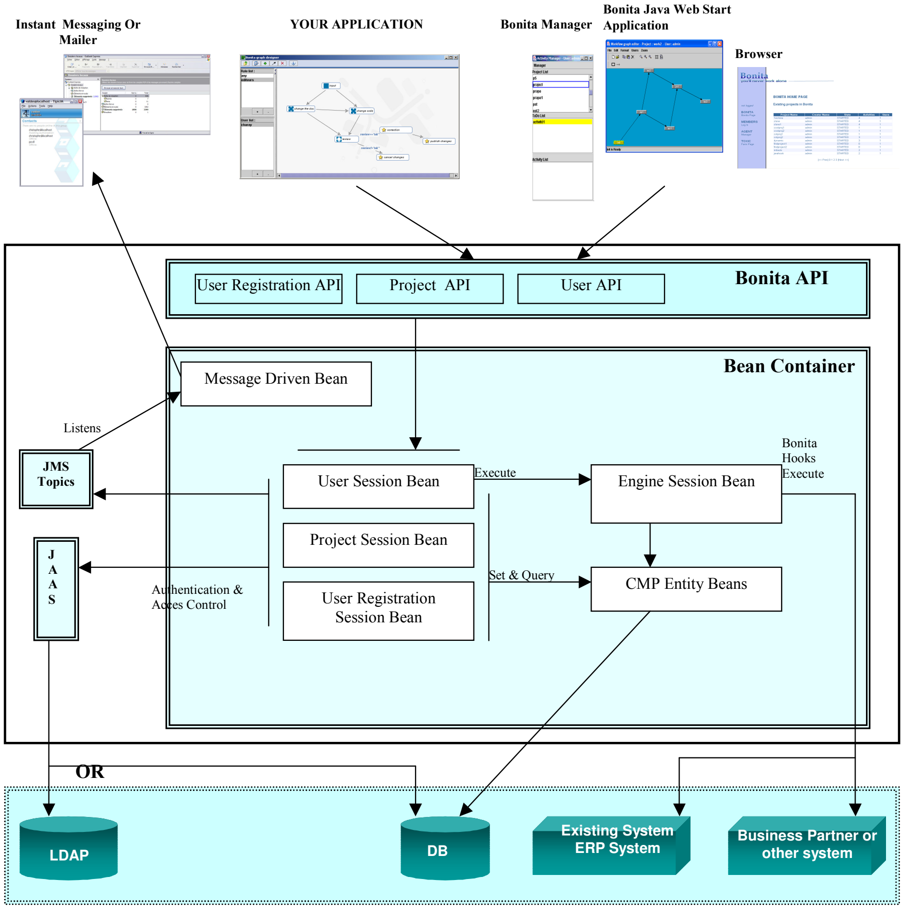


Figure 1 Bonita Workflow

- The **User Registration Session bean** provides 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
    - Listing and Modification of properties
- The **User Session Bean** implements commands and queries related to:
    - Projects of a user
    - Todo Lists
    - Executing activities
    - Start/terminate/Cancel commands
- The **Engine Bean** is a special session bean implementing the state machine and controlling Process execution. The Engine Bean 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 in user creation), the **Message Driven Bean** notifies the user using Instant Messaging services, or a Traditional Mailer.

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

Both User and project APIs are available 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 user defined logic adding automatic behavior to activities/nodes and workflow processes
- A **mapper** is a unit of work allowing dynamic role resolution at workflow instantiation.
- 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 workflow processes. These processes are mapped to three Bonita types:

- **Cooperative**: flexible workflow process allowing definition and execution operations just after the process is created
- **Model**: workflow process containing 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 is controlled by definition or at runtime by the workflow process administrator/s. Two possible statuses are allowed for a workflow process:

- **Active**: the workflow process can be modified or executed. This is the default status for a cooperative, model, or instance process.
- **Hidden**: the process is not yet available. Operations like execution, cancel, or termination of cooperatives and instances projects as well as model instantiation are not allowed. This is the status mode allowing model modifications after instantiation.

1.2.2 Life Cycle

BONITA has a very simple process life cycle

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

- A process is **started** as soon as the first activity starts. While executing, the process definition can be modified using the Project API. When all activities terminate, the process remains in state **started**. The process 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.

1.2.3 Cooperative processes

Bonita has 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 using the User API and be able to add new activities to the process definition. This brings flexibility to workflow participants, and is particularly convenient for so-called cooperative (ad hoc) processes.
You would typically set up a specific process to perform a given job between several colleagues. To allow some level of reuse of process definition, we introduced the concept of **process clone** (see 1.2.5 clone processes).

1.2.4 Models & Instances

There are usage scenarios where the reuse of process definition is of key importance; in these scenarios, a long-time is spent carefully defining a generic process model that instantiates in the same way many times. These processes are called administrative processes (process models).

A process model is a specific definition of a process that may be instantiated many times. They are based on a model-instance workflow paradigm. In this kind of workflow, the Project API is used to define the workflow model. When the process definition is complete, the workflow users are able to instantiate the workflow model via Project API. Once the model instance(s) are created, workflow participants can access the User API to obtain their todolist, to execute assigned activities, or other workflow user functions.
A process model keeps track of all its instances. That is, all instances of this process could be retrieved through the User API.

So, either cooperative or administrative workflows use the same component definition API that is the Project API. Depending on the type of the process created/initialized, this API must be initialized for.

There are also differences between those workflow types concerning process execution. Cooperative workflows are ready for execution and modification from their creation. On the other hand, administrative workflows must be instantiated before. The term process model refers to Bonita projects defined in the context of administrative workflow use.

In future releases of BONITA, the concept of the Process model will be extended with implementation of a Process Model Repository. This allows importing of process definitions in a variety of formats.

Bonita Instantiation mechanism:

Previous versions of the Bonita workflow engine were “duplicating”, in a new process instance, the whole process model (activities, properties, edges, hooks…) as a new clone of the project. This took a long time even for medium workflow processes, and it was a problem for users at instantiation.
Newer Bonita versions (1.4 and later), have been revamped to improve performance. Only the Ready state activities (properties, roles and users if existing) are copied at the creation of the new instance. Once an activity is started, hooks are executed under the Model Hooks (no longer copied). Then, after activity termination, edges and Ready or Executable following activities are copied as well.

Note:

- An instantiated process model can still be modified, but be aware that the modifications may cause errors in the instance execution,

- An instance can still be modified, but be aware that modifications may be conflicting with the model definition applied at execution time.

1.2.5 Clone processes

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

After the cloning operation:
- The process instance has the same set of activities as the process model, with each activity allocated to the same role as in the model. All activities are in initial state, and have the same properties as in the model, with the same associated value. All activities have the same hooks and the same transition conditions as those defined in the original process.
- The process properties are the same as the process model, with the same initial value.
- The users associated to the process are the same as those defined in the first process, and have the same associated roles.
- The process instance can be controlled without restrictions through the User and Project APIs.
- Iterations between activities are the same as in the process model.

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

1.2.6 Concept of Hooks

Hooks are user defined logic that can be triggered at some defined point in the process life cycle. Hook definitions:

- **OnInstantiate hook** must be called before the workflow instance is created. The OnInstantiate hook is not considered to be in the same transaction as the process instantiation action. In fact, this hook is not called directly by the workflow engine. To use OnInstantiate, you must invoke the "executeProcessHook" before the "instantiateProject" operation.
- **OnTerminate hook** is called after workflow instance termination ends. **This hook is not yet implemented.**

1.2.7 SubProcesses

Sometimes, an independently existing business process can take part in another more sophisticated process. Instead redefining the activities, edges, properties, hooks, in the parent process, it could be included as a "SubProcess" of a specific kind of node.
As execution logic is inside the subProcess, subProcess activities are started and terminated automatically by the engine according to the subProcess state.

Creating a subProcess activity:

- When a SubProcess activity is defined in the process model, the sub process is automatically cloned by Bonita as a new process and given the name of the subProcess activity defined in the parent process. Links are maintained between the sub process and the parent Process.

Instantiating a Process with a SubProcess activity:

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

As with any other activity, the 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 as another activity existing in the whole process.

Properties propagation
The properties of the SubProcess Activity in the global Process are propagated as Process properties in the SubProcess, as shown below:

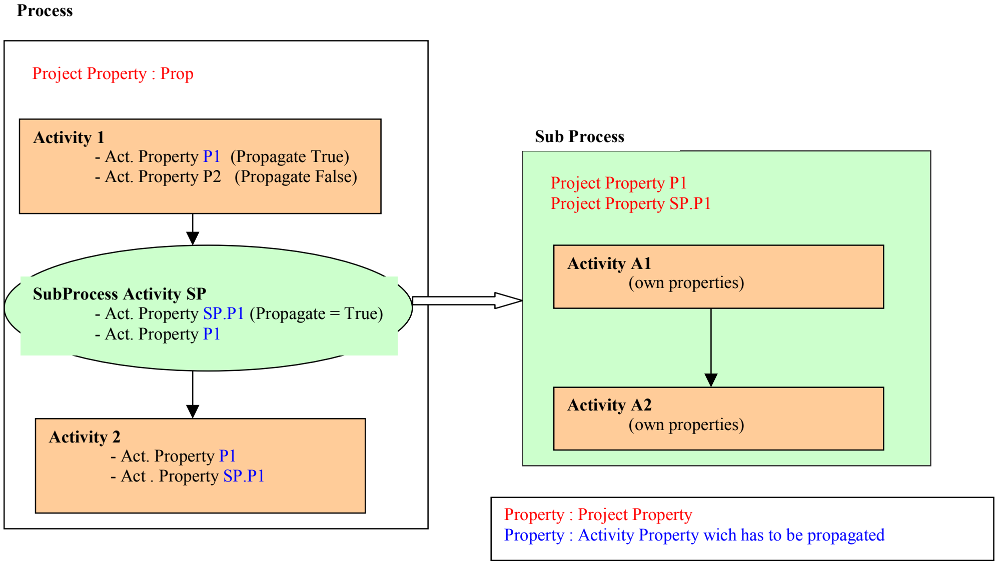


Figure 2 Properties Propagation for SubProcess

1.2.8 Relationship to Users

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

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

Users assuming the Admin role can modify the definition of the process. The Admin role is specific to a process. This means the Admin role for “process1” is different from the Admin role for “process2”.

The User on behalf of whom the project has been created is automatically assigned the Admin role. This User is responsible for the creation of other users in the process, and to allocate roles to other users (including the Admin role that could 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**: The BONITA enactment engine starts the activity when 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 as follows:

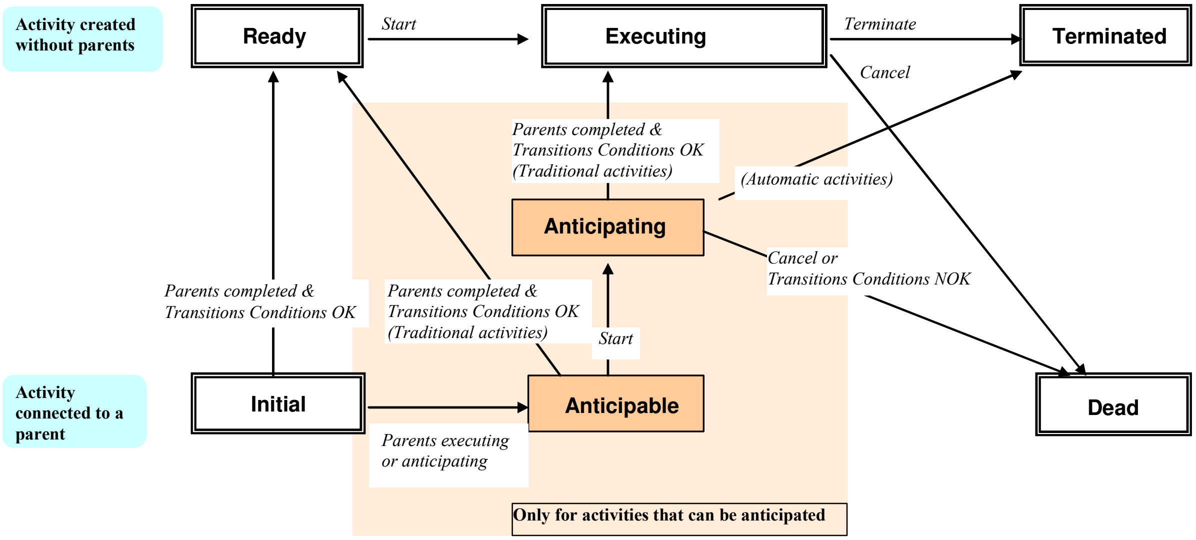

Activity created without parents
Ready
Start
Executing
Terminate
Terminated
Cancel
Parents completed & Transitions Conditions OK (Traditional activities)
(Automatic activities)
Anticipating
Cancel or Transitions Conditions NOK
Parents completed & Transitions Conditions OK
Parents completed & Transitions Conditions OK (Traditional activities)
Start
Activity connected to a parent
Initial
Anticipable
Dead
Parents executing or anticipating
Only for activities that can be anticipated

Figure 3 Activity life cycle

- **Ready:** This is the state of an activity ready to be started. There are two possible situations for this state to occur. In the first, an activity has no parent activity (this is the first activity of the workflow process). In the second, a normal activity has parent activities that have all terminated successfully, and whose transition condition to the activity has been successfully evaluated.
- **Initial:** This is the state of an activity waiting for some processing to complete before being ready to run. In the case of normal activities, at least one of the parent activities is still executing. In the case of an 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 parents activity to complete. All of 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 parent activities have terminated, and the transition conditions have been successfully evaluated.
- **Executing:** An activity in execution.
- **Dead:** An cancelled activity. All dependant activities are automatically cancelled. Cancellation occurs in two cases: explicit cancellation, or unsuccessful evaluation of an inner transition condition.
- **Terminated**: An activity that has terminated successfully.

For **automatic activities**, BONITA **automatically causes:**

- (For *non anticipable activities*) - Transition the state from ready to terminated ,
- (For *anticipable activities*) - Transition the state from anticipable to anticipating ,
- (For *anticipable activities*) - Transition the state from anticipating to terminated whenever all the parents complete.
- Execute any hooks

- Terminate the activity when the executing hooks complete

For activities involving a sub process, the life cycle is shown in Figure 4:

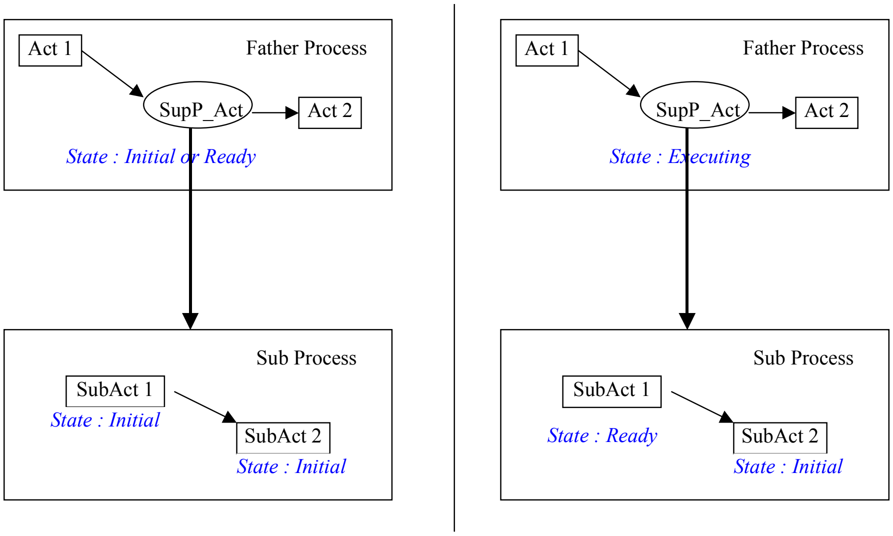


Figure 4 Activity life cycle with sub-process

An **activity** is associated with a **role**. All the users allocated that role in the scope of the process have the possibility to control the activity.

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

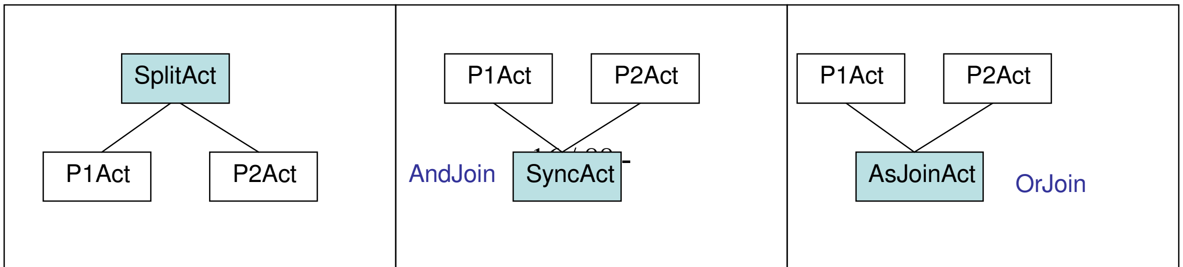
1.3.2 Transition between activities

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

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

The transition pattern is also determined from the number of outgoing edges in an activity; this is the SPLIT construct (this allows several activities to execute in parallel). This is not a specific type of activity; if there are several outgoing nodes from a given activity, it is a SPLIT construct.

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

The SyncAct (synchronous activity) is type AND-JOIN. It will execute only when both P1Act and P2Act are in the terminated state. If one of those activities is cancelled, then SyncAct is also cancelled.

The AsJoinAct (asynchronous activity) is of type OR-JOIN. It will execute whenever either P1Act or P2Act are terminated. If both of these activities are cancelled, then AsJoinAct is also cancelled.

Figure 5 Activity Transitions

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:

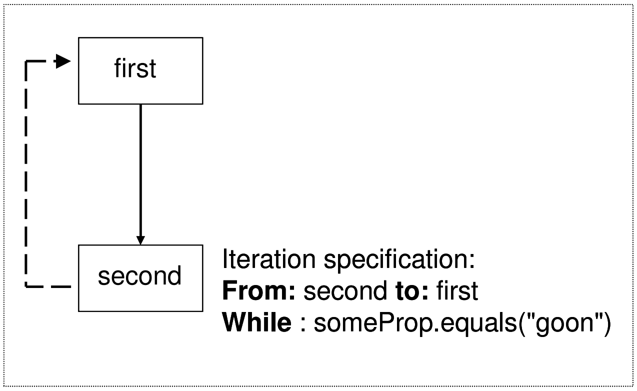
```
Prop.equals (“SomeString”)
(Prop.indexOf (“SomePart”) == 2)
(Prop.lenght() == 9)
(orderType.equals("PO")) && (new Integer(Qte).intValue() > 100)
```

1.3.3 Iterating activities

Bonita supports arbitrary cycles within a process, which means that one or more activities can be executed repetitively.

For this example, attach a single 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.

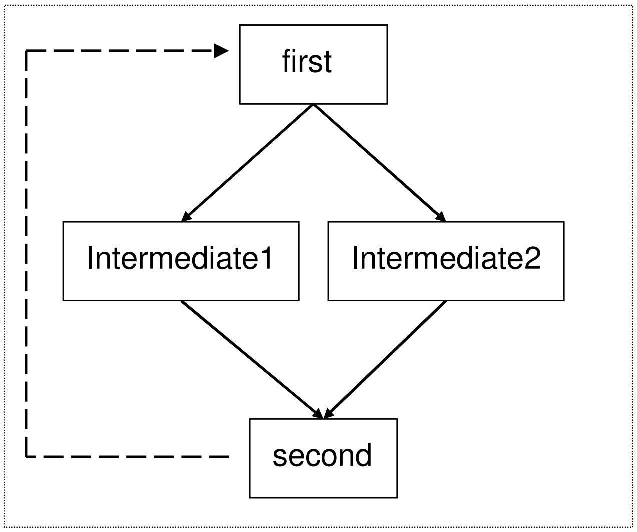
Here is an example of a simple loop:


first
second
Iteration specification:
From: second **to:** first
While : someProp.equals("goon")

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).

Figure 6 Simple Activity Iteration loop

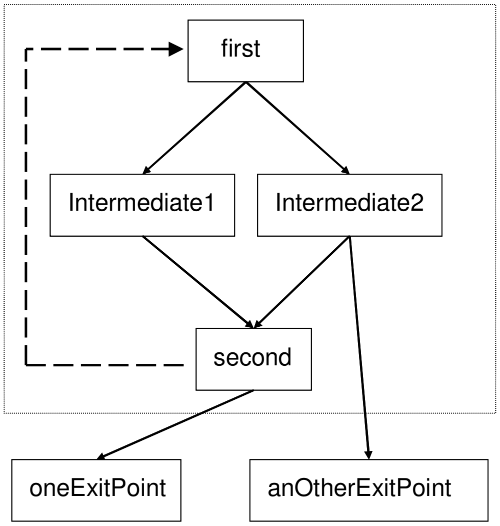
The following example is more complex:



Note that all the execution paths going from activity *first* to activity *second* are included in the cycle, just like in the example to the side, where *intermediate1* and *intermediate2* are iterated several times.

Figure 7 Complex Iteration loop

Iteration's behavior has been modified in Bonita v2.



Old behavior: (Bonita v1 series)

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

New behavior: (Bonita v2)

Frozen mode is removed.
Now, when the iteration is entered, it's possible to exit at any time.
Example: it is possible to exit from *Intermediate2* to *anOtherExitPoint* or to exit from *second* to *oneExitPoint.*

Figure 8 Iteration behavior

Note that with this new behavior we could iterate and leave the iteration at the same and arrive to a point after iterating where it could try to execute an activity that is already terminated.

The following *Guidelines* explain how to design iterations in our model:

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

5. Only one iteration is allowed between two connecting nodes
6. It is possible to have more than one iteration starting in the same node
7. All transitions exiting from a node starting the iteration must meet a condition. If there is more than one transition for exiting from that node, all transitions must meet a condition.
8. If there are multiple exit points within the iteration it is strictly necessary to have conditions on all the transitions exiting from that node. The Conditions must be mutually exclusive for those conditions to take a path to either continue iterating or to exit from the iteration.

NOTE: If guide lines 3 and 4 are not followed it may cause errors during the process execution.

To guarantee that a model is correctly defined and to avoid the problems mentioned above, a new API method has been added:

- ProjectSessionBean.*checkModelDefinition()*: The above guide lines are validated using this method.

This method should be called at the end of a process definition class. For more information see chapter 4.12: Checking model definition.

Below there is a comparison between the old iteration model and the new one:

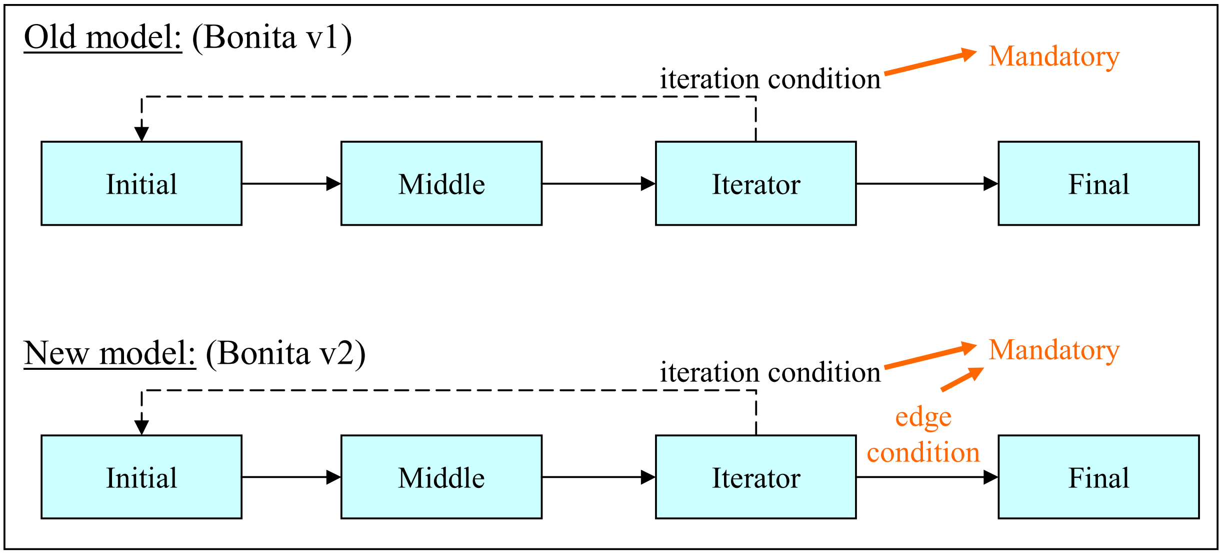


Figure 9 Iteration Model old and new

Guidelines applied:

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

4. Only one iteration is allowed between two connected nodes
5. It is possible to have more than one iteration started in the same node
6. All transitions exiting from a node starting the iteration must meet a condition. If there is more than one transition for exiting from that node, all transitions must meet a condition.

To guarantee this premise, the iteration condition and edge condition must be exclusive. This means that when one is true the other is false.

Only iterations from *Iterator* to *Initial are possible*. Conditions can be a group of conditions like: (((…) && (…)) || (…)). (Remember: only a single iteration between nodes is allowed.)

We can have another iteration starting in the *Iterator* activity going to *Middle* or to *Iterator* itself.

Iterations from *Iterator* to *Final* activity are not allowed because a cycle does not exist.

An edge condition from *Iterator* to *Final* activity is strictly necessary and must be the opposite of the iteration condition. If there are multiple edges outgoing from *Iterator* to other activities, all of them must meet a condition not equal to the iteration condition (this is necessary to accomplish the premise).

Below there is a comparison between the old iteration model and the new model with multiple exit points within the iteration:

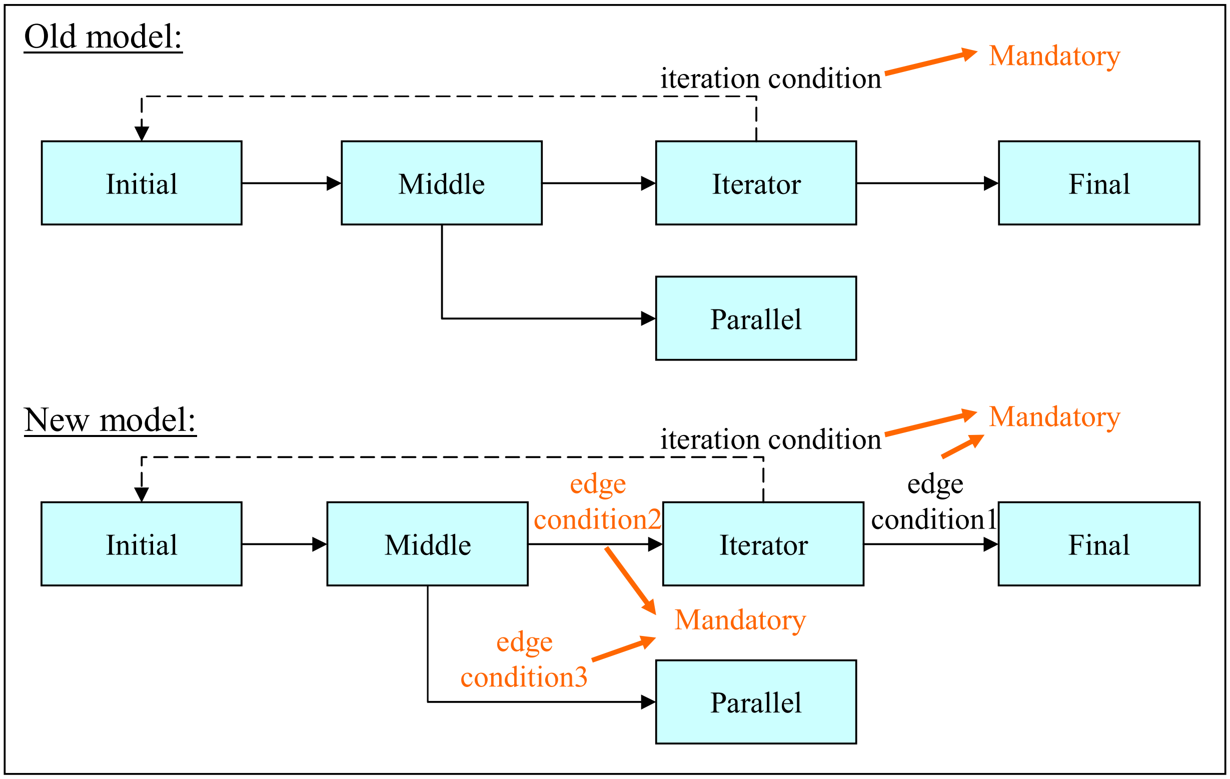


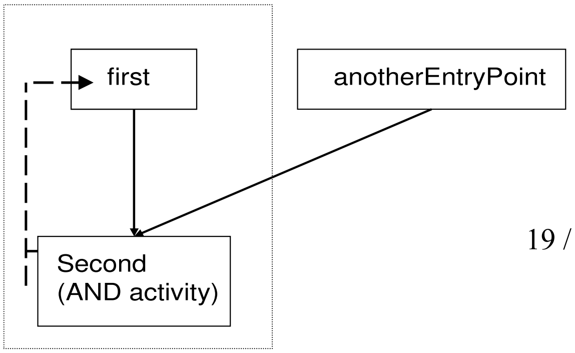
Figure 10 Iteration model with multiple exit points

Guidelines applied:

4. If there are multiple exit points within the iteration, it is strictly necessary to have conditions on all transitions exiting from that node. Conditions must be mutually exclusive for those conditions taking a path to either continue iterating or to exit from the iteration.

The main concept of these new constraints is to guarantee that the execution path does not arrive at an activity whose state is ”Terminated” or indeterminate while executing the iteration.
Remember, edge condition2 and *edge condition3* must be exclusive.

It is also possible to have multiple entry points into iterations, as shown in the following example:



Assume that the iteration is declared between *second* and *first* as in the example to the left:

Because *second* is an AND activity, it starts only when activity “*anotherEntryPoint”* has

Figure 11 Iteration entry points

terminated. This is only true for the first occurrence of *second*: for subsequent executions of this iteration, the incoming transitions from *anotherEntryPoint* are ignored.

1.3.4 Hook Concepts

Hooks are user defined logic that can be triggered at defined points in the life of an 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 as the activity. The Before Start hook is not triggered for automatic activities.
- **After Start hook** is called just after an activity starts. It is considered to be in the same transaction as the activity. The After Start hook is not triggered for automatic activities that cannot be anticipated.
- **Cancel hook** is called before canceling an activity and it's considered to be in the same transaction as the activity.
- **Before Terminate hook** is called just before an activity terminates. The Before Terminate hook is considered to be in the same transaction as the activity.
- **After Terminate hook** is called just after the activity has terminated. It is not considered to be in the same transaction as the activity.
- **Anticipating hook** is called when an automatic activity is started if the activity is anticipable. It is considered to be in the same transaction as the activity.
- **OnReady hook** is called when an activity becomes ready, so it would be very useful to notify the user responsible for executing it. It is not considered to be in the same transaction as the activity.
- **OnDeadline hook** is called when the activity deadline expires. It is not considered to be in the same transaction as the activity.

Hook Fault management

If an exception occurs during the execution of a hook, the error is propagated to the application having triggered the execution of the hook.

Consider the following simple scenario:

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

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

- Imagine that the terminate Activity statement in "Activity 1" completes normally, and "Activity 1" has an outgoing edge defined for automatic activity "Activity 2".
- "Activity 2" is 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".
- This means "Activity1" does not terminate (the activity stays in the executing state) and the system throws an exception due to the "Activity2" execution error.

The above examples show two error scenarios related to transactional hooks execution. **Be aware that Hooks can be executed in a transactional or in a non-transactional context, depending on their types (i.e. before start, after start, …)**

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

- Any changes performed on a transactional resource are included in the existing transactional context.
- Any exception raised by the Hook aborts the existing transaction, so the activity is re-executed later. Furthermore, all operations executed by the hook before the exception was raised are rolled-back.

Bonita also has the capability to create hooks for executing outside a transactional context. In that case, Before Start and After Terminate hooks are executed outside the activity transactional context.

- **It is extremely recommended not to use these hooks (Before Start and After Terminate), to access Bonita APIs or other transactional APIs.**
- **If one of these hooks fails during its execution, the system throws an exception but the activity starts/terminates without roll-back on the operation.**

Consider the last sample scenario described previously and change Before Terminate hook by After Terminate hook. Let’s go over the execution:

- Imagine that the terminate Activity statement on “Activity 1” completes normally, and that “Activity 1” has a defined outgoing edge to 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”.
- This means, “Activity1” terminates without problem, but the system throws an exception due to “Activity2” execution error.

1.3.5 Activity/hooks and transactions

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

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

BONITA aborts the transaction in two cases:

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

When Hooks are executed in a transactional context:

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

1.3.6 Practical steps in hook usage:

Loading Hooks

Hooks code can be stored in the Bonita database as *beanshell* programs. This type of hook is called an Interactive Hook, or "InterHook". To use an Interactive hook,store the hook programs in the Bonita database, either through 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). At execution time, the Bonita executive takes care of importing the code from the Bonita database.

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

- Create your source .java file, (i.e. *MyHook.java)*. The code must be within the package *hero.hook*.
- Copy your java source file to the directory *$BONITA_HOME/src/resources/hooks/hero/hook*
- Go to the *$BONITA_HOME* directory and type: *ant deployHook -DhookClass=<name of your java source file*>. For example: *ant deployHook –DhookClass=MyHook*

Hooks Interface

All hooks must implement the hook interface (*hero.hook.NodeHookI*). This interface is quite simple, with a single method having two parameters: an object EngineBean which is a session bean allowing access to 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.

- Direct use of the EngineBean object is not recommended.
- The BnNodeLocal object can be used to retrieve information about the currently executing activity.

1.4 User Interface

BONITA makes the distinction between Users and Participants:

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

First, a **user** must be registered in the Bonita System for authentication (using the **Bonita User Registration API**). Then, the user must be declared as a **participant** in each project they are involved in (using **Bonita Project API**). The user is then able to take part in the process.

Users are managed in a BONITA specific database (or thru a LDAP repository). This database allows storing of properties (also called preferences) for a given user. Properties are (key, value) pairs where both key and value 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.

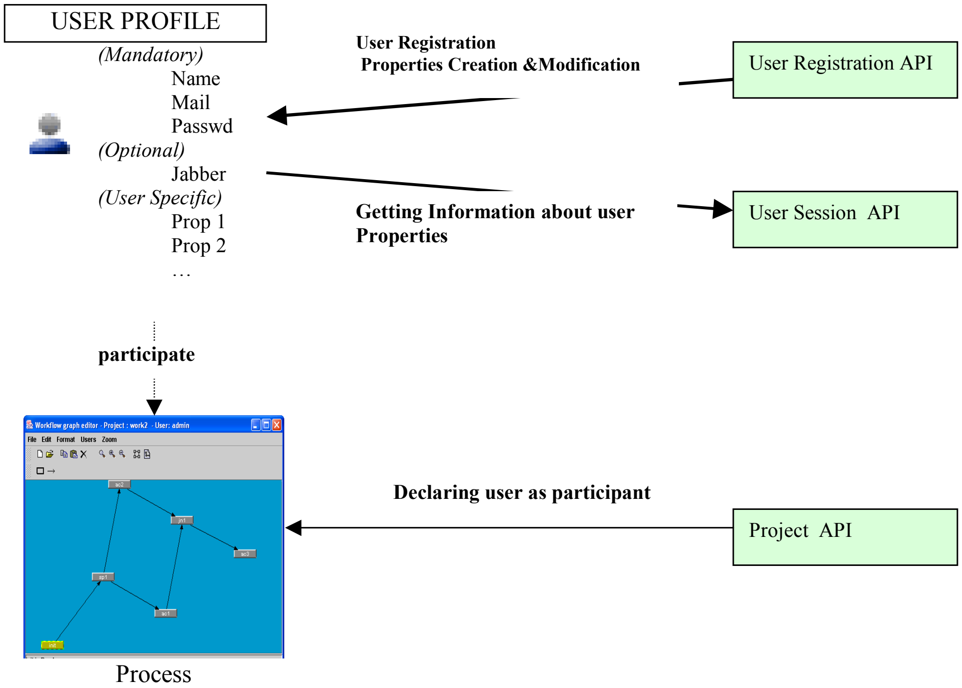

USER PROFILE
(Mandatory)
Name
Mail
Passwd
(Optional)
Jabber
(User Specific)
Prop 1
Prop 2
…
User Registration
Properties Creation &Modification
User Registration API
Getting Information about user Properties
User Session API
participate
Workflow graph editor - Project : work2 - User: admin
File Edit Format Users Zoom
Declaring user as participant
Project API
Process

Figure 12 Bonita User Database

1.4.1 User relationship to processes

Users must be explicitly associated to processes in order to participate and to have visibility of events occurring in those processes.

Two scenarios allow associating a User with a process (making 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 assumes the Admin role in the scope of the process.

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

1.4.2 User authentication scenario

BONITA performs User Authentication using either a specific database (i.e. mySql, Postgres, …), or a Ldap repository. Following is an example of code running authentication of the admin user. It uses the “TestClient” login context implemented in Bonita.
All other users are 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();

          ...
}
```

Figure 13 User authentication example

1.5 User Roles

1.5.1 J2EE Roles

The User Registration Interface, which allows creation of users in the Bonita database, is accessible without role restrictions. This means anyone can call its methods, with no need for authentication.

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

When created with the User Registration Interface, a Bonita user is automatically assigned the “Admin” J2EE role. Those users can 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, clone a process, or instantiate an existing process.

This J2EE security policy can be modified to enforce access control to Bonita Java beans methods, but in that case, be aware that Bonita beans source code has to be adapted to your policy (especially if you modify role names). If you use this security option, migration to newer Bonita versions is more difficult.

It is strongly recommended to leave the Bonita way of running as it stands, and to implement any user access restrictions using 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 within processes. Each Process has its specific role management. This permits differing semantics to associate to the same role name in the scope of two different processes.

Activities are associated with roles. A user, assuming a given role, administers an activity. There is a single role associated with each activity.
Users participate in a project, and within the scope of this project, a user can assume one or several roles.

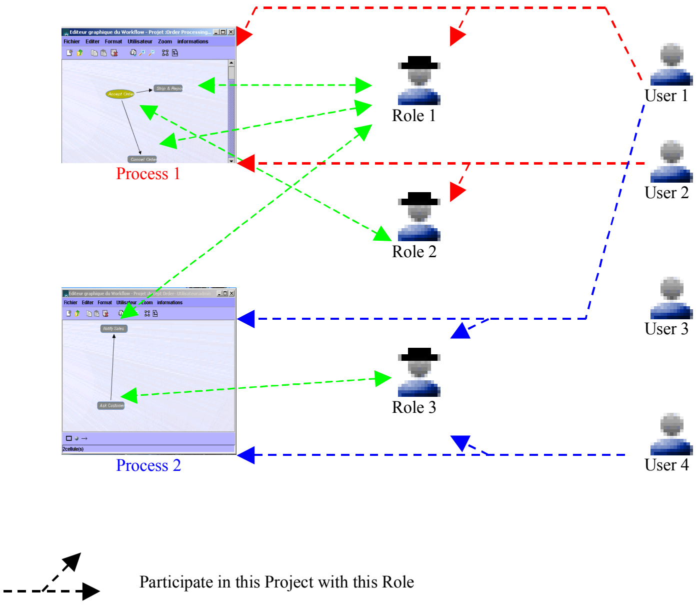


Figure 14 Bonita Roles

User2 and User1 have to execute Process1 alternatively. User1 can also execute all process2 activities due to User1 accreditation in roles for Process1 and Process2.

Note:
Despite that User 3 has no role to play in any process, Bonita User3 would be able to clone or instantiate (but not modify), any process. User3 only needs 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 API methods return the name of an existing process User3 is not involved in
- After instantiation, User3 is not able to start any activity due to standard Bonita role access control.

Default Bonita roles:
Bonita handles two pre-existing roles: “admin” and “InitialRole”. When created, an activity is automatically associated with the “InitialRole”. This role is modified to suit application functional requirements.
The InitialRole may be left as is for the first activity of the Workflow Process. This role could be granted to a participant of the process in charge of starting the workflow. This may be done independently of other functional roles that this activity may have in the process.

Additionally, this role could be left in place for automatic activities not required by other users.

1.5.3 Application Access control

As mentioned previously, standard Bonita access control is open and allows adaptation to organizational needs.

The Bonita access control mechanism has a basic authentication scenario based on workflow projects roles:

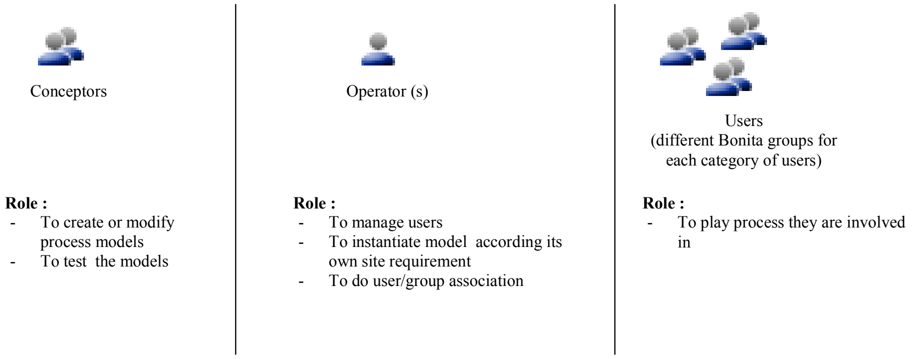
- A User creating the project becomes the admin of the project (user is assigned admin role).
- Only this admin user can add other participants/users to this project.
- Only admin users can modify the project (set, add and delete entities).
- Users taking part in the project are authorized to obtain project information (get entities data).
- Project hooks and mappers may contain confidential information, so get data methods are available to admin users only.
- Participants of the project can set/update properties of activities in which they have a corresponding role.

The Bonita Graph Editor application follows these constraints:

- Only the creator of a process and the users assigned the Bonita “admin” role can modify the process.
- Even if assigned a role to play in this process, another user cannot add, delete, or modify, any node within the process. That user though is able to visualize the evolution of the process.

For example, a typical workflow application distinguishes three categories of users:

- Conceptor
- Operator
- User



Role :
- To create or modify process models
- To test the models

Role :
- To manage users
- To instantiate model according its own site requirement
- To do user/group association

Role :
- To play process they are involved in

Figure 15 User catagories

The application interface (specifically the graphical interface), implements methods to restrain users actions.

Application restrictions could implement stronger access control than Bonita. It is advised not to implement lesser access control than the Bonita standard access control based on points mentioned previously.

- 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

The Mappers feature permits automatic definition of the Bonita roles as defined in the project model when the project is instantiated.

Three methods are available (3 types of mappers), depending on the method used to retrieve users in the system

- using an LDAP server to obtain groups/roles (*LDAP mapper*)
- calling a java class to request a database (*custom mapper*)
- getting the initiator of the project instance (*properties mapper*)

As with other definitions of process elements, access to this functionality is performed through the Bonita API (See the ProjectSessionBean API). Access is also available using the *graphEditor* (ProEd) application.
The Mapper function is particularly interesting for process instantiation usage of the Bonita workflow System. The automatic filling in of groups/roles happens at the first instantiation of the project model (for both the project model and the first instance). Thereafter it happens with each instance creation.

1.6.2 LDAP, Custom, and Properties Mappers

LDAP mapper:

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

- LDAP mapper specifics:
    - The location of the LDAP groups. This depends on the attributes: *roleDN* and *roleNameAttribute*.
    - There is no mapping between roles/groups defined in LDAP and roles defined in the Bonita database (same name for both bases).
    - The attribute name: *uid* is used to provide mapping between the actor identifier in the LDAP base and the userName in the Bonita database.
    - If the group does not exist in the LDAP an exception is thrown.
    - Users found in the groups must be deployed before usage of the mapper function. Otherwise an exception is thrown.
    - The name of the mapper may be any name.

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

Custom mapper

This mapper provides process developers use of their own user's storage base. When this type of mapper is utilized, a call to a java class is performed. The name of this mapper is the name of the called java class (i.e.: *hero.mapper.CustomSeachGroup*), located under *BONITA_HOME\src\resources\mappers\hero\mapper*. After retrieving user information, it must be added to the project instance and also to the targeted role. The Bonita workflow engine loads and executes these mapper classes at runtime. If you add a custom mapper, please follow the next steps:

- Look at the sample class above and implement the custom mapper logic in a new java file.
- Create a source .java file, i.e. *MyMapper.java*. It must be within the package *hero.mapper*.
- Copy the java source file created above into the directory *$BONITA_HOME/src/resources/mappers/hero/mapper*
- Go to *$BONITA_HOME* directory and type: *ant deployMapper -DmapperClass=<name of java source file>*. For example, : *ant deployMapper –DmapperClass=MyMapper*

Properties mapper

Presently, 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 useful for administrative workflow processes to assign the role specified in the property to the user instantiating the process.

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

1.7 Performer Assignment

Performer Assignment increases Bonita functionality by providing a means to modify standard assignment rules for activities.

1.7.1 Introduction

This feature permits additional assignment rules other than those defined in the standard Bonita model.
In the standard model (which is oriented toward cooperative workflow), all users defined in the group associated to the activity can see and execute (the ***toDo List***) within this group.
By adding Performer functionality, a specified user 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*)
- **Dynamically assign the activity to a user** by using an *activity property* (*properties performer assignment*)

Under Performer Assignment functionality, the user is notified (via mail notification), that the activity is ready to start.
The users of the groups, (called roles in Bonita), associated to the activity see the activity but cannot start or terminate it.

Performer assignment functionality is accessible within the Bonita API (see ***ProjectSessionBean API***), and within the Bonita ***graphEditor*** application.
Furthermore, an activity can be assigned to the initiator of the instance. This requires the use of a properties mapper as described above.

1.7.2 Description of performer assignments

Callback performer assignment

Callback performer assignment allows the process developer to code a request with its own algorithm of user selection. When callback **performer assignment** is used, a call to a java class is performed.

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

- Look at the sample class specified above and implement the performer assignment logic in a new java file.
- Create a source .java file, i.e. *MyPerformer.java*. It must be within the package *hero.performer*.
- Copy the java source file into the directory *$BONITA_HOME/src/resources/performers/hero/performer*

- Go to *$BONITA_HOME* directory and type: *ant deployPerformer -DperformerClass=<name of you java source file>*. For example: *ant deployPerformer –DperformerClass=MyPerformer*

Properties performer assignment

This allows the process developer to provide, at **properties performer assignment** creation, the activity property used by the workflow engine to assign the activity. This activity property must be defined either within a previously started activity, with the property propagation, or within the targeted activity about to be assigned.

1.8 Initiator Mapper

This feature implements restrictions to the workflow models in Bonita.

1.8.1 Introduction

The Initiator Mapper feature adds additional security constraints to the workflow instantiation operation.

Through use of Initiator Mapper, the definition of users with permission to instantiate a particular workflow models is restricted (normally all users by default may instantiate).

Initiator Mapper functionality permits:

- **Access to the LDAP directory to dynamically resolve the list of users permitted to instantiate a workflow process. This depends on** the LDAP logic organization using the default LDAP Initiator.
- **Dynamic resolution of the list of users allowed to instantiate the workflow model.** This depends on logic implementing a Custom Initiator

Initiator Mapper functionality is implemented in the Bonita API (see ***ProjectSessionBean API***). The resolution of this entity is done at getModels execution time.

1.8.2 Initiator description

__Custom Initiator__

The Custom Initiator permits the process developer to code a request with its own algorithm of user selection. When this type of custom **initiator mapper** is added, a call to a java class is performed.

The name of this Custom Initiator is the name of the called java class (i.e: *hero.initiatorMapper.CustomGroupMembers.java*) located under $*BONITA_HOME\src\resources\iniitatorMappers\hero\initiatorMapper*. As mappers and performer assignments, your custom initiators are loaded and executed by the Bonita workflow engine. If adding a custom initiator, follow the these steps:

- Look at the sample class above and implement initiator logic within a new java file.
- Create a source .java file, i.e. *MyInitiator.java*. It must be within the package *hero.initiatorMapper*.
- Copy the java source file into the directory *$BONITA_HOME/ src\resources\iniitatorMappers\hero\initiatorMapper*
- Go to *$BONITA_HOME* directory and type: *ant deployInitiator -DinitiatorClass=<name of you java source file*>. For example, : *ant deployInitiatorMapper –DinitiatorMapperClass=MyInitiator*

LDAP Initiator:

The LDAP Initiator uses the LDAP directory to retrieve users corresponding to a specific role defined in a Bonita Workflow project. Please refer to the documentation (Bonita LDAP configuration for JOnAS) to implement this type of initiator.

- The LDAP initiator specifies:
  - The location of the LDAP groups. This depends on the attributes: *roleDN* and *roleNameAttribute*
  - There is no mapping between roles/groups in the LDAP directory and roles specified in the Bonita database (same name may exist in both bases).
  - The attribute name: *uid* is used to provide the mapping between the actor identifier in the LDAP base and the userName specified in the Bonita base.
  - If the specified LDAP group does not exist an exception is thrown.
  - Users found in the LDAP groups must be deployed before using the mapper function. Otherwise an exception is thrown.
  - The name of the initiator may be any name
- Limitations within this version of Bonita Workflow:
  - Groups cannot be recursive. Group inclusions are ignored.
  - There is no verification that the distinguished names (dn) specified for users found in the LDAP groups are compatible with the LDAP tree containing users defined in the JOnAS LDAP realm configuration.

2 USER MANAGEMENT

2.1.1 Bonita User Management basic configuration

After Bonita installation and configuration, user specific data is stored in the Bonita database chosen during the configuration phase. This consists of tables created in the Bonita database providing security control and user management as shown below.

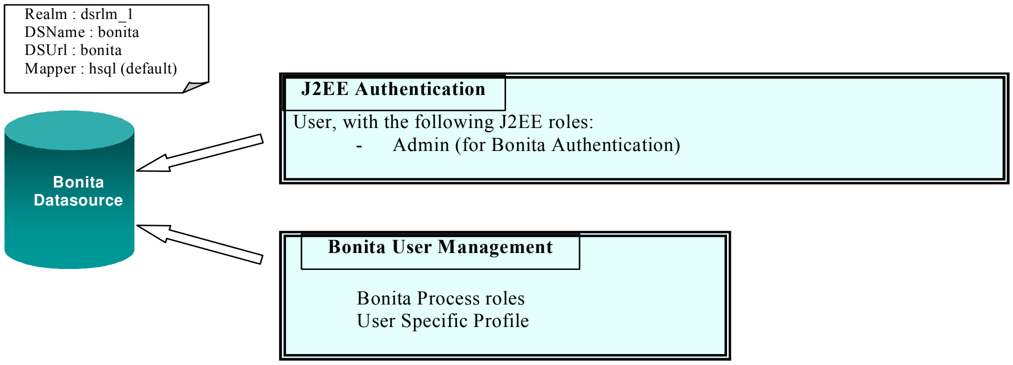


Figure 16 User Management basic configuration

This basic configuration could be changed by user preference. For example, the configuration is modified to utilize an existing user defined database or to use an enterprise LDAP Directory.

2.1.2 Changing the basic configuration

User Management may move to the following schema to make an application fully integrate an enterprise Information System. Bonita takes advantage of User Management defined at upper levels to interface with the workflow application.

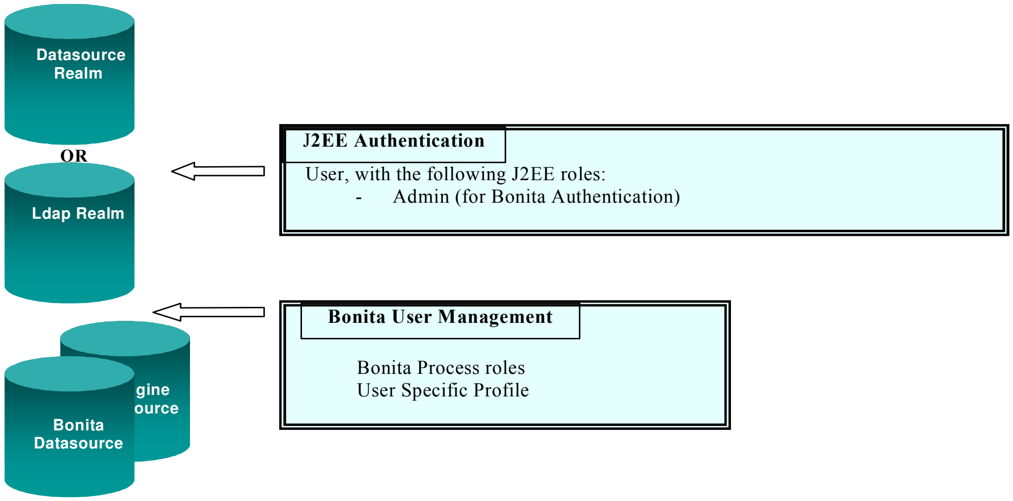


Figure 17 User management with J2EE authentication

2.1.2.1 J2EE Authentication

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

To use another Datasource Security Realm:

- Modify the existing datasource (called dsrlm_1) with selected user and roles queries.

To use an LDAP Security Realm

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

2.1.2.2 Bonita User Management

By default, Bonita uses the *hero.user.DefaultUserBase* implementation class to manage users. To add a User management class:

- Implement the *hero.user.UserBase* interface that provides users required information dealing with the specified user's management system (database, LDAP directory, User Interface…). This class must be located within the *hero.user* package.

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

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

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

3 USER REGISTRATION INTERFACE

3.1 Principle

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

For EJB Session access, the User Registration interface automatically retrieves the identity of the calling user in the J2EE security context. Because of this, calling the User Registration interface from an unidentified context fails.

Also, the Bonita source now permits only users with “Admin” or “users” J2EE roles to access Project and User Session Interfaces.

*** Important Note: UserRegistration API should only be used when the User Management configuration is the Bonita default configuration!! If you are using your own User Management implementation do not use the UserRegistration API.**

3.2 Creating the UserRegistrationBean

The UserRegistrationBean may be seen as a handle for adding a new user or role in the J2EE Application Server security context. First create the handle, and then 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();

          …
}
```

Figure 18 Code sample creating UserRegistrationBean API

3.3 Managing Users

3.3.1 Creating Users

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

This creates a user account. The user is automatically assigned to the “Admin” group.

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

This creates a user account with an instant messaging or mail address. The user is automatically assigned to the “Admin” group.

3.3.2 Defining Users

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

Set a new property for the specified user “*username*”. User properties define user preferences. User properties are a key/value pair.

`void setUserRole(String userName, String roleName)`

Set a new authorization role for the user.

3.3.3 Delete User

`void deleteUser(String userName)`

Delete a user from the Bonita database. If the specified user (“username”) is included in active projects this method throws an exception.

3.4 Roles

3.4.1 Creating Roles

`void roleCreate(String name, String roleGroup)`

This creates a new authorization role in the system for “name”. This 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 confused with Bonita roles associated with projects.

This function is useful for changing the defaults roles of Bonita and allows more precise control over access rights.

3.5 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 usrReg = 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();
}
```

Figure 19 Code sample creating Roles

4 PROJECT INTERFACE

4.1 Principle

The Project interface provides access to functions, permitting modification of execution for a given process.

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

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

An example of code using this interface is shown in Figure 20 below.

The code is from extracts of the SampleProjectApi.java example contained in the “samples” directory of Bonita. This example may be executed by entering “ant sample-project-api”.
Also, refer to the sample1xxx classes, which implement the user guide workflow example (Order Processing and Customer Service).
Execute the examples by entering “ant sample1-create-process-model” (Model creation), “ant sample1-admin-wf” (user administration and project instantiation), “ant sample1-running-session” (Process execution).

4.2 Creating the ProjectSessionBean

The ProjetSessionBean provides a handle into the BONITA workflow System. First, create the handle and then associate a given project to this handle 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();
```

Figure 20 Code Sample creating ProjectSessionBean

4.3 Initiating the ProjectSessionBean

4.3.1 Initiating the Session Bean (Cooperative projects & instances)

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

Creates or initializes a cooperative workflow project. This method may be used to initialize workflow instances.

The Project interface is initialized with the parameter name “projectName”.
If “*projectName”* does not exist, a new empty project is created and given this name.
By default, the user is assigned the Bonita “admin” role for this project. There are no restrictions on the number of characters in the process/project name.

4.3.2 Initiating the Session Bean (Models)

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

Creates or initializes workflow models.

The Project interface is initialized with the given string “modelName”. If *modelName* does not exist, a new empty model is created and given this name. By default, the user is assigned the Bonita “admin” role for this project. There are no restrictions on the number of characters in the process/project name.

4.3.3 Initiating a project using the clone project creation option

`Void` **`initProject`** `(String oldProject, String newProject)`
The Project interface is initialized after existing project with the name “oldProject” is cloned. This interface is initialized with the given *newProject* project name.
See Figure 21 below for an exaple.

NOTE: After using the initProject method, all subsequent interface methods deal with the corresponding project.

4.3.4 Initiating using the instantiate project creation option

`Void` **`instantiateProject`** `(String modelName)`
The Project interface is initialized after new project instance is created. This interface is initialized with the new project instance name given by Bonita automatically.
Bonita derives the instance name from the model name as follows:
<instance-name> = <model-name>_instance<sequence-number>
All subsequent interface methods 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 (if a RoleMapper entity was not defined). Also, users must be assigned roles to start/stop activities in this new project.

NOTE: Only workflow models can be instantiated. Cooperative projects are ready-to-define, ready-to-execute just after creation.

4.3.5 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
```

Figure 21 Code sample Instantiating project

4.4 Managing project

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

- Processes may exist by themselves without a relationship to a process model. In this category processes are created from scratch, or cloned from parent processes.
- A process may be a process model, from which process instances could be derived. Presently, a process model may be executed as well, but this behavior will be removed in the near future.
- Process instances are specific executable processes whose definitions are contained in a process model. At creation time, the specific context of this instance is taken into account to make the instance unique.

4.4.1 Project attributes

A project has a name, assigned at creation time thru the Project API.

The names of process instances are constrained. BONITA automatically allocates a name using following pattern:

*<Project Model Name>*_instance*<Project Instance Number>*.

The <*Project Instance Number*> is automatically assigned and managed by BONITA.

A project has properties, which are string key/value pairs. Enumeration String types are also permitted.

A project records the name of the user creating the project and the project creation date.

The constant values associated with process states are:

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

Figure 22 Process State constant values

The constant values associated with process types are:

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

Figure 23 Process Types constant values

The constant values associated with process status are:

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

Figure 24 Process Status constant values

4.4.2 Active/Hide a workflow process

public void **activeProcess**()

> This sets the process status to Active (model/cooperative/instance).
> Workflow processes can only be executed or instantiated if status equals active (see Figure 24.)

public void **hideProcess**()

> This sets the process status to Hidden (model/cooperative/instance).
> Hide this workflow process. This state allows workflow model modifications once they are instantiated.

4.4.3 Getting the name of a project or an instance

`public String getName();`

> Returns the name of the project being managed by the current instance of the ProjectSessionBean interface

`public String  getProjectNameOfInstance(String instanceName)`

Return the project name for the instance “instanceName”.

4.4.4 Getting the name of the parent project

`public String getParent()`

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

4.4.5 Getting the name of a project’s creator

`String  getCreator();`

Return a string with the name of the user creating the current Project. The projects creator name is retrieved automatically by the BONITA executive after a project is created thru the ProjectSessionBean Interface.

4.4.6 Properties

`void  setProperty (String key, String value)`

Creates a new property key named “key” and assigns the value “value” if the named “key” does not exist. .If the name “key” already exists, this function overrides the value of the existing property with the new value.

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

Return a collection of all properties existing for this project.

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

Return a collection of all the properties keys for the current project. The property is a pair key/value representing workflow relevant data.

`BnProjectPropertyValue getProperty(String key)`

Return the property value of the project for the specified key. The property is from a key/value pair associated to this project.

`void  deleteProperty(String key);`

Deletes a property of an existing project based on the specified key. The key/value property is removed.

4.4.7 Project details

`public BnProjectValue getDetails()`

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

4.4.8 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
 }
```

Figure 25 Code sample Project Properties

4.5 Defining and Getting Information about activities

4.5.1 Types of activities

An Activity type can be one of the following as defined in the Constant values in Figure 27 below:

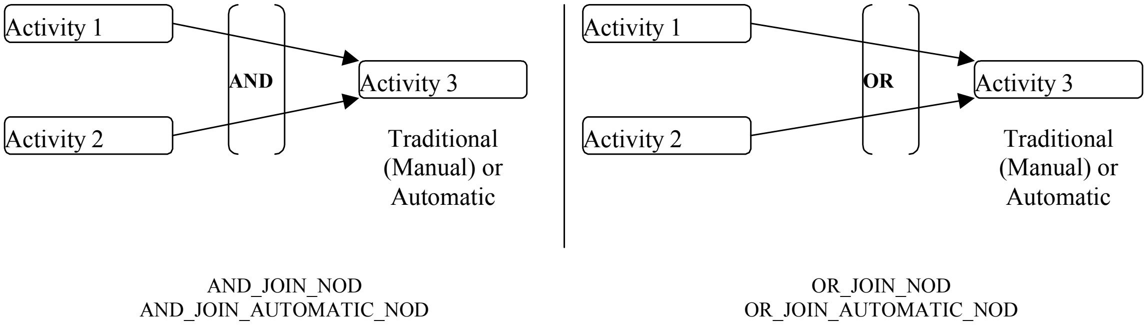


Figure 26 Activity Types

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

Here are the constant values associated with the types:

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

Figure 27 Activity Types constant values

4.5.2 Activities states

See the “Activities basics “section of this document.

The constant values associated with the main activities states 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

Figure 28 Activity State constant values

4.5.3 Creating an activity

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

Adds a node called "name" to the project. This method creates a node with the corresponding node type (Figure 27) and assigns to it a role equal to **InitialRole**. This role is not assigned to any user at creation time, so this activity isn't eligible for use until the setNodeRole method is called.

4.5.4 Creating SubProcess activity

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

Add subProcess node called "name" to the current 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.

4.5.5 Configuring an activity

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

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

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

Set the node in anticipable mode.

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

Set the node in automatic mode. The responsibility of activity execution is now with the Bonita engine.

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

Set an absolute node deadline (ex 11-05-2005). Activity deadline is the latest date when the activity must be finished. **Deprecated.** *Replaced by setNodeDeadlines(String name, Collection co) see below*

`void` **`setNodeRelativeDeadline`**`(String name, long date)`

Set a relative node deadline (ex: 2 hours). Activity deadline is the latest date or time in when the activity must be finished. **Deprecated.** *Replaced by setRelativeDeadlines(String name, Collection co) see below* `void` **`setNodeDeadlines`**`(String name, Collection co)`

Set one or more deadlines for the node. Activity deadline is the latest date the activity must be completed.

`void` **`setNodeRelativeDeadlines`**`(String name, Collection co)`

Set one or more deadlines for the node. Activity deadline is the latest date the activity must be completed.

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

Set the node description. Node description represents explicit execution related information for this task.

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

Set a property of a node for “nodeName”. A property is a pair key/value representing workflow relevant data. This method propagates the property is to other nodes automatically.

`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. The use of the propagate argument specifies whether to propagate this property.

`void` **`setNodePropertyPossibleValues`**`(String nodeName, String key, Collection values)`

Set property possible values for a specific node. The values argument represents acceptable values as possible property values. Key/value must be enumerated type.

`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).

4.5.6 Iterating activities

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

Add a new iteration between two nodes. The intent is to iterate “from” node B “to” node A. The “from” parameter is the name of the first node (node testing a value), the “to” parameter is the name of the node to execute based on the value.

Note: The iteration must be added to the node executing last (“from” or Activity B below). .In the example (Figure 29), activity A is executed, then some activities between A and B take place, and then B is executed. After the processing of B, control goes back to A if the iteration condition set in B evaluates to true.

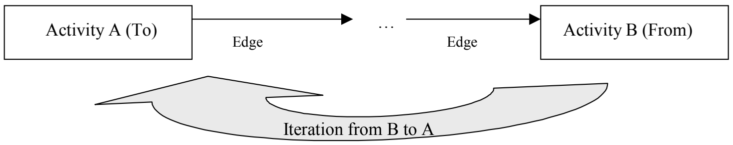


Figure 29 Activity Iteration

The condition may be something like "lastNodeProperty.equals(\"value\")", while the value of the property is evaluated depending on the execution of the process.

Code sample: To see activity iteration, refer to the **Sample1CreateProcessModel** example in the Bonita samples directory. **Sample1RunningSession** executes the first iteration of the process. To terminate execution after a second iteration, modify the value *"once_more"* in the *Receive Order* activity within the *Order Processing* Instance running.

4.5.7 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)

Return the names of all nodes in the project.

4.5.8 Getting information about a specific node

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

Get Node Value from a specific project (for values see Figure 27).

String **getNodeDeadline**(String nodeName)

Return a node deadline. Activity deadline is the latest date or time by which the activity must complete.

String **getNodeDescription**(String name)

Return the node description. Node description represents explicit execution related information of this task.

String **getNodeExecutor**(String name)

Return the node executor. Return the name of the user executing the activity.

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

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

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

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

int **getNodeState**(String name)

Return the state of the node. (See Figure 28 for node states.)

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

Return the type of the node. (See Figure 27.)

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

Return the node Value. See Figure 27 for node values.

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

Return true if the node is ready for execution in anticipated mode.

4.5.9 Deleting activity

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

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

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

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

4.6 Managing Edges

4.6.1 Adding an edge to an activity

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

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

`String` **`addEdge`**`(String in, String out);`

The two activities, named in and out, are connected by a new edge. The method returns the name of the newly created edge. In this case the name is assigned by BONITA.

4.6.2 Deleting an edge

`Void` **`deleteEdge`**`(String name);`

The edge named with the parameter name is deleted.

4.6.3 Getting connected activities from an edge

`String` **`getEdgeInNode`**`(String edgeName) ;`

Retrieve the node name of the inbound node of the parameter “edgeName”.

`String` **`getEdgeOutNode`**`(String edgeName) ;`

Retrieve the node name of the outbound node of the given edgeName.

4.6.4 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 examples of constructs is a valid condition:

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

4.6.5 Getting the condition on an edge

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

4.6.6 Get all existing edges in a project

```
Collection  getEdgesNames()   (String Collection)
```

Returns all existing edges in the project

4.6.7 Get all existing edges for an activity

```
Collection  getNodeInEdges()  (String Collection)
```

Returns all existing inbound edges for a given node

```
Collection  getNodeOutEdges() (String Collection)
```

Returns all existing outbound edges for a given node

4.6.8 Reading an edge as a Java Object

```
hero.interfaces.BnEdgeValue  getEdgeValue (String name);
```

Get the edge value.

4.6.9 Changing the state of an Edge

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

Set the edge state

4.7 Managing Hooks

Hooks are code executed at specific points during an activity life cycle.

Must document in a central place the different possible scripting strategies

Hooks may be coded in a scripting language (i.e. Beanshell), or as a java library (java code).

Hooks may be defined at the project level. These hooks are activated when a project is instantiated or when the project finishes.

Hooks may also be defined at the activity level.These hooks are activated only in the context of the related activity.

The hook interface is divided in two sets (Hooks and InterHooks).

Interactive Hooks/(InterHooks):
Script hooks are called interactive Hooks. Calls relative to interhooks contain “Inter” in their name. Their hook type is hero.hook.Hook.BSINTERACTIVE (See Figure 32.)

Hooks execute upon detection of one of the events in Figure 30. If the hook does not include that method, an exception is raised. This means a “hook” routine may contain multiple methods dealing with the listed events but the hook must specify which event is acted upon.

Node Hooks events:

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
hero.interfaces.Constants.Nd.ONREADY	"onReady";	onReady
hero.interfaces.Constants.Nd.ONDEADLINE	“onDeadLine”;	onDeadline

Figure 30 Node Hook Event constants

Project Hooks events:

EVENT	VALUE	METHOD
hero.interfaces.Constants.Pj.ONINSTANTIATE	"onInstantiate"	onInstantiate
hero.interfaces.Constants.Pj.ONTERMINATE	"onTerminated"	onTerminated

Figure 31 Project Hook Event constants

Different hooks types taken in to account by the Bonita engine:

HOOK TYPE	VALUE
hero.interfaces.Constants.Hook.JAVA	0
hero.interfaces.Constants.Hook.BSINTERACTIVE	6

Figure 32 Hook Type constants

4.7.1 Hook at the project level

Creating

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

Add an existing hook file to the project. This hook type, JAVA, (see Figure 30) references a Java class file loaded at run time. The parameter “hookName” represents the java class file to load by the system at run time. These class files must be located in the application server classpath definition to execute correctly.

Deleting

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

Deletes the hook specified by hookName in current project

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

The hook / interHook specified by “hookName” is deleted from all project nodes.

Managing

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

Returns all the hook names assigned to the project**.**

4.7.2 Hooks associated to a specific activity

Creating Hooks

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

Add an existing hook file to the node (activity). This type of hook uses a Java or TCL file loaded at run time. The parameter “hookName” represents the java class or TCL file loaded by the system at run time. These classes must exist in the application server classpath definition for correct hook execution. Place the hooks classes in $BONITA_HOME\src\resources\hooks and redeploy bonita.ear (ant task).

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

The hook name “hookName” is added to the node. The hook activation is triggered whenever the event “eventName” occurs for this activity. Please see events defined in Figure 30.

Deleting Hooks

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

Delete a node hook. **Deletes a hook defined for this activity. If the hook does not exist, an exception is returned.**

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

Delete a node interHook. The hook or the interHook with name hookName is deleted from the node. **If the hook does not exist, an exception is returned.**

Managing Hooks

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

Return all the Node hooks for the project.

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

Return all of the Interactive Node hooks for the project.

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

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

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

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

4.7.3 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
```

Figure 33 Code sample Managing Hooks

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

4.8 Managing users

4.8.1 Getting the list of all Bonita registered users

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

Return the names of all registered users in the Bonita System.

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

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

Return all users of the project.

4.8.3 Adding a user to a project

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

Add a user to this project (The user must exist in the Bonita database)

4.8.4 Checking whether a user is part of a project

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

Test if the “username” is associated to this project

4.8.5 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") ) ;
```

Figure 34 Code sample Managing Users

4.9 Managing Project roles

A role is the means by which a User is associated to an activity. A role has a name and a description.

First, roles must be declared in a project. Then the role can be associated to Users and Activities.

4.9.1 Declaring a new role in the project

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

Add /creates a role within this project. The role is specific to this project.

4.9.2 Allocating a role to a User

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

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

`Assigns “username” the role specified in “roleName”. If the user is not registered or if role name does not exist an exception is thrown.`

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

`Remove the role specified by “roleName” from the user specified by “username”. If user name or role name does not exist, an exception is thrown.`

4.9.3 Getting a list of roles that an user can assume

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

Return all the roles available for this user (independently of any project). If no roles exist “NULL” is returned.

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

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

Return all roles of the current project. These roles are associated with the nodes included in the project. If no roles exist, “NULL” is returned.

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

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

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

Return the roles of this user in the current project. If no roles are assigned for the user NULL is returned.

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

Return the role names of the user in the current project. If no roles are assigned “NULL” is returned.

4.9.5 Associating an activity with a role

Only a single role can take over a given activity.

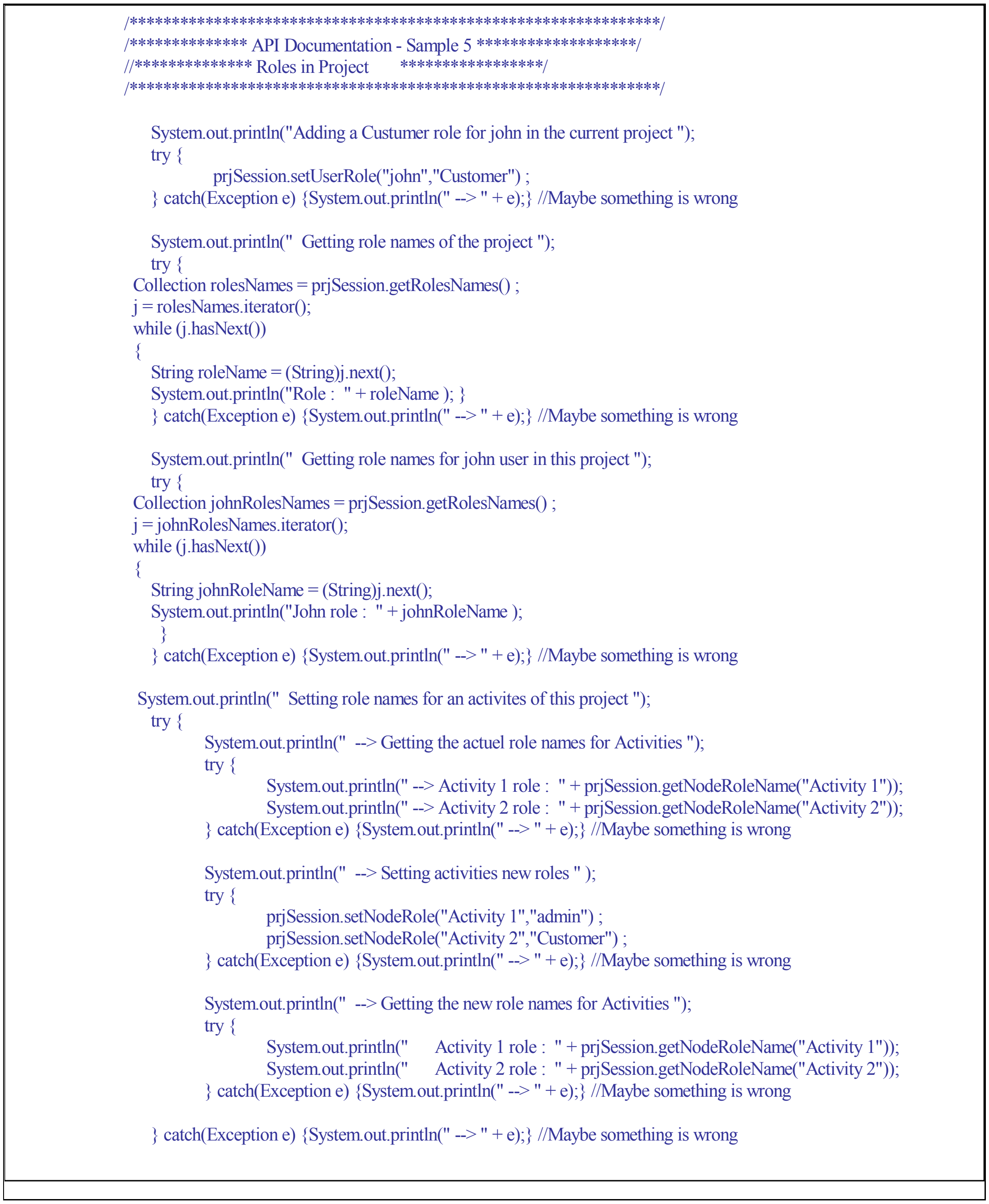
`String` **`getNodeRoleName`**`(String nodeName)`

Obtain the role name of the specified node. If “nodeName” does not exist or has terminated “NULL” is returned.

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

Sets or changes the role of an activity if the role name already exists. If the role name does not exist….. is returned…..

4.9.6 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
```

Figure 35 Code sample Roles

4.10 Mappers

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

Add an existing mapper to the role « roleName ». This type of mapper uses a Java file loaded at run time. If "mapperName" does not exist an exception is thrown. 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`

Figure 36 Mapper Constants

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

Delete a role mapper. If "roleName" does not exist an exception is thrown.

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

Return all the role mappers of the project. If "roleMapper " does not exist, XXXXX is returned.

4.10.1 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);
..../....
```

Figure 37 Code sample Mappers

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

4.11 Performer assignment

4.11.1 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 uses a Java file loaded at run time. If "performerAssignName" does not exist an exception is thrown.

PerformerAssignType can be one of the following:

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

Figure 38 Performer Assign Types

4.11.2 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");
```

Figure 39 Code sample Performer

4.12 Model definition ccheck

`void checkModelDefinition()`

This functionality was added in Bonita v2. This method checks that the model is defined correctly. It must be called at the end of process model definition.

Presently only iteration guidelines (explained in chapter 1.3.3 'Iterating activities') are verified, but in future versions this method may include other model definition verification.

4.12.1 checkModelDefinition method Verification

The next two examples/figures explain the *checkModelDefitinion()* method verification.verification.

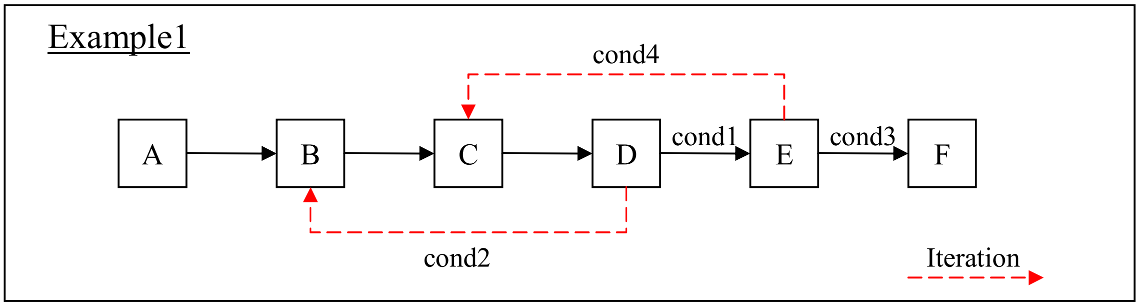


Figure 40 Check Model example 1

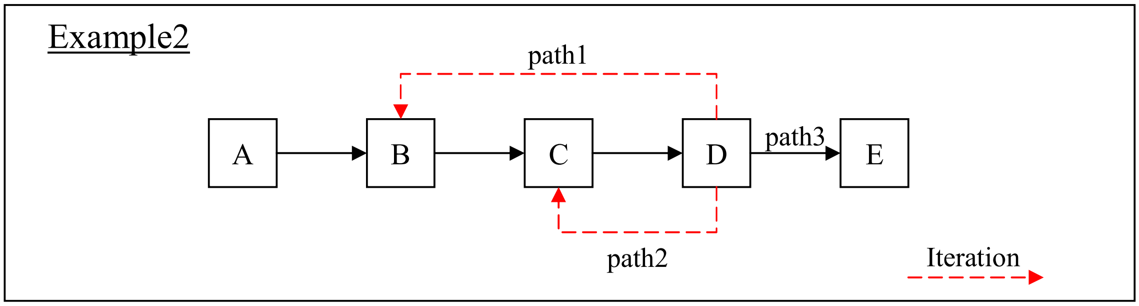


Figure 41 Check Model example 2

- checkIteration method:
  - Checks that the iteration's conditions are not empty
    In Figure 40 it is not possible to have a null value: i.e. cond2.equals("")). In that case a HeroException is thrown.
    NOTE:the value "true" is NOT allowed as this creates a condition producing an infinite loop.

- Checks if a path between from → to activities exists and is defined in the iteration. This process repeated to guarantee that the model is well defined (some transitions could have been removed).

- If multiple iterations exist in the same node, then a check verifies that the iteration conditions are different.
  In Figure 41: path1 condition must be different from path2 condition.

- checkMandatoryIterationConditions():

  - Verifies thatthat mandatory conditions on the out edges of nodes creatingcreatingan exit point from the iteration
    In Figure 40: cond1 and cond3 have to be set.
    · throws a HeroException if these edges don't specify a condition
    · throws a HeroException if the condition is empty (NULL): cond1.equals("")
    · throws a HeroException if the condition value is "true": cond1.equals("true")

  - Verifies that mandatory conditions on the out edges for the node starting the iteration differ from the iteration's condition.
    In Figure 40: cond1 must be different from cond2 and cond3 must be different from cond4.
    ·Throws a HeroException if the iteration's starting condition and out edge's condition are equal.

4.12.2 Code sample

```
.../...
ProjectSessionHome prjHome = (ProjectSessionHome) ProjectSessionUtil.getHome();
ProjectSession prjSession = prjHome.create();
prjSession.initModel("DoubleIteration");
try {
    // Activities creation
    prjSession.addNode("A", hero.interfaces.Constants.Nd.AND_JOIN_NODE);
    prjSession.addNode("B", hero.interfaces.Constants.Nd.AND_JOIN_AUTOMATIC_NODE);
    prjSession.addNode("C", hero.interfaces.Constants.Nd.AND_JOIN_AUTOMATIC_NODE);
    prjSession.addNode("D", hero.interfaces.Constants.Nd.AND_JOIN_AUTOMATIC_NODE);
    prjSession.addNode("E", hero.interfaces.Constants.Nd.AND_JOIN_AUTOMATIC_NODE);
    prjSession.addNode("F", hero.interfaces.Constants.Nd.AND_JOIN_AUTOMATIC_NODE);

    // Setting Activities types
    prjSession.setNodeTraditional("A");
    prjSession.setNodeTraditional("B");
    prjSession.setNodeTraditional("C");
    prjSession.setNodeTraditional("D");
    prjSession.setNodeTraditional("E");
    prjSession.setNodeTraditional("F");

    // Adding project properties
    prjSession.setProperty("condition1", "50");    // % to do 1st iteration
    prjSession.setProperty("condition2", "50");    // % to do 2nd iteration
    prjSession.setProperty("randomNum", "0");      // random number who decides
                                                   //    if we iterate or not
    // Adding edges between activities
    prjSession.addEdge("A", "B");
    prjSession.addEdge("B", "C");
    prjSession.addEdge("C", "D");
    String fromDtoE = prjSession.addEdge("D", "E");  // Exit conditions from iterations
    String fromEtoF = prjSession.addEdge("E", "F");

    // Adding 'D' & 'E' edge conditions
    prjSession.setEdgeCondition(fromDtoE,
          "(new Integer(randomNum).intValue() >= new Integer(condition1).intValue())");
    prjSession.setEdgeCondition(fromEtoF,
          "(new Integer(randomNum).intValue() >= new Integer(condition2).intValue())");

    // Adding D & E hooks (that generate random values saved in randomNum property)
    .../...

    // Adding iterations: between D--->B and E--->C
    prjSession.addIteration("D", "B",
          "(new Integer(randomNum).intValue() < new Integer(condition1).intValue())");
    prjSession.addIteration("E", "C",
          "(new Integer(randomNum).intValue() < new Integer(condition2).intValue())");

    // Check model definition
    prjSession.checkModelDefinition();
}
```

Figure 42 Code sample Check Model Definition

5 USER SESSION INTERFACE

5.1 Principle

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

In cases of the EJB Session access, the User interface automatically retrieves the identity of the calling user in the J2EE security context. Therefore, calling the User interface from an unidentified user context fails.

Many of the User interface methods require the Project name as a parameter. This name may be retrieved by the application logic. Alternatively, the application may retrieve the project name using various search criteria.

NOTE: At this time, the corresponding search methods are not implemented.

The UserSessionBean, is an stateful session bean providing user API methods for obtaining information on user Todo lists and started activities. Also, the UserSessionBean may be used to produce activity events (i.e. start, terminate, cancel).
The UserSessionBean is based on the Bonita Engine Session Bean: that is, a recursive implementation that manages previous execution operations and propagates the activity state changes to activities connected to this one.

The UserSessionBean API provides information about user projects and activities (i.e. project list, todo list, and activity list). The UserSessionBean may also be used to obtain useful information about project instances or user preferences. With this API users can perform task/activities using start, terminate, and cancel methods. The user may also terminate workflow processes.

Coding examples using the User Session interface API are shown below in Figure 43.

These below examples are extracts from the SampleUserApi.java file located in the "samples" directory of Bonita. This example may be executed by using "ant sample-user-api" in the $BONITA_HOME directory.
Also refer to the sample1xxx classes implementing the user guide workflow example (Order Processing and Customer Service).
Execute the sample1xxx examples by using "ant sample1-create-process-model" (Model creation), "ant sample1-admin-wf" (user administration and project instantiation), "ant sample1-running-session" (Process execution).

5.2 Creating the UserSessionBean

The UserSessionBean is seen as a connection handle into the BONITA workflow System. After user authentication, this handle must be created with the user identity.
Every subsequent call to the User Session API functions are 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();
```

Figure 43 Code sample creating UserSession

5.3 User Properties

5.3.1 Setting User Properties

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

Set the property using *key* name to the value *value*.

If the property already exists, the current value is overridden. If the property 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 the Bonita database.

5.3.2 Getting User Information

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

Return the name of the current authenticated User.

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

Return the user password

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

Return the mail address for this user from Bonita database. If “username” does not exist, an exception is thrown.

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

Return the properties defined for the current authenticated User. If no properties exist, null is returned.

5.4 User and Projects

5.4.1 Getting the list of projects for the User

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

Return the Workflow processes associated to this user.

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

Return project list names for this user.

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

Return Workflow projects associated with a property.

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

Return Workflow projects associated with a property.

5.4.2 Getting the list of instances for the User

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

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

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

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

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

Return Workflow instances of this project. If projectName does not exist, an exception is thrown.

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

Returns workflow instances names of this project.

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

Return Workflow instances from a property.

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

Return a list of project instances from a property.

5.4.3 Managing the project for the User

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

Delete a Workflow project

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

Attempts to terminate a project (termination occurs when all project activities are terminated)

5.5 User and Activities

5.5.1 Getting the list of activities for the User

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

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

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

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

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

Obtain executing user activities matching the property value (executing and anticipating state activities).

5.5.2 Getting Information on User activity

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

Return Node Value for a specific project.

5.5.3 Getting the ToDo list for the User

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

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

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

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

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

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

5.5.4 Managing activities for the User

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

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

`void` **`terminateActivity`**`(String projectName, String nodeName)`
Attempts to terminate an activity (when activity state is executing or anticipating)

`void` **`cancelActivity`**`(String projectName, String nodeName)`
Attempts to cancel an activity (when activity is executing or anticipating)

5.6 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
```

Figure 44 Code sample Managing User Activities

6 BONITA ENTITIES

Many entry points in the API allow retrieving data about the process entities, such as the relevant information 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 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 entities managed at the API level.

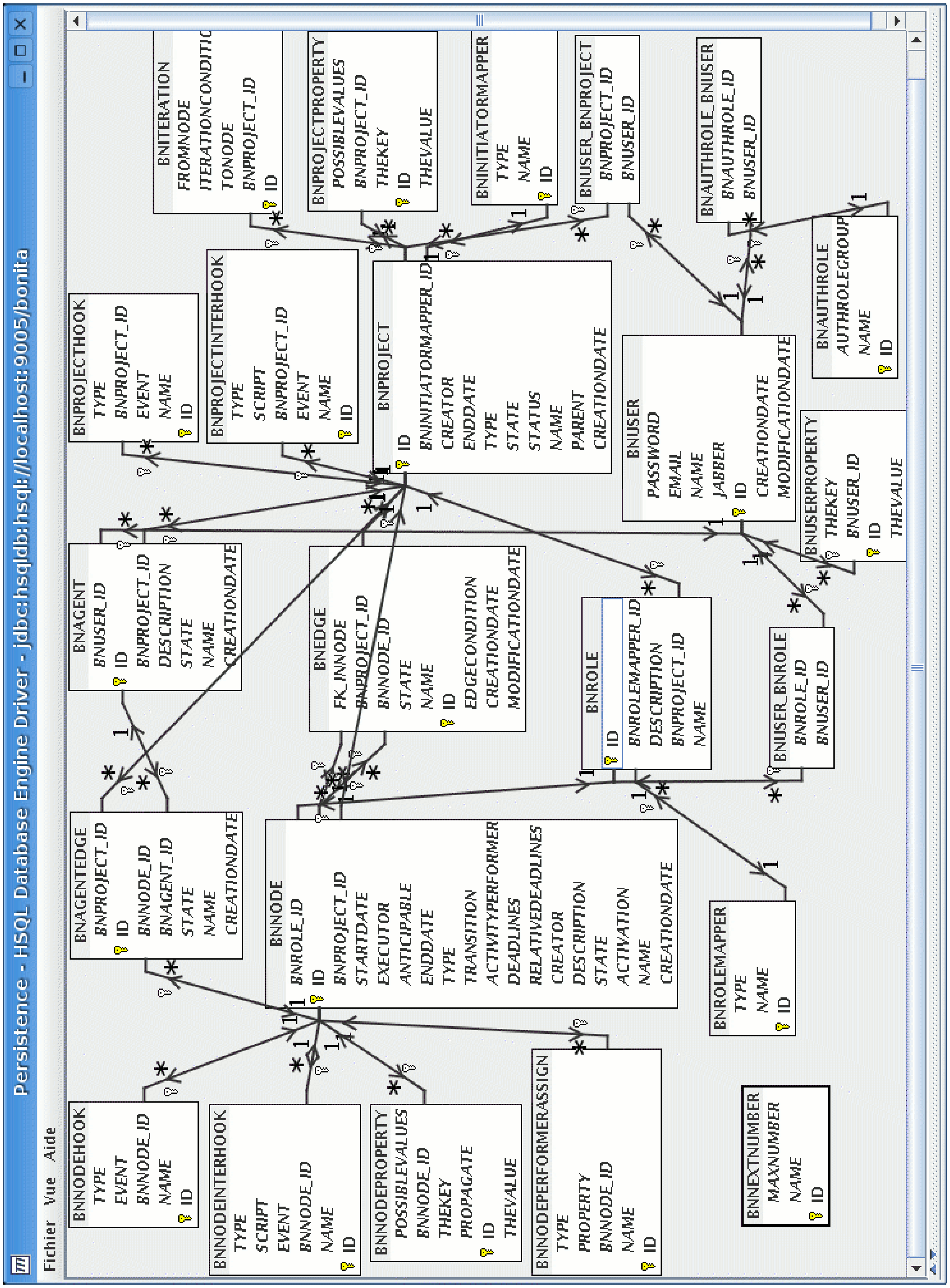
If *Entity* is the name of the internally used Enterprise Java Bean, *EntityValue* is the name of the corresponding plain old java object, *EntityLightValue* is the name of a simpler java object (very often, *EntityLightValue* has only fields that have a simple type).

If you want to directly use the internal EJB thru the remote or local interfaces (We do not recommend this choice), each of these entities may be accessed using its name suffixed by hero.interfaces.

6.1 Entities diagrams

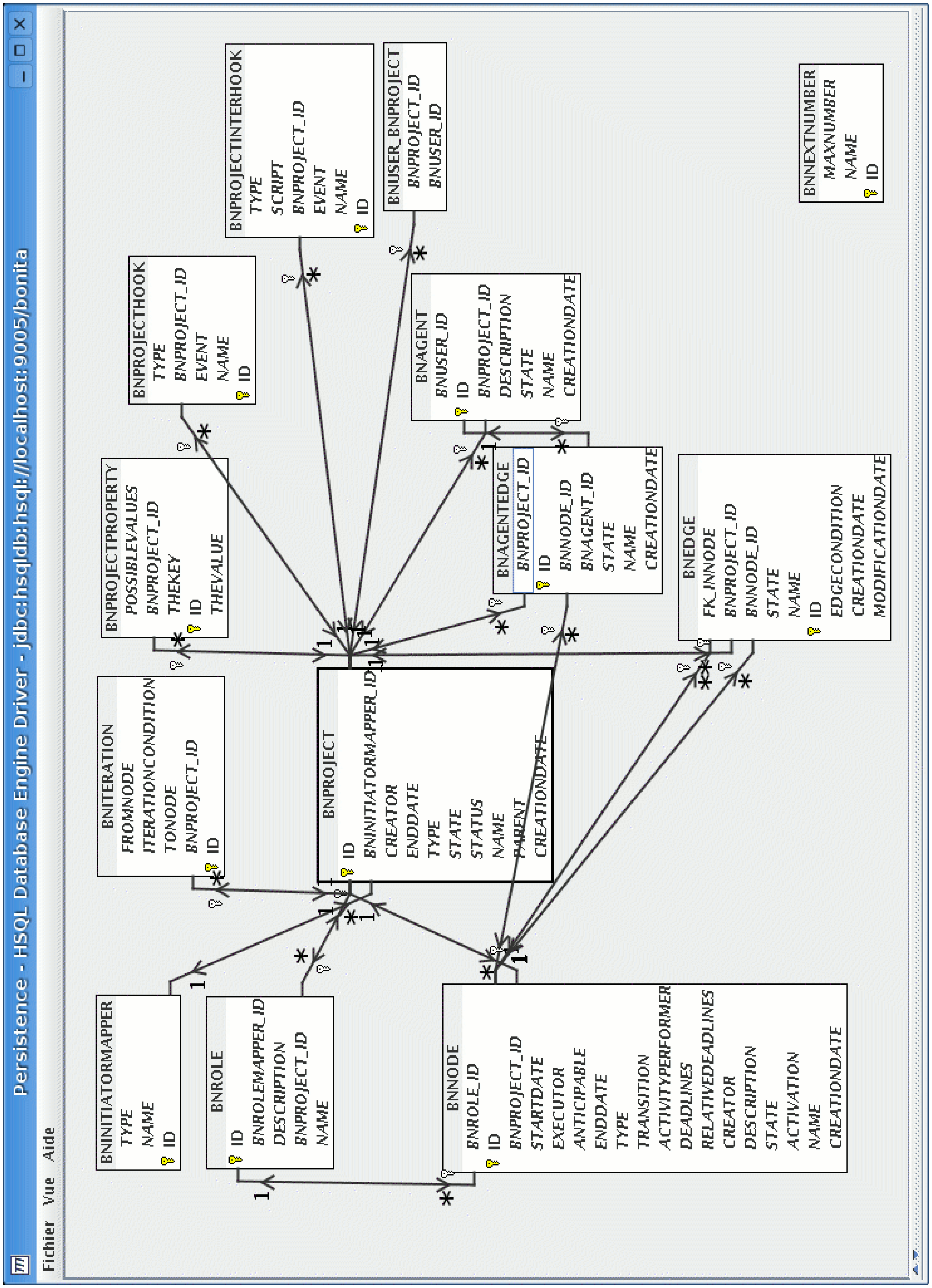
6.1.1 Global diagram

Figure 45 Entity Global Diagram


Persistence - HSQL Database Engine Driver - jdbc:hsqldb:hsql://localhost:9005/bonita
Fichier Vue Aide
BNITERATION FROMNODE ITERATIONCONDITIO TONODE BNPROJECT_ID ID
BNPROJECTPROPERTY POSSIBLEVALUES BNPROJECT_ID THEKEY ID THEVALUE
BNINITIATORMAPPER TYPE NAME ID
BNUSER_BNPROJECT BNPROJECT_ID BNUSER_ID
BNAUTHROLE_BNUSER BNAUTHROLE_ID BNUSER_ID
BNPROJECTHOOK TYPE BNPROJECT_ID EVENT NAME ID
BNPROJECTINTERHOOK TYPE SCRIPT BNPROJECT_ID EVENT NAME ID
BNPROJECT ID BNINITIATORMAPPER_ID CREATOR ENDDATE TYPE STATE STATUS NAME PARENT CREATIONDATE
BNUSER PASSWORD EMAIL NAME JABBER ID CREATIONDATE MODIFICATIONDATE
BNAUTHROLE AUTHROLEGROUP NAME ID
BNUSERPROPERTY THEKEY BNUSER_ID ID THEVALUE
BNAGENT BNUSER_ID ID BNPROJECT_ID DESCRIPTION STATE NAME CREATIONDATE
BNEDGE FK_INNODE BNPROJECT_ID BNNODE_ID STATE NAME ID EDGECONDITION CREATIONDATE MODIFICATIONDATE
BNROLE ID BNROLEMAPPER_ID DESCRIPTION BNPROJECT_ID NAME
BNUSER_BNROLE BNROLE_ID BNUSER_ID
BNAGENTEDGE BNPROJECT_ID ID BNNODE_ID BNAGENT_ID STATE NAME CREATIONDATE
BNNODE BNROLE_ID ID BNPROJECT_ID STARTDATE EXECUTOR ANTICIPABLE ENDDATE TYPE TRANSITION ACTIVITYPERFORMER DEADLINES RELATIVEDEADLINES CREATOR DESCRIPTION STATE ACTIVATION NAME CREATIONDATE
BNROLEMAPPER TYPE NAME ID
BNNODEHOOK TYPE EVENT BNNODE_ID NAME ID
BNNODEINTERHOOK TYPE SCRIPT EVENT BNNODE_ID NAME ID
BNNODEPROPERTY POSSIBLEVALUES BNNODE_ID THEKEY PROPAGATE ID THEVALUE
BNNODEPERFORMERASSIGN TYPE PROPERTY BNNODE_ID NAME ID
BNNEXTNUMBER MAXNUMBER NAME ID
1
*

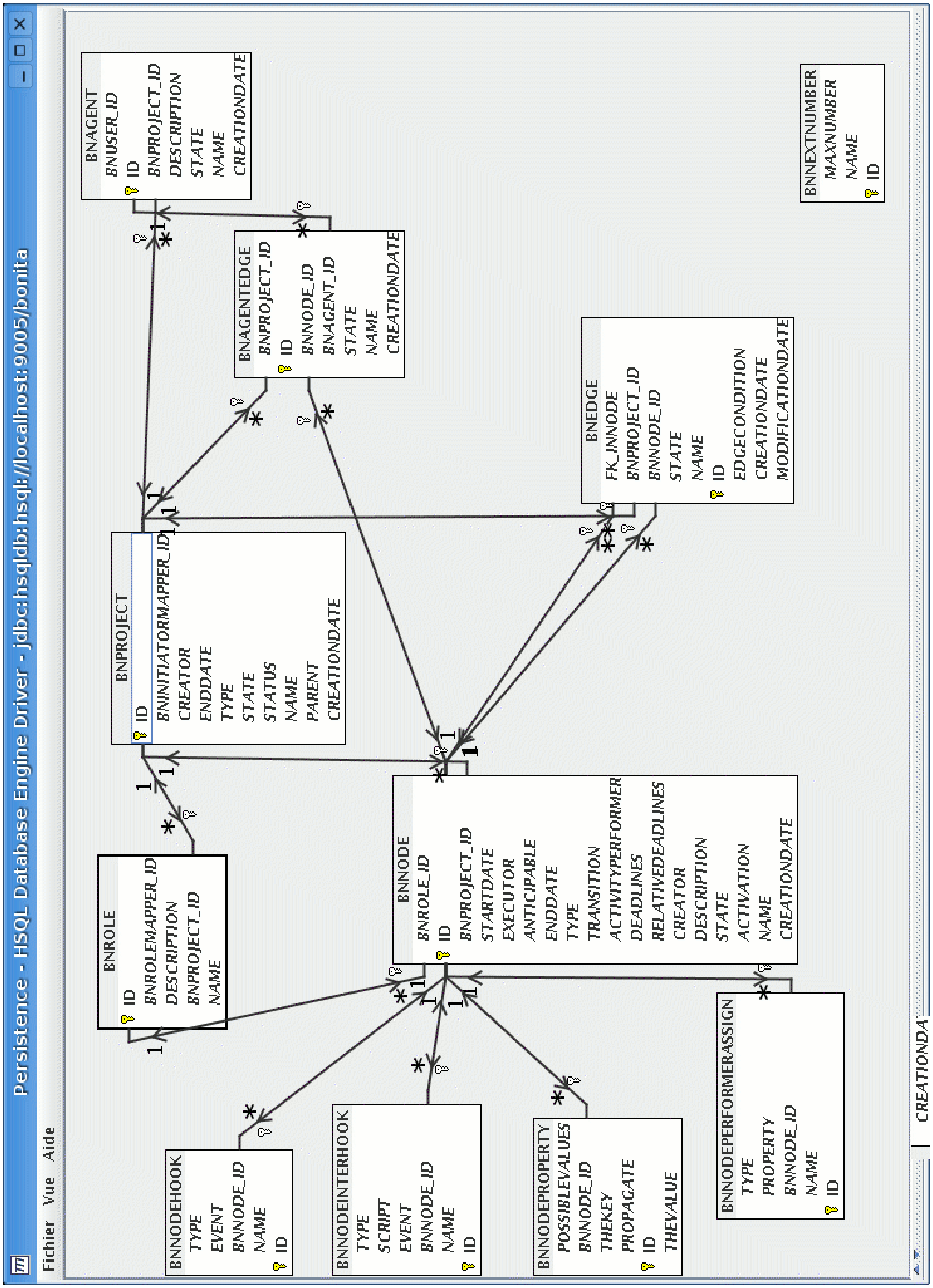
6.1.2 Diagram focused on Project entity relations

Figure 46 Entity relations



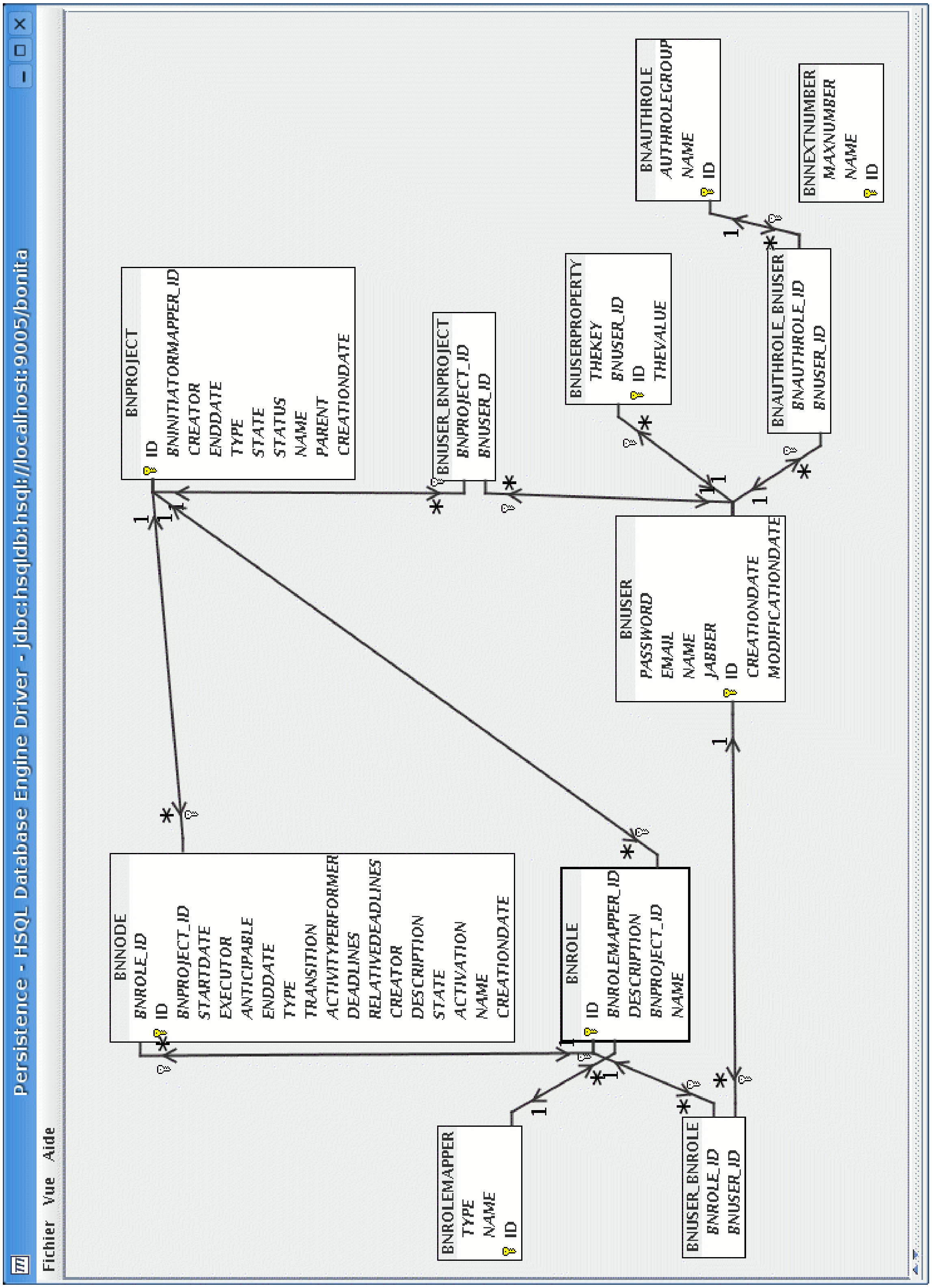
6.1.3 Diagram focused on Node entity relations

Figure 47 Node entity relations



6.1.4 Diagram focused on User-Role entities relations

Figure 48 User-Role entity relations



6.2 Entities Attributes

6.2.1 BnAuthRoleValue

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

Figure 49 BnAuthRole Values

6.2.2 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;**	

Figure 50 BnEdge Values

6.2.3 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;**	

Figure 51 BnInstance values

6.2.4 BnIterationValue

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

Figure 52 BnIteration values

6.2.5 BnNodeHookValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	

String	**name;**	
boolean	**nameHasBeenSet**	
String	**event;**	
boolean	**eventHasBeenSet**	
int	**type;**	
boolean	**typeHasBeenSet**	
hero.interfaces.BnNodeHookPK	**pk**	

Figure 53 BnNodeHook values

6.2.6 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**	

Figure 54 BnNodeInterHook values

6.2.7 BnNodePerformerAssignValue

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

Figure 55 BnNodePerformer values

6.2.8 BnNodePropertyValue

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

Figure 56 BnProperty values

6.2.9 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.BnNodePK	**pk;**	

Figure 57 BnNode values

6.2.10 BnProjectHookValue

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

Figure 58 BnProjectHook values

6.2.11 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;	

Figure 59 BnProjectInterHook values

6.2.12 BnProjectPropertyValue

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

Figure 60 BnProjectProperty values

6.2.13 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;	

Figure 61 BnProject values

6.2.14 BnRoleMapperValue

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

Figure 62 BnRoleMapper values

6.2.15 BnRoleValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
String	description;	
boolean	descriptionHasBeenSet	
String	name;	
boolean	nameHasBeenSet	
hero.interfaces.BnRoleMapperValue	BnRoleMapper;	
boolean	BnRoleMapperHasBeenSet	
hero.interfaces.BnRolePK	pk;	

Figure 63 BnRole values

6.2.16 BnUserPropertyValue

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

Figure 64 BnUserProperty values

6.2.17 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.BnUserPK	pk;	

Figure 65 BnUser values