Bonita Workflow

Bonita API

BONITA WORKFLOW

Bonita Workflow

Bonita API

Bonita Workflow v3.0

Software

January 2007

Table of Contents

List of Figures

List of Tables

Preface

This document explains the use of Bonita Workflow API functions. Using the information within document assists a Bonita Workflow designer in the implementation of Workflow projects, models, and administration using Bonita API functions.

Chapter 1.Introduction

Bonita is a workflow system with innovative features such as;

- Activities that can start in anticipation,
- Awareness infrastructure – this allows user notification of any events during the execution of a given process,
- Automatic activation of user's code according to a defined activity life cycle.

Traditional workflow features, such as dynamic user/roles resolution, activity performer, and sequential execution, are also included in Bonita supporting 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 through either project and/or user EJB's.

Bonita processes are created using either a graphical definition tool or by using the Bonita Project interface API. A Bonita process is defined as a set of activities and an associated execution model. The Bonita enactment engine maintains activity scheduling based on the defined execution model. The Bonita User API provides full control over process execution: for example, starting or stopping of an activity. Bonita also supports dynamic modification of an existing process, that is, the Bonita Project interface API can be applied to a running process.

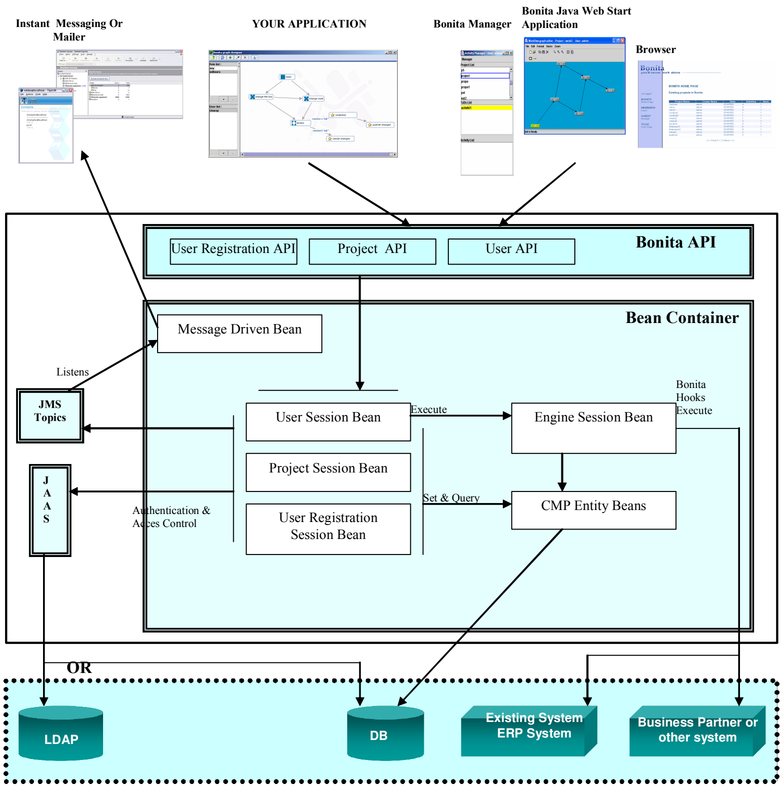


Figure 1-1. Bonita Workflow Diagram

- **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
    - Activity execution
    - 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 via 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 a Session Bean, or as web services.

Chapter 2.Concepts

2.1 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 or specific 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.

2.2 Process

2.2.1 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 either by definition or at runtime by the workflow process administrator(s). Three possible statuses are allowed for a workflow process:

- **Active**: the workflow process can be modified or executed. Active 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.

- **Undeployed**: as for hidden processes, the undeployed status means that the process is not yet available. However, the restrictions are quite different: only the instantiation operation is not allowed. This status is suitable when dealing with different versions of a same model.

2.2.2 Life Cycle

Bonita has a very simple process life cycle and goes through the following states:

- Once it is created, a process is in **initial** state. As soon as the process is in the initial state, it can be controlled using either User API and/or Project API requests. The User API allows monitoring of process execution. When the first activity starts, 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.
- When the first activity begins execution, a process is **started** (enters started state). While executing, the process definition may be modified using the Project API. When all activities terminate, the process remains in the **started** state and the process is still modifiable. For example, new activities could be added to the process.
- A process is **terminated (enters terminated state)** once it has been explicitly terminated by an application through the User API. In the **terminated** state, the process definition cannot be modified.

2.2.3 Cooperative Processes

Bonita has a simple view of cooperative process enactment: once a process is defined, it is enacted. For example; using the Project API we could create a process with a single activity, and then be able to execute it using the User API and have the ability to dynamically add new activities to the process definition. This brings flexibility to workflow participants, and is particularly convenient for cooperative (ad hoc) processes.

Typically, you would set up a specific process to perform a given job between several colleagues. To allow some level of reuse of a process definition, we introduce the concept of **process clone** (see section 2.2.5 clone processes).

2.2.4 Models & Instances

There are scenarios where the reuse of a 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 multiple times. These processes are based on a model-instance workflow paradigm. In this kind of workflow process, 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 accomplish the following; obtain their ToDo list, execute assigned activities, or other workflow user functions.

A process model keeps track of all its instances. That is, all instances of this process are retrievable through the User API functions.

Cooperative or administrative workflows use the same component (i.e. Project) API. Depending on the type of the process created/initialized, this API must also be initialized. There are also differences between the above workflow types concerning process execution. Cooperative workflows are ready for execution and modification from creation. On the other hand, administrative workflows must be instantiated first. The term process model refers to Bonita projects defined in the context of administrative workflow use.

In future releases of Bonita Workflow, the concept of the Process model will be extended with the 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” the whole process model (activities, properties, edges, hooks…) as a new clone of the project in a new process instance. This duplication took a long time even for medium workflow processes, and was a problem for users at instantiation.

Newer Bonita versions (1.4 and later) are revamped to improve performance. Only those activities in the Ready state (including their properties, roles, and existing users) are copied at the creation of the new instance. Once an activity starts, hooks are executed under the Model Hooks (which are not copied). Then, after activity termination, edges and Ready or Executable follow on activities are copied as well.

Notes:

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 conflict with the model definition applied at execution time.

2.2.5 Versioning

There are scenarios where multiple versions of the same workflow model are required. In general this requirement is due to some improvements/changes to be added to an existing workflow model. The versioning feature allows workflow administrators to control the migration of a deployed workflow model when this model needs to be modified or improved.

For information about how to use versioned models, refer to Chapter 5, "Project Interface."

To maintain compatibility with Bonita versions prior to v2.1, the engine will automatically manage models without a particular version as models under 1.0 version.

2.2.6 Clone Processes

A process clone is a duplicate of an existing process. Once the cloning operation is complete, 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 same role defined in the model. All activities are in an initial state, and have the same properties as defined in the model, with the same associated value(s). 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 defined in the process model, with the same initial value(s).
- 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 defined in the process model.

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

2.2.7 Concept of Hooks

Hooks are user-defined logic that can be triggered at some defined point in the process life cycle. The types of Hooks are:

- **OnInstantiate hook** is called before the workflow instance is created. The OnInstantiate hook is not considered to be in the same transaction as the process instantiation action.
- **OnTerminate hook** is called automatically after workflow instance termination ends.

2.2.8 SubProcesses

Sometimes, an independently existing business process can take part in another more sophisticated process. Instead of redefining the activities, edges, properties, and hooks in the parent process, the independent process could be included as a “subProcess” within a specific node.

As the execution logic is inside the subProcess, the subProcess activities are started and terminated automatically by the Workflow 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:

Instantiating a Process with a subProcess activity causes new instance of the parent process to be created. The Bonita engine will identify the subProcess instance by means of an identifier.

As with any other activity, the subProcess activity can be iterated.

Constraints:

As in a normal process, activities, properties, and hooks in the sub-process must not have the same name as another activity, property, or hook 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 in the following figure:

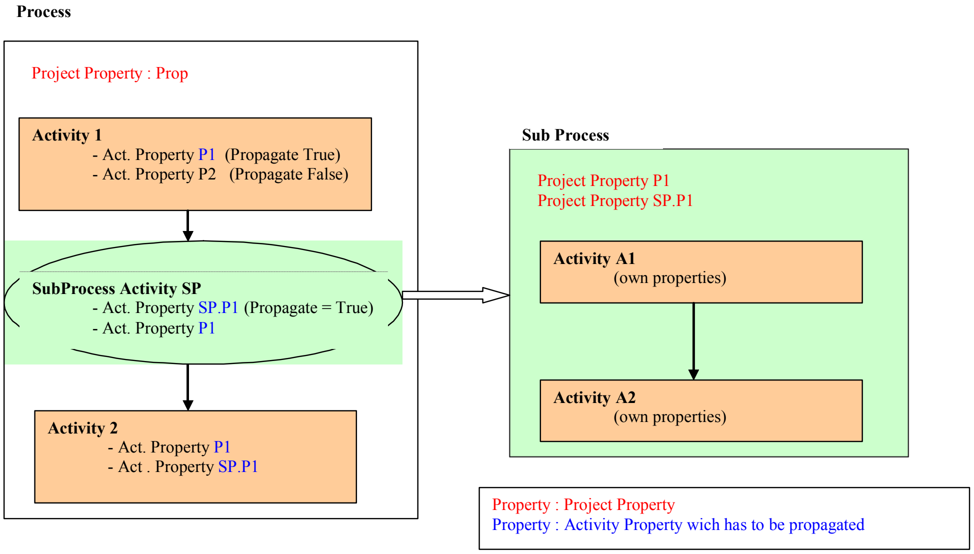


Figure 2-1. SubProcess Diagram

2.2.9 Relationship to Users

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

- 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).

2.3 Activities

2.3.1 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:

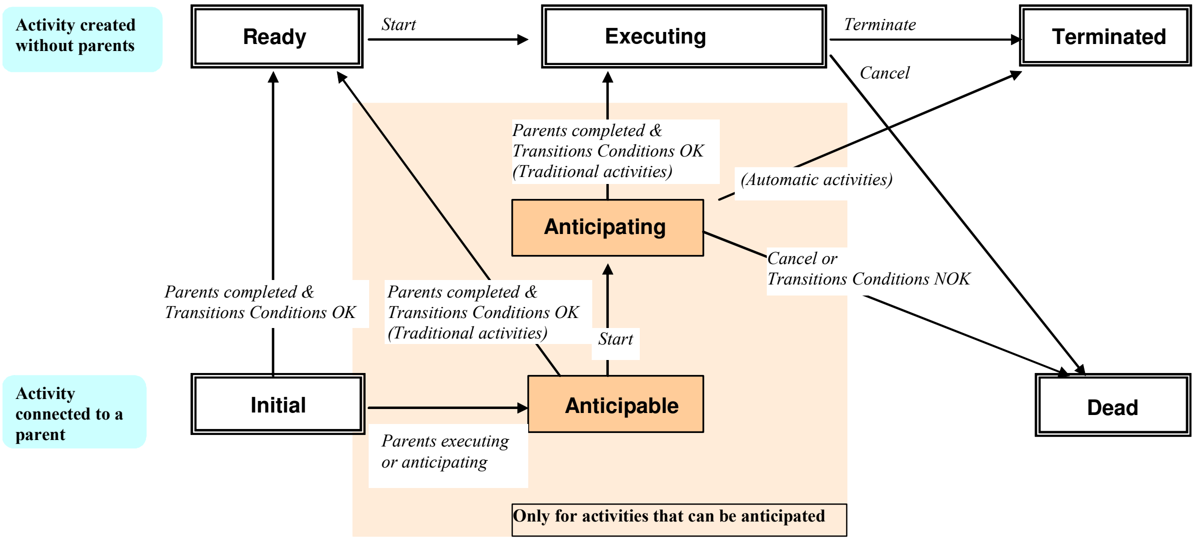

Activity created without parents
Ready
Start
Executing
Terminate
Terminated
Cancel
Parents completed & Transitions Conditions OK (Traditional activities)
(Automatic activities)
Anticipating
Cancel or Transitions Conditions NOK
Parents completed & Transitions Conditions OK
Parents completed & Transitions 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 2-2. 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 conditions to the activity have 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.
- **Anticipatable:** This is the state of an activity that can be started without waiting for its parent activity(s) to complete. However, all of the parent activities must have started.

Note:

A brief explanation of anticipatable and traditional modes. If we have two activities, A and B, with A as the parent activity, the anticipatable mode means an activity (B) may start before its parent activities (A) have completed. In the traditional mode, activity B must wait for activity A to complete before starting.

- **Anticipating:** A previously anticipatable activity that has been started. Automatic activities are automatically transitioned from anticipatable 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:** A 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 anticipatable activities*) - Transition the state from ready to terminated, (For *anticipatable activities*) - Transition the state from anticipatable to anticipating,
- (For *anticipatable activities*) - Transition the state from anticipating to terminated state whenever all parent activities complete.
- Execute any hooks
- Terminate the activity when the executing hooks complete

For activities involving a sub process, the life cycle is described below:

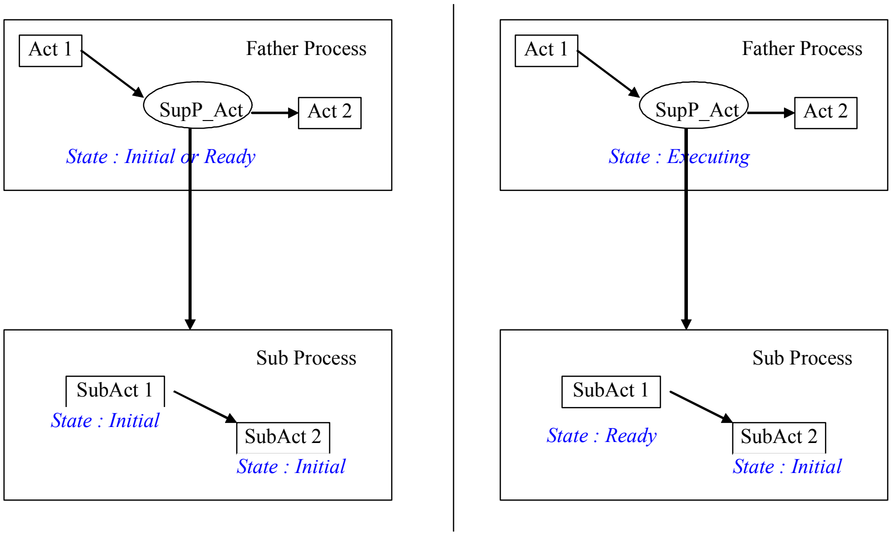


Figure 2-3. Activity Life Cycles with SubProcess

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

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

2.3.2 Transition between Activities

Most of the usual transition patterns can be achieved using Bonita Workflow. There is no special node to achieve these patterns; rather, any activity can behave 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 called the SPLIT construct (allowing 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.

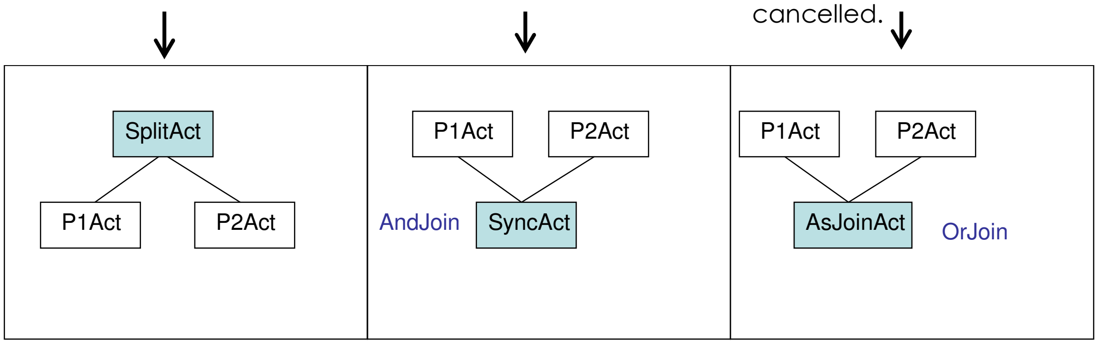


Figure 2-4. Activity Patterns

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)
```

2.3.3 Iterating Activities

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

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.

The following figure is an example of a simple loop.

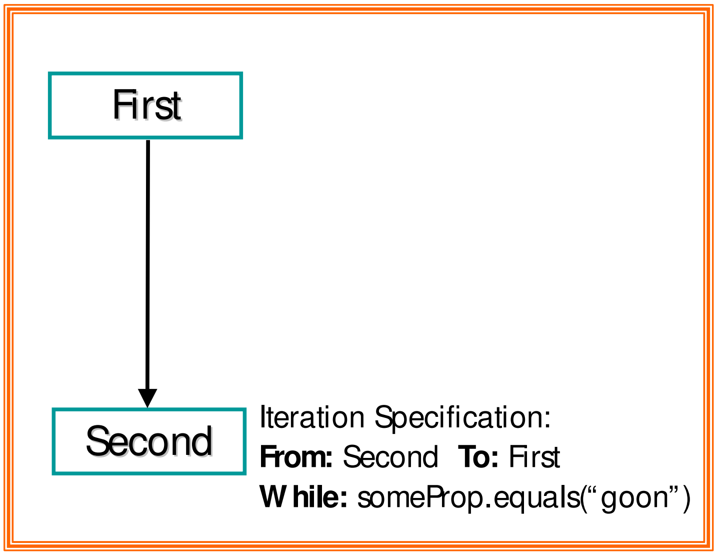

First
Second
Iteration Specification:
From: Second **To:** First
While: someProp.equals("goon")

Figure 2-5. Simple Iteration Loop

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 a project property).

The following is an example of a more complex iteration loop.

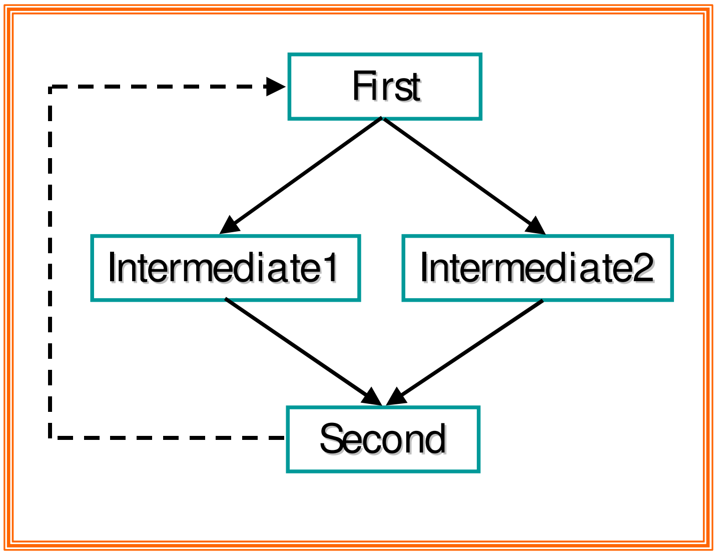


Figure 2-6.   Complex Iteration

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

Iteration in Bonita V2

Iteration behavior is modified in Bonita v2.

Old/previous 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)

The 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.
> **Example:** it is possible to exit from Intermediate2 to anOtherExitPoint or to exit from second to oneExitPoint.

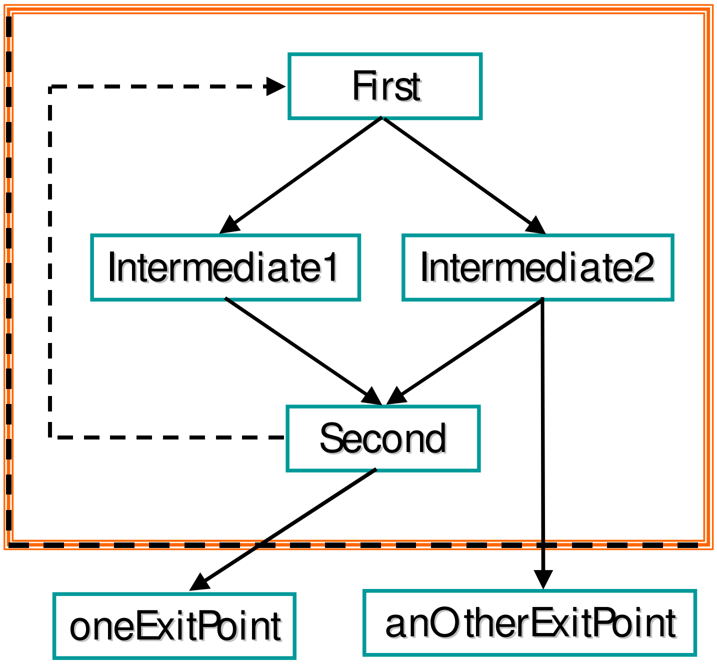


Figure 2-7. Iteration with Multiple Exit Points

Note:

With this new behavior it is not possible to iterate and leave the iteration at the same time.

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.

1. Only one iteration is allowed between two connecting nodes
2. It is possible to have more than one iteration starting in the same node
3. 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.
4. 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 guidelines 3 and 4 are not followed, errors may occur 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 guidelines are validated using this method. This method should be called at the end of a process definition class. For more information see Chapter 0, “Checking Model Definition.”

The following figure shows a comparison between the old iteration model and the new iteration model.

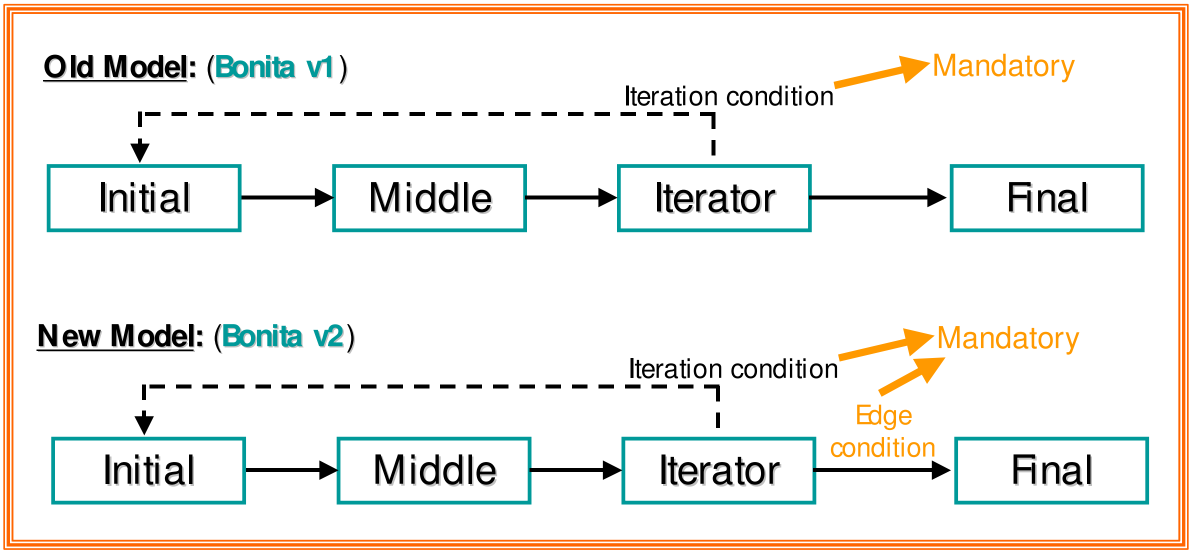


Figure 2-8. Comparison between Old and New Iteration Models

The following *Guidelines* apply.

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

1. Only one iteration is allowed between two connecting nodes
2. It is possible to have more than one iteration starting in the same node
3. 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.)
There could be another iteration starting in the *Iterator* activity going to *Middle* or to *Iterator* itself.
Iterations from *Iterator* to *Final* activity are not allowed because a cycle does not exist.
An edge condition from *Iterator* to *Final* activity is strictly necessary and must be the opposite of the iteration condition. If there are multiple edges outgoing from *Iterator* to other activities, all of them must meet a condition not equal to the iteration condition (this is necessary to accomplish the above premise).

The following figure shows a comparison between the old iteration model and the new model with multiple exit points within the iteration:

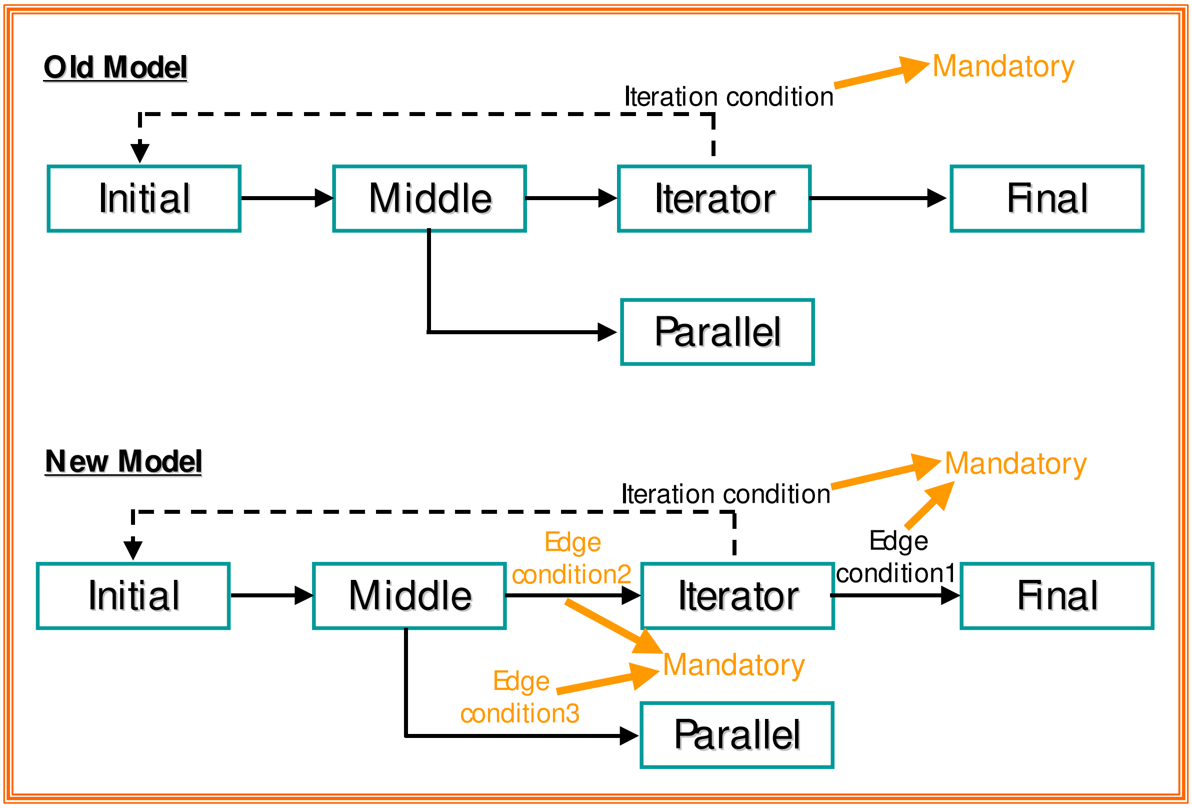


Figure 2-9. Comparison of Old and New Models with Multiple Exit Points

Guidelines applied:

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 in Figure 2-10:

Because *second* is an AND activity, it starts only when activity "*anotherEntryPoint*" has terminated. This is only true for the first occurrence of *second*: for subsequent executions of this iteration, the incoming transitions from *anotherEntryPoint* are ignored.

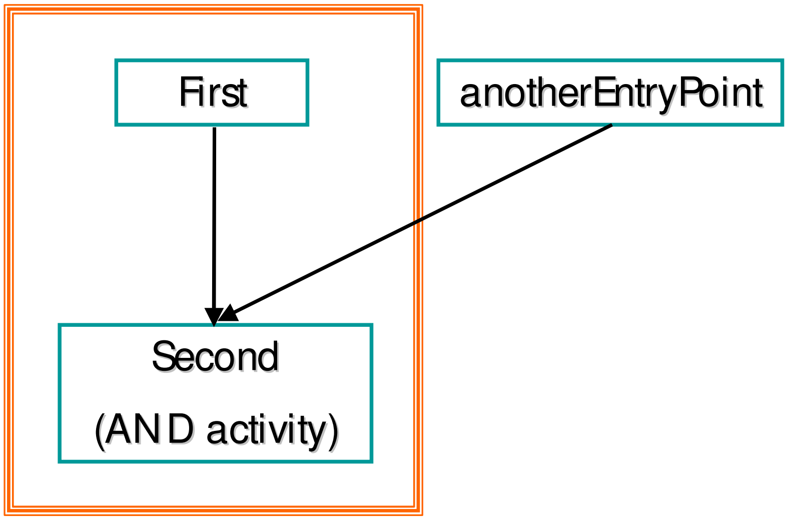


Figure 2-10. Iteration with Multiple Entry Points

2.3.4 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 automatic activities are used:

- 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 previous examples show two error scenarios related to transactional hooks execution.

Important:

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 the following section, "Activity/Hooks and Transactions").

- 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.

Important:

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 to 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" is started and terminates 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.

2.3.5 Activity/Hooks and Transactions

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

This 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. In this manner, Bonita keeps transactional consistency across activities.

Bonita aborts a 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.

2.3.6 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 may 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 (UNIX)
  %BONITA_HOME%/src/resources/hooks/hero/hook (Windows)
- 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. Also, each hook must define the META data return string for example:

```
public String getMetadata() {
        return Constants.Nd.XXXXXX;
```

Where XXXXXX is a value specified in Table 6-1, Node Hooks Events Constants.

Notes:

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

2.4 User Interface

Bonita makes a 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 (also 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 defined (key, value) pairs where both key and value are String variables. The application can set and retrieve properties via the User API interface. Bonita makes use of specific user properties in order to store the User preferences.

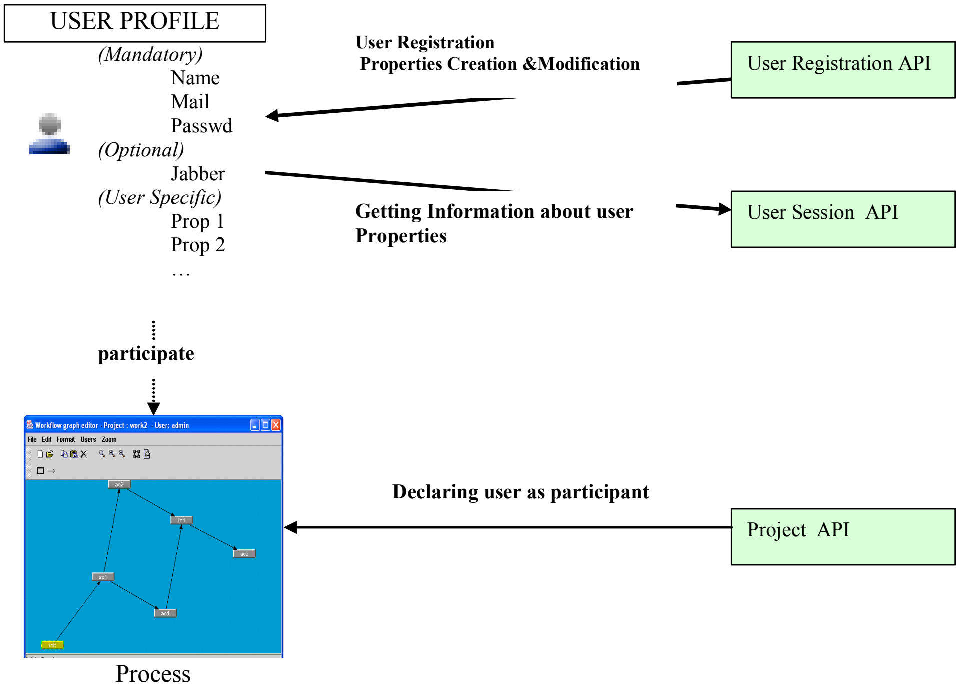


Figure 2-11. User Interface

2.4.1 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: that is, 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.

2.4.2 User Authentication Scenario

Bonita performs User Authentication using either a specific database (i.e. mySql, Postgres …), or an Ldap repository. The following code is an example of authentication of the ADMIN user. It uses the “TestClient” login context implemented in Bonita.

All other users are authenticated the same way.

```
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 2-12. Code Example for User Authentication

2.5 User 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.

2.5.1 J2EE Roles

Some 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 is, 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.

2.5.2 Bonita Roles

Bonita roles are related to activities access within processes. Each Process has its specific role management. This permits different means 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.

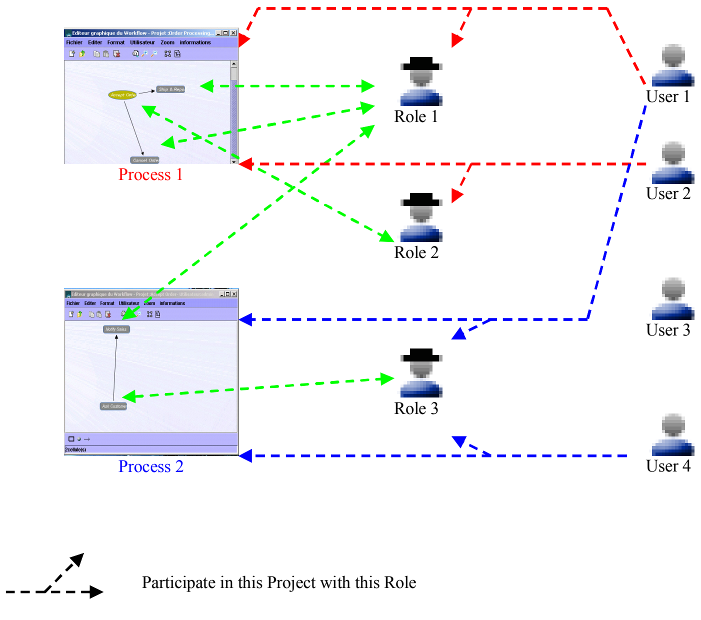


Figure 2-13. Role Association Illustration

User2 and User1 execute Process1 independently. 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 needs only 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 will return the name of an existing process that 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.

2.5.3 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 project 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:

- Designer
- Operator
- User

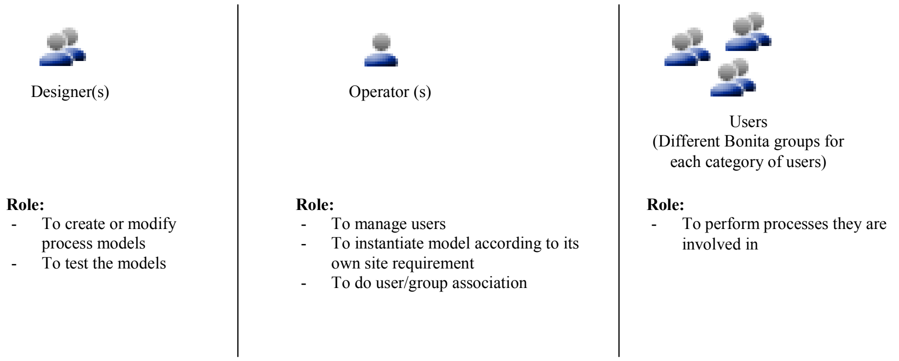

Designer(s)
Operator (s)
Users
(Different Bonita groups for each category of users)
Role:
- To create or modify process models
- To test the models
Role:
- To manage users
- To instantiate model according to its own site requirement
- To do user/group association
Role:
- To perform processes they are involved in

Figure 2-14. User Categories

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

Application restrictions could implement stronger access control than Bonita access control. It is advised, based on points previously mentioned, that lesser access control than the Bonita standard access control NOT be implemented.

- 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
- In this project, this user can access the node if he has the node role

2.6 Mappers Feature: Automatic Filling In of the Bonita Groups

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

2.6.1 Overview

Three methods (three types of mappers) are available, depending on the method employed to retrieve users in the system.

1. Using an LDAP server to obtain groups/roles (*LDAP mapper*)
2. Calling a java class to request a database (*custom mapper*)
3. 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 in section 5.2). Access is also available using the *graphEditor* (ProEd) application.

The Mapper function is of particular interest 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.

2.6.2 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:

1. Look at the sample class above and implement the custom mapper logic in a new java file.

2. Create a source .java file, i.e. *MyMapper.java*. It must be within the package *hero.mapper*.

3. Copy the java source file created above into the directory *BONITA_HOME/src/resources/mappers/hero/mapper*

4. Go to the *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*.

2.7 Performer Assignment

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

2.7.1 Overview

This feature permits additional assignment rules other than those defined in the standard Bonita model.

In the standard model (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 this 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*)

When this functionality is used, the user is notified (via mail notification), that the activity is ready to start.

The users of the groups, (roles in Bonita), associated to the activity will see the activity but cannot start or terminate it.

This functionality is accessible within the Bonita API (see ***ProjectSessionBean API***) and inside 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 previously).

☞ Note:

Only one performer assignment is valid at a given point. This means that although multiple performer assignments via the Bonita API may execute, only the last assignment is reflected in the structure BnNodeValue (see Table 11.2.8, BnNodeValue Attributes).

2.7.2 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, please follow the next steps:

1. Look at the sample class specified above and implement the performer assignment logic in a new java file.
2. Create a source .java file, i.e. *MyPerformer.java*. It must be within the package *hero.performer*.
3. Copy the java source file into the directory *BONITA_HOME/src/resources/performers/hero/performer*
4. 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

Properties Performer Assignment 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.

2.8 Initiator Mapper

This feature implements restrictions to the workflow models in Bonita.

2.8.1 Overview

The Initiator Mapper feature adds additional security constraints to the workflow instantiation operation. Through use of Initiator Mapper, the definition of users allowed to instantiate a particular workflow models is restricted (normally all users by default).

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

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

2.8.2 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 next steps:

1. Look at the sample class above and implement initiator logic within a new java file.
2. Create a source .java file, i.e. *MyInitiator.java*. It must be within the package *hero.initiatorMapper*.
3. Copy the java source file into the following directory:

   BONITA_HOME/ src\resources\iniitatorMappers\hero\initiatorMapper
4. 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 The LDAP Initiator.

- 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 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 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 of this version of Bonita:
  - 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.

Chapter 3.User Management

3.1 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.

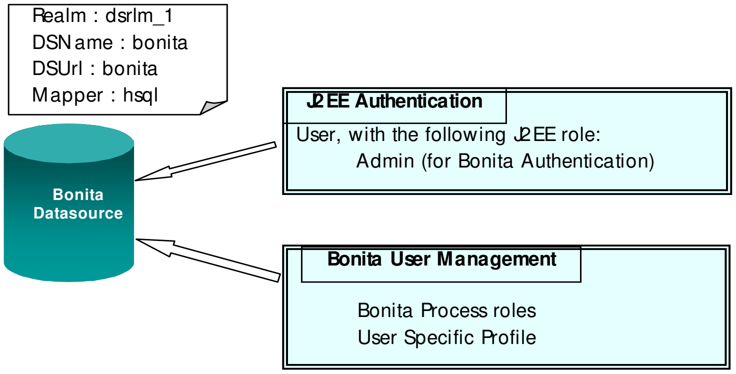


Figure 3-1. User Management Basic Configuration

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

3.2 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.

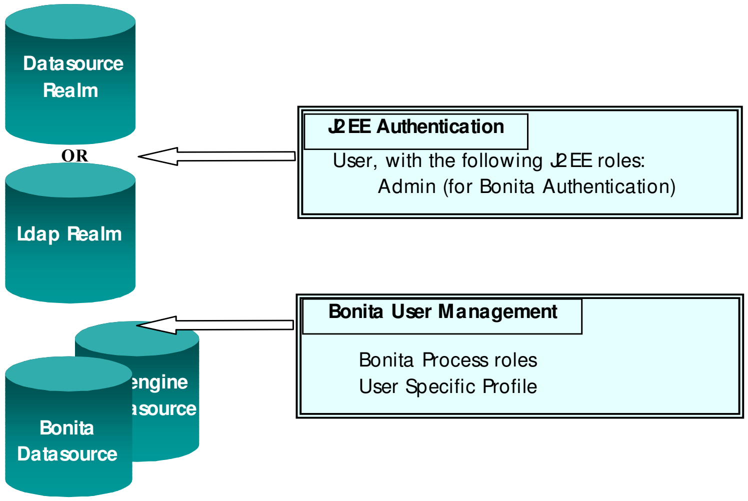


Figure 3-2. J2EE Authentication

3.2.1 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. Take a look at http://jonas.objectweb.org/current/doc/Config.html#Config-Security (look for *Configure LDAP resource in the jonas-realm.xml file*)

3.2.2 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).

Chapter 4.User Registration Interface

4.1 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 with an exception.

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

Important:

UserRegistration API should only be used when the User Management configuration is the Bonita default configuration. If a user-defined User Management implementation is being used, do NOT use the UserRegistration API.

4.2 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.

```
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 4-1. Code Example for the UserRegistrationBean

4.3 Managing Users (via the UserRegistrationBean)

4.3.1 Creating Users

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

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

An exception is thrown in the following cases:

- If user name already exists
- If an invalid parameter is specified

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

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

An exception is thrown in the following cases:

- If user name already exists
- If an invalid parameter is specified

4.3.2 Defining Users

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

This API function is used with the UserRegistration Bean and sets a property for the specified user “*username*”. User properties define user preferences. User properties are a key/value pair. If key already exists assign the new value. If key does not exist, create key and assign value.

An exception is thrown in the following cases:

- If user name does not exist
- If an invalid parameter is specified

`void setUserRole (String userName, String roleName)`

Set a new authorization role for the user.

An exception is thrown in the following cases:

- If user name does not exist
- If roleName cannot be added (I.e. role name not found).

4.3.3 Deleting Users

`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.

An exception is thrown in the following cases:

- If user name does not exist
- If user name is involved in other projects/processes

4.4 Creating User 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.

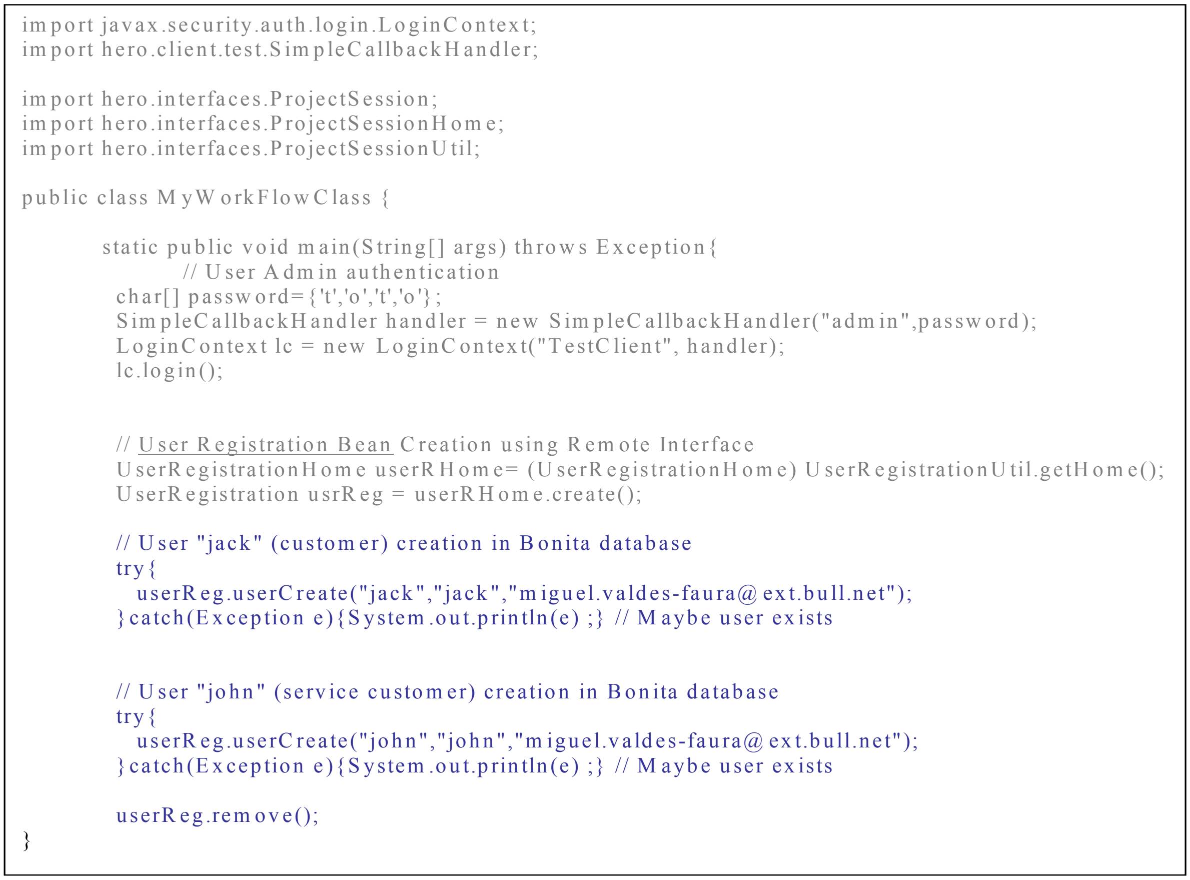
For more information about security roles and J2EE refer to:
http://jonas.objectweb.org/current/doc/PG_Security.html#PG_Security

An exception is thrown in the following case:

- If the role name already exists

4.5 Code Example

The following figure is a code Example for user creation.



```
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 4-2. Coding Example of User Creation

Chapter 5.Project Interface

5.1 Principle

The Project interface provides access to API functions that permit modification of execution for a given process.

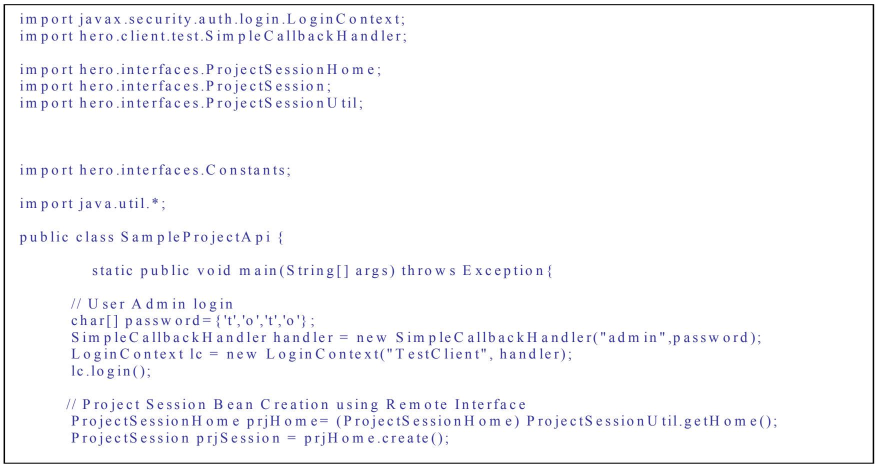
In the case of EJB Session access, the Project interface automatically retrieves the identity of the calling user in the J2EE security context. In this case, calling the Project interface from an unidentified context fails with an exception. Therefore, this interface is initiated for a given user. Only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

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

An example of code using this interface is shown below.

5.2 Creating the ProjectSessionBean

Think of the ProjetSessionBean as a handle into the Bonita workflow System. First, create the handle then associate a given project to this handle in order to modify it.



```
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 5-1. Coding Example for a Project Session Bean

5.3 Initiating the ProjectSessionBean

5.3.1 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 given projectName.

If the *projectName* does not exist, a new empty project is created and given this name. 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.

An exception is thrown in the following cases:

– If project name is null
– If project name already exists a warning message is displayed
– If there is a problem in the project version, type, or status

5.3.2 Initiating the Session Bean (Models)

`void initModel (String modelName)`

Creates or initializes workflow models.

The Project interface is initialized with the given modelName. If the corresponding *modelName* does not exist, a new empty model is created and given this name. 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.

An exception is thrown in the following cases:

– If model name is null
– If model name already exists a warning message is displayed
– If there is a problem in the project version, type, or status

5.3.3 Initiating a Project Using the Clone Project Creation Option

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

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

Important:

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

An exception is thrown in the following cases:

- If either old project name or new project name is null
- If new project name already exists
- If the creator does not have ADMIN privileges
- If creator name does not exist

5.3.4 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.

An exception is thrown in the following case:

- If there is any error detected in creating the instance of this project `void` **`initModelWithVersion`** `(String modelName, String modelVersion)`

Create a workflow process model or Initialize the Project Session Bean for this model for a particular version. This method is called after the API "create" call. After execution, all the API methods of ProjectSession API are available.

An exception is thrown in the following case:

– If modelName is null

`String` **`instantiateProject`** `(String project, Hashtable initProperties)`

Create a project instance. Call this method after executing the "create" API call. After execution, an instance of the specified project and all methods of ProjectSession API are available. This call uses the default project version.

Return the Hashtable – the default value for properties defined at process level.

`String` **`instantiateProject`** `(String project, String version, Hashtable initProperties)`

Create a project instance with the specified version. Call this method after executing the "create" API call. After execution, an instance of the specified project and all methods of ProjectSession API are available.

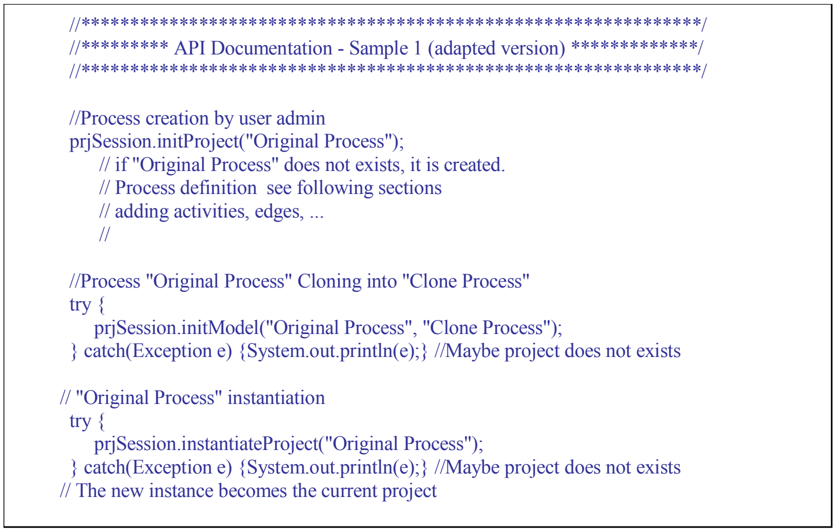
Return the Hashtable – the default value for properties defined at process level.

`String` **`instantiateProject`** `(String project, String version)`

Create a project instance with the specified version. Call this method after executing the "create" API call. After execution, an instance of the specified project and all methods of ProjectSession API are available.

Return a new Hashtable – the default value for properties defined at process level.

5.3.5 Code Example



```
 //*******************************************************************/
 //********* 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 5-2. Cloning Code Example

5.4 Managing a Project

With Bonita, there is a single API dealing with projects (i.e. the ProjectSessionBean). 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.

5.4.1 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 key/value pairs. Enumeration String types are also permitted.

When a project is created Bonita records the name of the user creating the project and the project creation date and other attributes.

The constant values associated with process states are shown in the following table.

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

Table 5-1. Process State Constants

The constant values associated with process types are shown in the following table.

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

Table 5-2. Process Type Constants

The constant values associated with process status are shown in the following table.

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

Table 5-3. Process Status Constants

5.4.2 Active/Hide a Workflow Process

`void` **`activeProcess`** `()`

This sets the process status to Active (model/cooperative/instance).

Workflow processes can only be executed or instantiated if status equals active

An exception is thrown in the following cases:
- If the issuing application does not have ADMIN privileges
- If the process is already active, a warning message is issued.

`void` **`hideProcess`** `()`

This sets the process status to Hidden (model/cooperative/instance).

This state allows workflow model modifications once they are instantiated.

An exception is thrown in the following cases:
- If the issuing application does not have ADMIN privileges
- If the process is already hidden, a warning message is issued.

5.4.3 Getting the Name of a Project or an INSTANCE

`String` **`getName`** `();`

Return the name of the project being managed by the current instance of the ProjectSessionBean interface. If there is no current instance, null is returned.

`String` **`getProjectNameOfInstance`** `(String instanceName)`

Return the project name for the instance “instanceName”.

An exception is thrown in the following case:
- If instanceName is not a project instance

5.4.4 Getting the Name of the Parent Project

`String` **`getParent`** `()`

If the current project is a subProcess, returns the name of its parent project. If this is not a sub process, returns the name of the created project.

5.4.5 Getting the Name of a Project's Creator

```
String getCreator ();
```

A string is returned with the name of the user creating the current Project. The projects creator name is automatically saved by the Bonita executive after a project is created thru the ProjectSessionBean Interface.

5.4.6 Properties

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

If key does not exist, creates a new property and assigns the value. If key exists, this function overrides the value of the existing property with the new value.

An exception is thrown in the following cases:

- If the value specified is not allowed (see setPossibleValues)
- If an invalid parameter is specified
- If issuer does not have access to this project

```
void setPropertyPossibleValues (String key, Collection values)
```

Set property possible values. Argument "values" represent a collection of data as possible property values. If "values" already exist for the specified key, the "values" remain unchanged (see updatePropertyPossibleValues). These values are compared when the setProperty API call is executed.

An exception is thrown in the following cases:

- If the user of the issuing application is not defined in this project
- If "key" is not found

```
void updatePropertyPossibleValues (String key, Collection values,
Collection defaultValues)
```

This API method allows users to dynamically change the possible values defined for this property (see setPropertyPossibleValues). This API method is used for enumeration types only. A default value is mandatory for this method.

An exception is thrown in the following cases:

- If the user of the issuing application is not defined in this project
- If "key" is not found
- If a default value is not one of the possible values

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

Return a `(BnProjectPropertyValue Collection)` of all properties existing for this project. See Table 11-11 BnProjectPropertyValue Attributes. If no properties exist for the project, "NULL" is returned.

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

Return a `(String Collection)` of all the properties keys for the current project. The referenced property is a pair key/value representing workflow relevant data but just the key names are returned. If no key values are assigned, "NULL" is returned.

`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);`

Delete a property of an existing project based on the specified key.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the specified key/property is not found

5.4.7 Project Details

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

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

Values returned are shown in Table 11-12, BnProjectValue Attributes.

`ArrayList` **`getChoices`** `(String value, Collection possibleValues)`

Get the list of choices into the value of the enumerated property. This API returns a list of possible choices based on the possible values. An example of usage is a checkbox with multiple selections.

`String` **`getVersion`** `()`

Return a String with the project version.

`String` **`getStatus`** `()`

Return a String with the project status (Active or Hidden).

`String` **`getType`** `()`

Return the type of the project (Cooperative/Model/Instance).

`Collection` **`getRolesValue`** `()`

Return a Collection of BnRoleValue objects – representing the roles of the project.

`BnRoleValue` **`getRoleValue`** `(String roleName)`

Return the values of this role in the project. See Table 11-14, BnRoleValue Attributes.

`Collection` **`getUsersRole`** `(String roleName)`

Return a collection of users matching with roleName in the current project.

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

Get a property value of the project. Return the properties associated to key/value pair for this project.
Returns null if the value is not found.

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

Return a collection of ProjectInterHookValue containing the project InterHooks.
Returns null if no InterHooks exist.

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

Returns a String with the inter hook value script. This method returns the hook script associated to all project nodes.
An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

`Object[]` **`getIterations`** `()`

Return a collection of BnIterationLightValue with all project iterations.

`Collection` **`getIterations`** `(String from)`

Get node destinations for this iteration. Return a collection of BnIterationLightValue of project iteration destinations for node “from”.

`boolean` **`getIterationExist`** `(String from)`

Returns TRUE if this node starts one or more iterations.

`Collection` **`getIterationConditions`** `(String from)`

Return a Collection of iteration conditions that start in node “from”. There may be multiple iterations specified for node “from”.

An exception is thrown in the following case:

- If String from (node) is not found

`boolean` **`isTerminated`** `()`

Test if all the projects nodes are terminated (in terminated state.) Returns TRUE if all project nodes are in terminate state. Returns FALSE if any node is not in terminate state.

`boolean` **`existingProject`** `(String projectName)`

Test to see if a project with projectName name exists. Returns TRUE if project name exists. This API uses the default project version (set to 1.0 in EventConstants.java).

An exception is thrown in the following case:

- If projectName is null

`boolean` **`existingProject`** `(String projectName, String projectVersion)`

Test to see if a project with projectName name exists with the specified projectVersion. Returns TRUE if project and version name exist.

An exception is thrown in the following case:

- If projectName is null

5.4.8 Code Example

```
 /****************************************************************/
 /************** 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 5-3. Project Properties Code Example

5.5 Defining and Obtaining Activity Information

5.5.1 Types of Activities

An Activity type can be one of the following types.

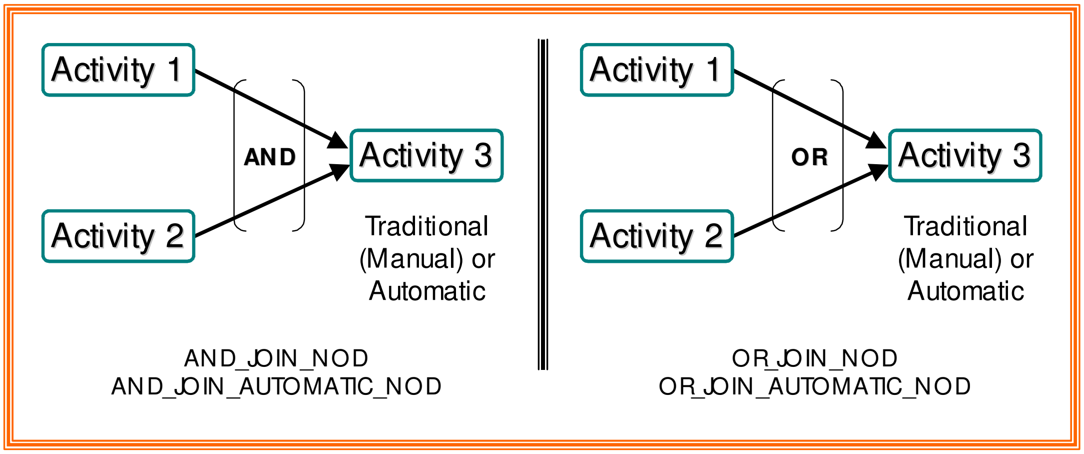


Figure 5-4. Activity Types

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

The following table displays the constant values associated with activity 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

Table 5-4. Constant Values for Node Types

5.5.2 Activities States

See:

See the “Activities basics “section 2.3.1 of this document.

The constant values associated with the main activities states are shown in the following table.

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	4
hero.interfaces.Constants.Nd.EXECUTING	5
hero.interfaces.Constants.Nd.EXECUTED	6
hero.interfaces.Constants.Nd.INERROR	7
hero.interfaces.Constants.Nd.FINISHED	8
hero.interfaces.Constants.Nd.TERMINATED	9
hero.interfaces.Constants.Nd.CHECKEDOUT	10
hero.interfaces.Constants.Nd.ANT_SUSPENDED	11
hero.interfaces.Constants.Nd.EXEC_SUSPENDED	12
hero.interfaces.Constants.Nd.BAD_TRANSITION	13
hero.interfaces.Constants.Nd.INITIAL	14

Table 5-5. Constant Values for Node States

5.5.3 Creating an Activity

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

Add "String name" node to the project. This method creates a node with the corresponding node type (See Table 5-4, Constant Values for Node Types) and assigns to it a role equal to **InitialRole**. This role is not assigned to any user at creation time, so this activity is not eligible for use until the setNodeRole method is called. This API call uses a default version of 1.0.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the specified node name already exists
- If node name is null
- If a parameter value is invalid

Note:

If JMS is enabled, a message is sent to the caller.

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

Add "String name" node to the project. This method creates a node with the corresponding node type (See Table 5-4. Constant Values for Node Types) and assigns to it a role equal to **InitialRole**. This role is not assigned to any user at creation time, so this activity is not eligible for use until the setNodeRole method is called. If the nodeType is a sub process, the version of the parent and sub process must match.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the specified node name already exists
- If node name is null
- If a parameter value is invalid
- If created node is a sub process, the projectVersion does not match the parent projectVersion

Note:

If JMS is enabled, a message is sent to the caller.

5.5.4 Creating SubProcess Activity

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

Add “name” subProcess node to the specified 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

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If an error occurs in the add sub process

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

Add “name” subProcess node to the specified 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

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If an error occurs in the add sub process

`void` **`deleteSubProcessNode`** `(String name, String version)`

Delete a node (“name”) with type subProcess from the project if this node is not in the executing state.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the state of the node is executing
- If name cannot be found
- If name has a subProcess (all subProcess nodes must be removed first)

5.5.5 Configuring an Activity

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

Set the information on node changes (including role, description, and deadline). This is commonly used by the graphical client application included in the Bonita distribution (struts based)

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

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

Set the node in anticipable mode. The anticipable attribute is set true. See Section 11.2.8, BnNodeValue attributes. Please see Section 2.3.1 for an explanation of traditional versus anticipable mode.

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

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

Set the node in automatic mode (sets node to anticipating state). The responsibility of activity execution is now under control of the Bonita engine. (That is, the Node “name” is no longer manually started.)

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

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

Set an absolute node deadline (i.e. 11-05-2006). The activity deadline is the latest date by which the activity must be finished. This API Call is **deprecated**. ***Call is*** *replaced by setNodeDeadlines following below.*

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the specified date is before the current date.

void **setNodeDeadlines** (String name, Collection co)

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

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the new deadline is before/earlier than current date.

Note:

If JMS is enabled, a message is sent.

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

Set a relative node deadline (ex: 2 hours). Activity deadline is the latest date or time in which the activity must be finished. Call is **deprecated and replaced** *by setRelativeDeadlines explained below.*

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

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

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

Note:

If JMS is enabled, a message is sent.

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

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

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

`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 defined key/value property to other nodes automatically. If the key name does not exist, the key is created and assigned the value.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If key name does not exist a warning message is issued.
- If the supplied value is not allowed (see setNodePropertyPossibleValues)

`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. If the key name does not exist, the key is created and assigned the value.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If key name does not exist a warning message is issued.
- If the supplied value is not allowed (see setNodePropertyPossibleValues)

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

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

If the key name does not exist, the key is created and assigned the value.

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

`Void` **`updateNodePropertyPossibleValues`** `(String nodeName, String key, Collection values, Collection defaultValues)`

Update possible values for a specific node. Key/value must be enumerated type. The collection of defaultValues represents the values replacing the collection of (current) values.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If key name does not exist.
- If current value does not compare to stored value (invalid value error.)

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

Set the node in traditional mode. When a node is traditional the anticipable attribute is set to false. This method must be used if you want to execute this activity in a traditional. Refer to Section 2.3.1 for an explanation of traditional versus anticipable mode.

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

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

Set the node type. Change the current type of the node (if node is not executing). See Table 5-1, Constant Values for Node Types.

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

`void` **`addInitiatorMapper`** `(String mapperName, int mapperType)`

Add or Update a mapper for the INITIATOR role. This type of mapper uses a Java file loaded at run time. If the mapper name exists it is updated, otherwise the mapper name is created.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If a parameter value is invalid

5.5.6 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 following figure, 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.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the iteration request does meet the listed constraints in Section 2.3.3, Iterations.

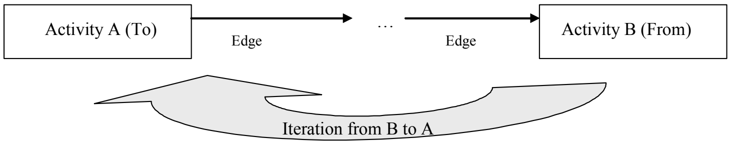


Figure 5-5. Iteration Example

The iteration condition may be something like “lastNodeProperty.equals (\”value\”)”. The value of the property is evaluated depending on the execution of the process. That is, the value may change during process execution and is evaluated to make the iteration decision.

`void deleteIteration (String from, String to)`

Delete iteration between two nodes.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the “From” and/or “To” nodes do not exist

5.5.7 Getting Information about Nodes in the Project

`Object` **`getNodes`** `()`

Returns project nodes data as an array of StrutsNodeValue. This is especially useful for a Struts-based web application, but may be used in any type of application.

`Collection` **`getNodesNames`** `()`

Return a String Collection of all node names in the project. If no nodes exist, “NULL” is returned.

5.5.8 Getting Information about a Specific Node

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

Get Node Value from a specific project (See Table 11.2.8, BnNodeValue Attributes).

An exception is thrown in the following cases:

- If the project name does not exist
- If the named project is not accessible by this requestor

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

Return a node deadline. Activity deadline is the latest date or time by which the activity must complete. If no deadline exists null is returned. This call is **deprecated** and replaced by getNodeDeadlines.

An exception is thrown in the following case:

- If the node name does not exist

`Collection` **`getNodeDeadlines`** `(String nodeName)`

Return a collection of deadlines for the node. Activity deadline is the latest date by which the activity must finish. If no deadlines exist for the node, null is returned.

An exception is thrown in the following case:

- If the node name does not exist

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

Return the node description. Node description represents explicit execution related information for this task. See Table 11.2.8, BnNodeValue Attributes.

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

Return the node executor. Return the name of the user executing the activity. See Table 11.2.8, BnNodeValue Attributes.

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

Returns a (BnNodePropertyValue Collection) of Node properties as a list of pair key/value properties assigned to the node. See Table 11.2.8, BnNodeValue Attributes. If no key/value properties exist, “NULL” is returned.

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

Return Node property value. Get the pair key/value properties associated to the node. See Table 11.2.8, BnNodeValue Attributes. If no key/value properties exist, “NULL” is returned.

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

Return the state of the node. See Table 11.2.8, BnNodeValue Attributes.

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

Return the type of the node. See Table 11.2.8, BnNodeValue Attributes.

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

Return the node name Value. See Table 11.2.8, BnNodeValue Attributes.

An exception is thrown in the following case:

- If the node name does not exist

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

Return true if the node is ready for execution in anticipated mode. See Table 11.2.8, BnNodeValue Attributes.

5.5.9 Deleting an Activity

`void deleteNode (String name)`

Delete a node from the project.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the named node is executing
- If the named node has a sub process (must delete sub process first)

`void deleteNodeProperty (String nodeName, String key)`

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

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the named key/property is not found
- If the named node does not exist

5.5.10 Model Definition Check

`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 Section 2.3.3. Iterating Activities) are verified, but in future versions this method may include other model definition verification.

checkModelDefinition Method Verification

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

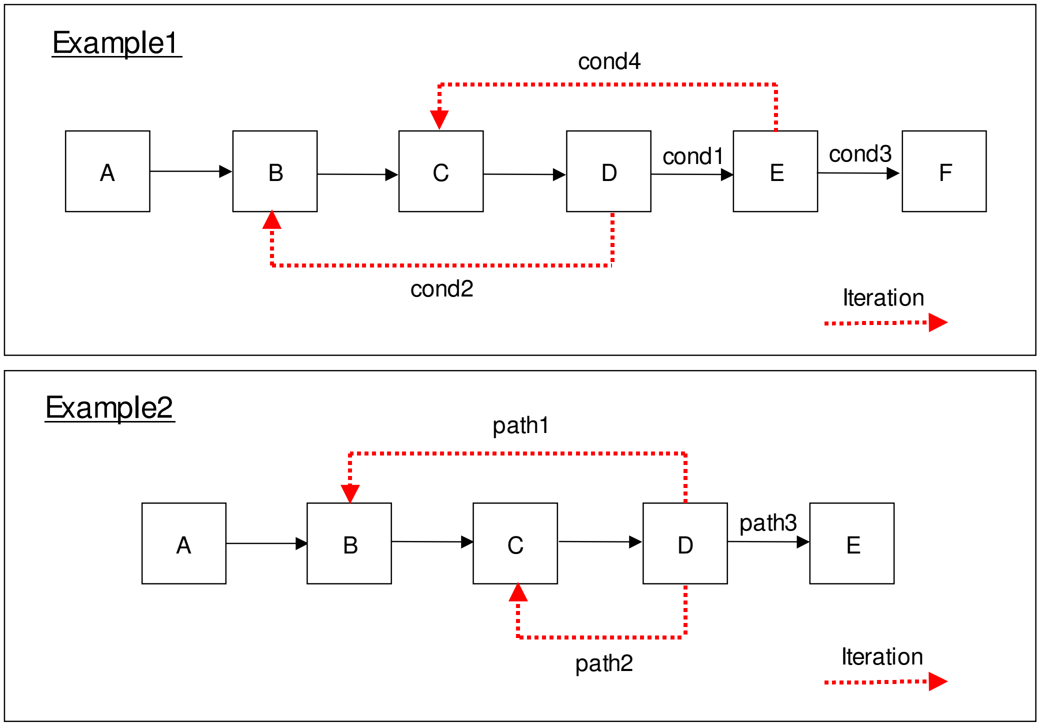


Figure 5-6. Check Model and Verification

checkIteration method:

- Checks that the iteration's conditions are not empty
- In example1 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 <u>NOT</u> allowed as this creates a condition that produces 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 the path does not exist, an exception is thrown.
- If multiple iterations exist in the same node, then a check verifies that the iteration conditions are different. If the conditions are not different, an exception is thrown. In example2 of Figure 5-6 (Check Model and Verification), path1 condition must be different from path2 condition.

checkMandatoryIterationConditions ():

- Verifies that mandatory conditions on the out edges of nodes creating an exit point from the iteration.

  In example1 of Figure 5-6 (Check Model and Verification), 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 example1 of Figure 5-6 (Check Model and Verification), cond1 must be different from cond2 and cond3 must be different from cond4.

  Throw a HeroException if the iteration's starting condition and out edge's condition are equal.

Code Example

```
.../...
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 5-7. CheckModelDefinition Code Example

5.6 Managing Edges

5.6.1 Adding an Edge to an Activity

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

Edges have unique names in the scope of the project. The name of the edge can be assigned by the application, or automatically generated by Bonita. Edges may also be created using the ProEd facility.

```
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.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the “IN” or “OUT” node does not exist
- If a path from “IN” to “OUT” does not exist
- If an edge already exists
- If “OUT” is in execution

5.6.2 Deleting an Edge

```
void deleteEdge (String name);
```

The edge with the parameter “name” is deleted.

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If the named edge does not exist
- If out edge is in execution

Note:

If JMS is enabled, a message is sent.

5.6.3 Getting Connected Activities from an Edge

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

Return the name of the inbound node of the given edgeName.
An exception is thrown in the following case:

- If the named edge does not exist

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

Get back the name of the outbound node of the given edgeName.
An exception is thrown in the following case:

- If the named edge does not exist

5.6.4 Setting a Condition on an Edge

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

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

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

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

Note:

During execution an Edge condition is evaluated. If a condition does not follow correct format an exception is thrown.

5.6.5 Getting the Condition for an Edge

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

Get the edge Condition. This condition is evaluated at run-time in order to perform activity transition. See setEdgeCondition above.

An exception is thrown in the following case:

- If the named edge does not exist

5.6.6 Get All Existing Edges in a Project

`Collection` **`getEdgesNames`** `()`

Return a String collection of all existing edges in the project. If no edges exist, “NULL” is returned.

5.6.7 Get All Existing Edges for an Activity

`Collection` **`getNodeInEdges`** `()`

Return a (String Collection) of all existing inbound edges for a given node. If no edges exist, “NULL” is returned.

`Collection` **`getNodeOutEdges`** `()`

Return a (String Collection) of all existing outbound edges for a given node. If no edges exist, “NULL” is returned.

5.6.8 Reading an Edge as a Java Object

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

Get the edge value. See Table 11-2.

An exception is thrown in the following case:

- If the named edge does not exist

5.6.9 Changing the State of an Edge

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

Set the edge state (see the following table) to integer state. Table 11-2, BnEdgeValue Attributes.

Note:

If JMS is enabled, a message is sent.

CONSTANT	VALUE
INITIAL	0
ACTIVE	1
ANTICIPATING	2
DEAD	3

Table 5-6. Edge Constants States

Chapter 6.Hook Interface

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

Hooks may be coded in a scripting language (i.e. XPDL), 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.
- Hooks execute upon detection of one of the following events. 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.

  For example, the following code lists multiple hook events, but the return String from getMETAdata specifies what event this hook code acts upon.

```
public String getMetadata() {
         return Constants.Nd.BEFORETERMINATE;
}
public void create(Object b,BnNodeLocal n) throws HeroHookException {}
public void beforeStart(Object b,BnNodeLocal n) throws HeroHookException {}
public void afterTerminate(Object b,BnNodeLocal n) throws HeroHookException {}
public void onCancel(Object b,BnNodeLocal n) throws HeroHookException {}
public void anticipate(Object b,BnNodeLocal n) throws HeroHookException {}
public void onDeadline(Object b,BnNodeLocal n) throws HeroHookException {}
public void afterStart(Object b, BnNodeLocal n) throws HeroHookException {}
public void onReady(Object b,BnNodeLocal n) throws HeroHookException {}
public void beforeTerminate(Object b,BnNodeLocal n) throws HeroHookException {
try {
         String nodeName = n.getName();
         BnProjectLocal project = n.getBnProject();
         String prjName = project.getName();
```

For more examples refer to Section 4 of the Bonita Workflow Developer's Guide.

Table 6-1 displays 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

Table 6-1. Node Hooks Events Constants

Project **Hooks** Events:

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

Table 6-2. Project Hook Events 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

Table 6-3. Hook Type Constants

6.1 Project Hook Management

6.1.1 Creating Hooks

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

Add an existing hook file to the project. This hook type 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.

An exception is thrown in the following cases:
- If the issuing application does not have "ADMIN" privileges
- If hook name already exists in the project
- If an invalid value is specified for type (see Table 6-3, Hook Type Constants)

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

Add an InterHook to the project. Creates a new hook associated to all project activities. See Section 6.1.3, Managing Hooks, for parameter values. The fourth parameter "String value" represents a hook script used with InterHook. See API call addNodeInterHook for a script example.

An exception is thrown in the following cases:
- If the issuing application does not have ADMIN privileges
- if hookName already exists

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

Set the Interhook value. This value is the new Interhook script associated to all project nodes. See addNodeInterHook below for a script example.

An exception is thrown in the following cases:
- If the issuing application does not have ADMIN privileges

6.1.2 Deleting Hooks

`void **deleteHook** (String hookName)`

Deletes the hook specified by hookName in current project.

An exception is thrown in the following cases:
- If the issuing application does not have "ADMIN" privileges
- If hook name does not exist in the project

`void **deleteInterHook** (String hookName)`

The hook or interHook specified by "hookName" is deleted from all project nodes.

An exception is thrown in the following cases:
- If the issuing application does not have "ADMIN" privileges
- If hook name does not exist in the project

6.1.3 Managing Hooks

`Collection **getHooks** ()`

Return a (ProjectHooksValue Collection) of all the hook names assigned to the project. If no hooks exist in the project, "NULL" is returned.

`void **executeProcessHook** ()`

Execute the OnInstatiate hook associated to this process. This method can only be called before a workflow model is instantiated.

An exception is thrown in the following case:
- If the project type indicates MODEL status (this would indicate the process is already instantiated.)

6.2 Node Hook Management

6.2.1 Creating Specific Hooks

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

Add an existing hook file to the node (activity). 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 (an ant or ant light-main tasks). Please refer to the Workflow Process Console Developers Guide for detailed explanation.

An exception is thrown in the following cases:

- If the issuing application does not have “ADMIN” privileges
- If nodeName does not exist
- If hookName already exists
- If an invalid parameter is specified (See Table 6-3. Hook Type Constants)

`void` **`addNodeInterHook`** `(String nodeName, String hookName, String eventName, int hookType, String script)`

The interhook name “hookName” is added to the node specified by nodeName. The hook activation is triggered whenever the event “eventName” occurs for this activity. See events defined in Table 6-1, Node Hooks Events Constants. The InterHook uses a Java or a beanshell scripting file (See example below), executed at run time.

An exception is thrown in the following cases:

- If nodeName does not exist
- If the issuing application does not have ADMIN privileges
- If hookName already exists
- If an invalid parameter is specified (See Table 6-3, Hook Type Constants)

Script file example:

The API call “addNodeInterHook” references a value “script” in parameter four. As shown below, “script is defined as an ASCII string.

```
string script =
  "import hero.interfaces.BnProjectLocal;\n"
  + "import hero.interfaces.BnNodeLocal;\n"
  + "afterStart (Object b,Object n) {\n\n\n"
  + "System.out.println(\"InteractiveBnNode Hook test, node:
        \"+n.getName());"
  + "}";

prjSession.addNodeInterHook("projectInterTest",hero.interfaces
.Constants.Nd.AFTERSTART,Constants.Hook.BSINTERACTIVE,script);
```

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

Set the node Interhook value. This defines the script used by this node for the specified hook.

An exception is thrown in the following case:

- If the issuing application does not have ADMIN privileges

6.2.2 Deleting Specific Hooks

`void` **`deleteNodeHook`**`(String nodeName, String hookName)`

Delete a node hook. Delete the hook (specified by hookName) for the activity/node specified by nodeName.

An exception is thrown in the following cases:

- If hookName does not exist.
- If nodeName does not exist
- If the issuing application executing the function does not have ADMIN privileges.
- If the EJB does not permit removal

`void` **`deleteNodeInterHook`**`(String nodeName, String interHookName)`

Delete a node interHook in node specified by nodeName. The hook or the interHook with name interHookName is deleted from the node.

An exception is thrown in the following cases:

- If interHookName does not exist.
- If nodeName does not exist
- If the issuing application executing the function does not have ADMIN privileges
- If the EJB does not permit removal

6.2.3 Managing Specific Hooks

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

Return a (NodeHookValue Collection) of the Node hooks of the specified node. If no hooks exist, "NULL" is returned.

An exception is thrown in the following case:
- If nodeName does not exist.

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

Return a (NodeInterHookValue Collection) of all Interactive Node hooks of the specified node. If no interhooks exist, "NULL" is returned.

An exception is thrown in the following case:
- If nodeName does not exist.

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

Return all the node inter hook data associated to the hook of name « interHook » for the node « nodeName ». If interhook does not exist, "NULL" is returned.

An exception is thrown in the following case:
- If nodeName does not exist.

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

This method returns the hook script associated with the interhook name « hook » of this node. If Hook value does not exist, null is returned.

An exception is thrown in the following cases:
- If node Name does not exist
- If application does not have "ADMIN" privileges

6.2.4 Code Example

```
/*****************************************************************/
/************** 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 6-1. Code Example Activity Properties

Chapter 7.User Management Interface

The following methods are dedicated to managing users for a particular workflow model or instance.

7.1 Getting the List of All Bonita Registered Users

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

Return a (String Collection) with the names of all registered users in the Bonita System. If no users are found, “NULL” is returned.

7.2 Getting the List of Users for a Project

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

Return a (String Collection) of all users of the current project. If no users exist, “NULL” is returned.

7.3 Adding a User to a Project

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

Add a user to this project (This user must exist in the Bonita database)
An exception is thrown in the following cases:

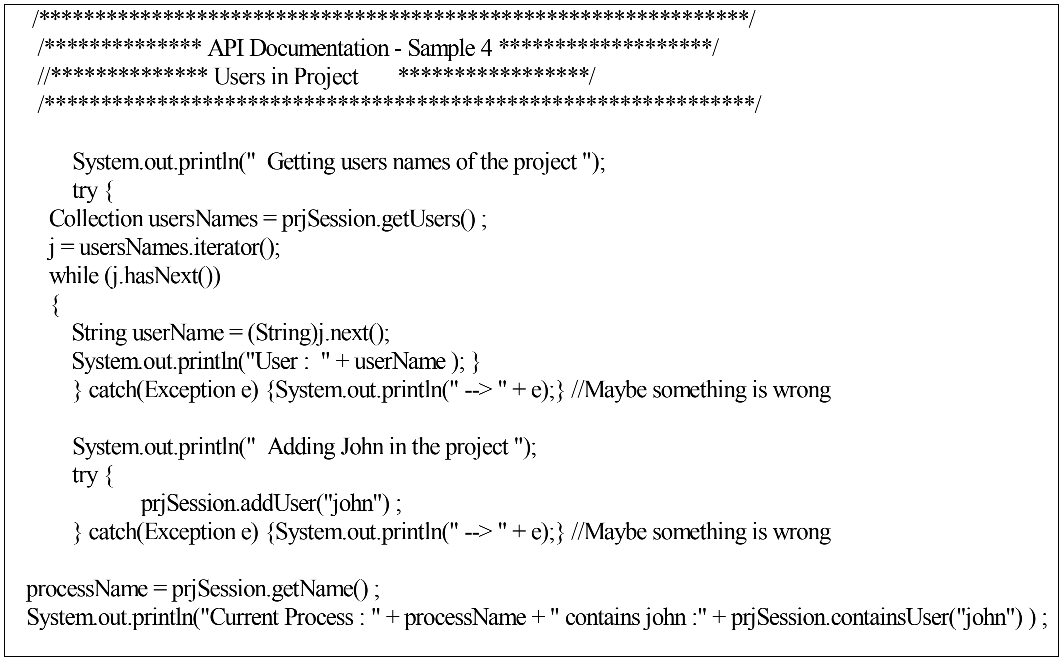
- If the user name is not found for the project (user not registered)
- If the issuing application does not have “ADMIN” privileges

7.4 Checking Whether a User Is Part of a Project

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

Test if the “username” is associated to this project. Returns “true” if user found.

7.5 Code Example



```
/****************************************************************/
 /************** 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 7-1. Code Example Get User Names

Chapter 8.Project Role Management

8.1 Managing Project Roles

Using ProjectSession Bean

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

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

8.1.1 Declaring a New Role in the Project

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

This function creates a role within this project. The created role is specific to this project.

An exception is thrown in the following case:

- If the issuing application does not have “ADMIN” privileges

`void **deleteRole** (String roleName)`

Delete a role (and the Role mapper if it exists).

An exception is thrown in the following cases:

- If the issuing application does not have ADMIN privileges
- If roleName does not exist

8.1.2 Allocating a Role to a User

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

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

Assigns to “username” the role specified in “roleName”.

An exception is thrown in the following cases:
- If the issuing application does not have “ADMIN” privileges
- If the user name and/or role name is not found

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

Remove the role specified by “roleName” from the user specified by “username”.

An exception is thrown in the following cases:
- If the issuing application does not have “ADMIN” privileges
- If the user name is not found
- If the role name is not found

Note:

If JMS is enabled, a message is issued.

8.1.3 Getting a List of Roles That a User Can Assume

```
Collection getUserRoles (String userName)
```

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

An exception is thrown in the following case:
- If the user name is not found

8.1.4 Getting a List of Roles That a User Can Assume in the Scope of a Project

`Collection **getRoles** ()`

Return a (BnRoleLocal Collection) of 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** ()`

Return a (String Collection) of the names of all roles for the current project as a collection of String objects. If no roles are found, "NULL" is returned.

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

Return a (BnRoleValue Collection) of the roles of this user in the current project. If no roles are assigned for the user, "NULL" is returned.

An exception is thrown in the following case:

- If the user name is not found

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

Return a (String Collection) of the role names of the user in the current project. If no roles are assigned, "NULL" is returned.

An exception is thrown in the following case:

- If the user name is not found

8.1.5 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.

An exception is thrown in the following case:

- If the node name is not found

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

Sets or changes the role of an activity if the role name already exists. If JMS is enabled a message is issued.

An exception is thrown in the following cases:

- If the issuing application does not have “ADMIN” privileges
- If the activity name is not found
- If the role name is not found

8.1.6 Code Example

```
/**********************************************************************/
/************** 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 8-1. Code Example Project Roles

8.2 Mappers

8.2.1 Adding and Deleting Role 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 a role mapper does not exist, one is created with mapperName.

The 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

An exception is thrown in the following case:

– If the issuing application does not have "ADMIN" privileges
– If the role name does not exist
– If an invalid value is used

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

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

An exception is thrown in the following cases:

– If the issuing application does not have "ADMIN" privileges
– If the role name does not exist

```
Collection getRoleMappers ()
```

Return a (BnRoleMapperValue Collection) of all the role mappers of the project. If "roleMapper "does not exist, "NULL" is returned.

8.2.2 Code Example

```
      .../....
        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 8-2. Code Example Add Role Mapper

Examples of Mapper code are available under:

BONITA_HOME/src/resources/mappers/hero/mapper.

8.3 Performer Assignment

8.3.1 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.

PerformerAssignType can be one of the following:

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

Multiple assignments are possible but only the last assignment is valid and reflected in any "get". This means, if the node has a current performer assignment, the function updates the current assignment with the new values.

An exception is thrown in the following cases:

– If the issuing application does not have "ADMIN" privileges
– If the node name does not exist

8.3.2 Code Example

```
..../....
   // 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 8-3. Performer Assignment Code Example

`BnNodePerformerAssignValue` **`getNodePerformerAssign`** `(String nodeName)`

Return a value of type BnNodePerformerAssignValue. If no performer assignment has occurred a null value is returned. See Table 11-6, BnNodePerformerAssignValue Attributes.

An exception is thrown in the following case:

– If the node name does not exist

`BnRoleLocal getNodeRole (String nodeName)`

Return BnRoleLocal object – with role data.

`BnNodeLightValue` **`getNodeLightValue`** `(String name)`

Return the node Light Value basic node information.

An exception is thrown in the following case:

– If String name is not found

Chapter 9.User Session Interface

9.1 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.

For 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.

Much of the User interface methods require the Project name as a parameter. The project 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 a 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: a recursive implementation that manages previous execution operations and propagates the activity state changes to activities connected to this one.

The User Session Bean API provides information about user projects and activities (i.e. project list, ToDo list, and activity list), and is also 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.

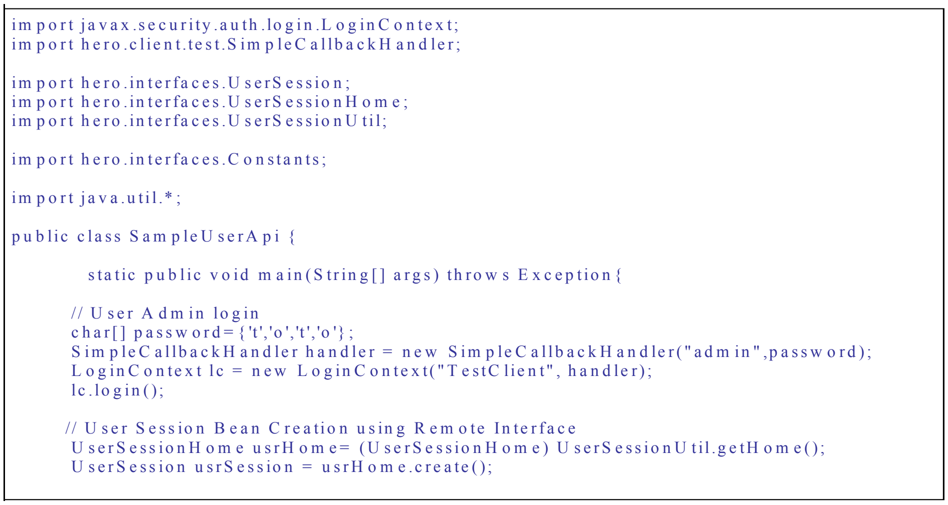
Coding examples using the User Session interface API are shown in the following sections.

9.2 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.

Subsequent calls to the User Session API functions are related to this identity.

Code Example



```
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 9-1. Session Interface Code Example

9.3 User Properties

9.3.1 Setting User Properties

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

This function is in the UserSessionBean. Using name “*key*”, this function sets the property (identified by the key), to the value “*value*”.

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

An exception is thrown in the following cases:
- If the key name does not exist
- If an invalid value is used

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

Set the mail of this user into the Bonita database.

An exception is thrown in the following case:
- If the user name does not exist

9.3.2 Getting User Information

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

Return the name of the current authenticated User.

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

Return the current user password

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

Return the mail address for this user from Bonita database.

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

Return a (BnUserPropertyValue Collection) of the properties defined for the current authenticated User.

`Collection` **`getModelList`** `()`

Return a (BnProjectLightValue Collection) of the current Workflow models associated with this user.
If no models exist for the user, null is returned.

`Collection` **`getModelList`** `(int offset, int numrows)`

Return a collection of BnProjectLightValue objects of a user model list starting with "offset" for "numrows". This method is equivalent to getProjectList but returns only the current models of the user.
If no models exist, null is returned.

`Collection` **`getModels`** `()`

Return a (BnProjectLightValue Collection) of the current available Workflow models. If the user issuing the API call is part of an Initiator Mapper (see section 2.8), then the initiator mapper file is used.
An exception is thrown in the following case:
- No active models exist

`Collection` **`getModelsLight`** `()`

Return a (BnProjectLightValue Collection) of the current available Workflow models without a check for Initiator Mapper.

An exception is thrown in the following case:
- No active models exist

`Collection` **`getCooperativeList`** `()`

Similar to getProjectsList, returns a (BnProjectLightValue Collection) of the current available Workflow projects with type cooperative.

If no cooperative projects exist for the user, null is returned.

`Collection` **`getCooperativeList`** `(int offset, int numrows)`

Return a collection of BnProjectLightValue objects of a user cooperative projects list starting with “offset” for “numrows”. This method is equivalent to getProjectList but returns only the current cooperative projects of the user.

If no cooperative projects exist, null is returned.

`Collection` **`getTerminatedListAllInstances`** `()`

Return a BnNodeValue collection of all terminated project instances or cooperative projects.

If no terminated instances or terminated cooperative projects exist for the user, null is returned.

`Collection` **`getModelInstancesTodoList`** `(String projectName)`

Return a String collection of activity names of the model instances assigned to this user.

An exception is thrown in the following case:
- The supplied projectName does not exist

`Collection` **`getModelInstancesTerminated`** `(String projectName)`

Return a String collection of activity names of the model instances terminated by this user.

An exception is thrown in the following case:
- The supplied projectName does not exist

`Collection` **`getModelInstancesTodoList`** `(String projectName, int offset, int numrows)`

Return a String collection of activity names of the model instances assigned to this user starting with “offset” for “numrows”.

An exception is thrown in the following case:

- The supplied projectName does not exist

`Collection` **`getModelInstancesTerminated`** `(String projectName, int offset, int numrows)`

Return a String collection of activity names of the model instances terminated by this user starting with “offset” for “numrows”.

An exception is thrown in the following case:

- The supplied projectName does not exist

`Collection` **`getInstancesActivityTodoList`** `(String projectName, String nodeName)`

Return a BnNodeLightValue collection of activity instances assigned to this user.

An exception is thrown in the following case:

- The supplied projectName does not exist
- The supplied nodeName does not exist

`Collection` **`getInstancesActivityTerminated`** `(String projectName, String nodeName)`

Return a BnNodeLightValue collection of activity model instances terminated by this user.

An exception is thrown in the following case:

- The supplied projectName does not exist
- The supplied nodeName does not exist

`Collection` **`getUserInstancesProject`** `()`

Return a String collection of model names with ongoing instances for this user.

If no instances exist for this user, null is returned.

`Collection` **`getUserInstancesProjectNodes`** `(String projectName)`

Return a String collection of ready, anticipable, and executing activities for current user instances.

An exception is thrown in the following case:

- The supplied projectName does not exist

`Collection` **`getUserInstancesProject`** `(int offset, int numrows)`

Return a String collection of model names with ongoing instances for this user starting with offset for numrows.

If no instances exist for this user, null is returned.

`Collection` **`getInstanceNodes`** `(String instanceName, String userName)`

Return a String collection of ready and executing activities for a specified user instance.

An exception is thrown in the following cases:
- The supplied projectName does not exist
- The supplied userName does not exist

`Collection` **`getInstancesList`** `(int offset, int numrows)`

Return a collection of BnProjectLightValue objects of a user instances list starting with offset for numrows. This method is equivalent to getProjectList but returns only the current instances of the user.

If no instances exist, null is returned.

`Collection` **`getInstancesListNames`** `(int offset, int numrows)`

Return a String collection of instances list names for this user starting with “offset” for “numrows”. This method is equivalent to getProjectListNames but returns only the current instances of the user.

If no instances exist, null is returned.

`String` **`getUserJabber`** `()`

Return a String with the user jabber address.

9.4 Users and Projects

9.4.1 Getting the List of Projects for the User

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

Return a (BnProjectLightValue Collection) of the Workflow processes associated to this user.

An exception is thrown in the following case:
- If the user does not exist in the database

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

Return a (String Collection) of the project list names for this user.

An exception is thrown in the following case:
- If the user does not exist in the database

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

Return a (BnProjectValue Collection) of Workflow projects associated with a property.

An exception is thrown in the following case:
- If the key/value names do not exist

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

Return a (String Collection) of Workflow projects associated with a property.

An exception is thrown in the following case:
- If the key/value names do not exist

`BnProjectLightValue` **`getLightDetails`** `()`

Return the basic project information: project attributes (without relationships).

9.4.2 Getting the List of Instances for the User

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

Return a (BnProjectLightValue Collection) of the user instances list. This method is equivalent to getProjectList but returns only the current instances of the user. If no instances exist, null is returned.

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

Get a (String Collection) of the instances list names for this user. This method is equivalent to getProjectListNames but returns only the current instances of the user. If there are no entries in the list, null is returned.

`Collection` **`getProjectInstances`** `(String projectName, String version)`

Return a (BnProjectValue Collection) of the Workflow instances of this project.

An exception is thrown in the following case:

- If projectName does not exist

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

Return a (BnProjectValue Collection) of the Workflow instances of this project.

An exception is thrown in the following case:

- If projectName does not exist

A default version value is used in this API call.

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

Return a (String Collection) of Workflow instance names of this project.

An exception is thrown in the following case:

- If projectName does not exist

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

Return a (String Collection) of workflow instance names of this project.

An exception is thrown in the following case:

- If projectName does not exist

A default version value is used in this API call.

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

Return a (BnProjectValue Collection) of Workflow instances from a property.

An exception is thrown in the following case:
- If key/value does not exist

`Collection getInstancesByPropertyNames (String key, String value)`

Return a (String Collection) of a list of project instances from a property.

An exception is thrown in the following case:
- If key/value does not exist

9.4.3 Managing the Project for the User

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

Delete a Workflow project. The application must have "ADMIN" privileges or an exception is thrown.

An exception is thrown in the following cases:
- If the project name does not exist
- If the project name has running instances.
- If you are not the parent process (sub-process trying to remove project).

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

Attempt to terminate a project. (Termination occurs when all project activities are terminated).

An exception is thrown in the following cases:
- If the project name does not exist
- If project name is active.

9.5 Users and Activities

9.5.1 Getting the List of Activities for the User

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

Return a (String Collection) of all user activities for a specific project in executing and anticipating state. See also the **getToDoList** for activities in ready state.

An exception is thrown in the following case:
- If the project name does not exist

If no activities are found in executing state null is returned.

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

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

If no activities are found in executing state null is returned.

`Collection getActivityListByProperty (String key, String value)`

Return a (BnNodeValue Collection) of executing user activities matching the property value (executing and anticipating state activities).

If the key/value is not found, null is returned.

9.5.2 Getting Information on User activity

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

Return a type BnNodeValue for the specified node and project.

An exception is thrown in the following cases:
- If the project or node name does not exist
- If access to the project is denied

9.5.3 Getting the ToDo list for the User

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

Return a (String Collection) of all user activities from the specified project (those in ready and anticipable state).

An exception is thrown in the following case:

- If the project name does not exist

If no users exist in the correct state, null is returned.

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

Return a (BnNodeValue Collection) of a list of ToDo activities for the user for all instances (ready and anticipable state).

If no users exist in the correct state, null is returned

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

Return a (BnNodeValue Collection) of a list of ToDo activities for the user matching the property value (ready and anticipable state activities).

If no users exist in the correct state, null is returned.

`Collection` **`getToDoListByProperties`** `(int operation, Hashtable properties)`

Return a BnNodeValue collection of a list of ToDo activities (in ready and anticipable state), matching the property value(s). Property values are a key/value pair and up to three properties may be specified. Operation must be a value of either hero.interfaces.Constants.OR or hero.interfaces.Constants.AND.

An exception is thrown in the following case:

- Greater than three properties are specified.

(Contact the Bull HN Workflow project if greater then three properties are required.)

If no data is found matching the specified properties, null is returned.

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

Return a BnNodeValue collection of a list of ToDo activities (in ready and anticipable state), matching the key/value strings.

If no data is found matching the specified properties, null is returned.

9.5.4 Managing Activities for the User

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

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

An exception is thrown in the following cases:

- If the project name and/or node name does not exist and the activity cannot start
- If the project or model is in hidden status

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

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

An exception is thrown in the following cases:

- If the project name or node name does not exist and the activity cannot terminate
- If the project or model is in hidden status

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

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

An exception is thrown in the following cases:

- If the project name or node name does not exist and the activity cannot terminate
- If the project or model is in hidden status

9.6 Code Example

```
//****************************************************************/
//************** 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 9-2. User and Activities Code Example

Chapter 10. Bonita Pagination

For each available `get` method in the UserSession and AdminSession APIs there is a paginated version of the existing method: e.g., the getToDoListAllInstances method and the getToDoListAllInstancesAsPK method. The "AsPK" identifies a method retrieving a collection of primary Keys corresponding to the 30 activities available in the user ToDo list. Then, by means of the getToDoListAllInstancesByPK method, the user or the end client application retrieves a defined number of elements from the previous collection. Note that this method also has a parameter defining the value of the "offset" between two different calls.

`Collection` **`getModelsAsPK`** `()`

Return a Collection of BnProjectLightValue objects - the model list to be instantiated. This API call uses the initiator mapper check to verify the user executing the call is included within the Initiator Mapper. If the user is not a defined user, null is returned.

`Collection` **`getModelsByPK`** `(Collection list, int offset, int numrows)`

Return a Collection of BnProjectLightValue objects. The collection contains a list of models starting with "offset for "numrows".

`Collection` **`getModelsLightAsPK`** `()`

Get available workflow models. This method is similar to getModels () with the exception that the initiatorMapper check is not performed.

Return Collection of BnProjectLightValue objects reflecting the model list.

`Collection` **`getModelsLightByPK`** `(Collection list, int offset, int numrows)`

Get available workflow models. This method is similar to getModels () with the exception that the initiatorMapper check is not performed.

Return a Collection of BnProjectLightValue objects - the model list. The returned collection of BnProjectLightValue objects is based on the supplied "list", starting with "offset" for "numrows".

`Collection` **`getToDoListAsPK`** `(String projectName)`

Used to obtain all user activities from specific project (those in ready and anticipable state) and return a String Collection with the list of ToDo activities of the user for the specified project.

An exception is thrown in the following case:

- If projectName does not exist

`Collection` **`getToDoListByPK`** `(String projectName, Collection list, int offset, int numrows)`

Used to obtain all user activities from a specific project (those in ready and anticipable state) and return a String Collection with the list of ToDo activities of the user for the specified project. The returned String collection is based on the supplied “list”, starting with “offset” for “numrows”.

An exception is thrown in the following cases:

- If projectName does not exist

`Collection` **`getActivityListAsPK`** `(String projectName)`

Used to obtain all user activities from a specific project (those in executing and anticipating state) and return a String Collection with the list of active activities of the user for the specified project.

An exception is thrown in the following cases:

- If projectName does not exist

`Collection` **`getActivityListByPK`** `(String projectName, Collection list, int offset, int numrows)`

Used to obtain all user activities from a specific project (those in executing and anticipating state) and return a String Collection with the list of active activities of the user for the specified project. The returned String collection is based on the supplied “list”, starting with “offset” for “numrows”.

An exception is thrown in the following cases:

- If projectName does not exist

`Collection` **`getToDoListAllInstancesAsPK`** `()`

Used to obtain all ToDo user activities for all instances/cooperative projects (those in ready and anticipable state) and return a BnNodeValue Collection of the list of ToDo activities of the user for all instances.

If no activities are found a null list is returned.

`Collection` **`getToDoListAllInstancesByPK`** `(Collection list, int offset, int numrows)`

Used to obtain all ToDo user activities for all instances/cooperative projects (those in ready and anticipable state) and return a BnNodeValue Collection of the list of ToDo activities of the user for all instances. The returned BnNodeValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no activities are found a null list is returned.

Collection **getActivityListAllInstancesAsPK** ()

Used to obtain a list of executing user activities for all instances/cooperative projects (those in executing and anticipating state), and return a BnNodeValue Collection with the list of executing activities for the user for all instances.

If no activities are found a null list is returned.

Collection **getActivityListAllInstancesByPK** (Collection list, int offset, int numrows)

Used to obtain a list of executing user activities for all instances/cooperative projects (those in executing and anticipating state), and return a BnNodeValue Collection with the list of executing activities for the user for all instances. The returned BnNodeValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no activities are found a null list is returned.

Collection **getTerminatedListAllInstancesAsPK** ()

Used to obtain a subset of the terminated user activities for all instances/cooperative projects and return a BnNodeValue Collection containing the list of terminated activities for the user for all instances.

If no activities are found a null list is returned.

Collection **getTerminatedListAllInstancesByPK** (Collection list, int offset, int numrows)

Used to obtain a subset of the terminated user activities for all instances/cooperative projects and return a BnNodeValue Collection containing the list of terminated activities for the user for all instances. The returned BnNodeValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no activities are found a null list is returned.

Collection **getToDoListByPropertyAsPK** (String key, String value)

This API obtains a user activities “ToDo” BnNodeValue list matching the project property value (the key/value pair). Returned are those activities in ready and anticipable state only.

If no activities match the key/value or exist in the required state, a null list is returned.

`Collection` **`getToDoListByPropertyByPK`** `(String key, String value, Collection list, int offset, int numrows)`

This API obtains a user activities “ToDo” BnNodeValue list matching the project property value (the key/value pair). Returned are those activities in ready and anticipable state only. The returned BnNodeValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no activities match the key/value or exist in the required state, a null list is returned.

`Collection` **`getToDoListByPropertiesAsPK`** `(int operation, Hashtable properties)`

This API obtains a user activities “ToDo” BnNodeValue list matching the project properties (key/value pairs). The returned BnNodeValue list contains only those activities in the ready and anticipable state. This method is oriented to administrative workflow (workflow instances) and supports up to three properties (key/value pairs). The Operation constants values are: “hero.interfaces.Constants.AND”, or “hero.interfaces.Constants.OR”.

An exception is thrown in the following case:

- If more than 3 properties are entered (if more than 3 are required please contact your Bull HN representative.

`Collection` **`getToDoListByPropertiesByPK`** `(int operation, Hashtable properties, Collection list, int offset, int numrows)`

This API obtains a user activities “ToDo” list matching the project properties (key/value pairs). The returned BnNodeValue list contains only those activities in the ready and anticipable state. This method is oriented to administrative workflow (workflow instances) and supports up to three properties (key/value pairs). The Operation constants values are: “hero.interfaces.Constants.AND”, or “hero.interfaces.Constants.OR”.

The returned BnNodeValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no activities match the key/value or exist in the required state, a null list is returned.

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

This API obtains a user activities “ToDo” list matching the activity properties (key/value pairs). The returned BnNodeValue list contains only those activities in the ready and anticipable state.

If no activities match the key/value or exist in the required state, a null list is returned.

`Collection` **`getToDoListByActivityPropertyByPK`** `(String key, String value, Collection list, int offset, int numrows)`

This API obtains a user activities "ToDo" list matching the activity properties (key/value pairs). The returned BnNodeValue list contains only those activities in the ready and anticipable state. The returned BnNodeValue collection is based on the supplied "list", starting with "offset" for "numrows".

If no activities match the key/value or exist in the required state, a null list is returned.

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

This API obtains a BnNodeValue Collection of executing user activities names matching the property value (key/value pair). The list contains only those activities in executing and anticipating state.

If no activities match the key/value or exist in the required state, a null list is returned.

`Collection` **`getActivityListByPropertyByPK`** `(String key, String value, Collection list, int offset, int numrows)`

This API obtains a BnNodeValue Collection of executing user activities names matching the property value (key/value pair). The list contains only those activities in executing and anticipating state. The returned BnNodeValue collection is based on the supplied "list", starting with "offset" for "numrows".

If no activities match the key/value or exist in the required state, a null list is returned.

`Collection` **`getProjectInstancesNamesAsPK`** `(String projectName, String version)`

Return a String Collection containing the names of project instances.

An exception is thrown in the following cases:

- If projectName does not exist
- If the specified name and version is not found

`Collection` **`getProjectInstancesNamesAsPK`** `(String projectName)`

Return a String Collection containing the names of project instances using the default version.

If no project instances are found, a null list is returned.

An exception is thrown in the following case:

- If projectName does not exist

`Collection` **`getProjectInstancesNamesByPK`** `(String projectName, Collection list, int offset, int numrows)`

Return a String Collection containing the names of project instances using the specified version. The returned String collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no project instances are found, a null list is returned.

`Collection` **`getProjectInstancesAsPK`** `(String projectName, String version)`

Return a BnProjectValue Collection containing a list of project instances for the specified version.

If no project instances are found, a null list is returned.

An exception is thrown in the following case:

– If projectName and/or version does not exist

`Collection` **`getProjectInstancesAsPK`** `(String projectName)`

Return a BnProjectValue Collection containing a list of project instances for the default version.

An exception is thrown in the following case:

– If projectName does not exist

`Collection` **`getProjectInstancesByPK`** `(String projectName, Collection list, int offset, int numrows)`

Return a BnProjectValue Collection containing a list of project instances for the specified version. The returned BnProjectValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no project instances are found, a null list is returned.

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

Return a BnProjectValue Collection containing a list of project instances meeting the key/value property.

If no Instances are found, null is returned.

`Collection` **`getInstancesByPropertyByPK`** `(String key, String value, Collection list, int offset, int numrows)`

Return a BnProjectValue Collection containing a list of project instances meeting the key/value property. The returned BnProjectValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no Instances are found, null is returned.

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

Return a String Collection containing a list of project instances based on the supplied key/value pair.

If no instances are found, null is returned.

`Collection` **`getInstancesByPropertyNamesByPK`** `(String key, String value, Collection list, int offset, int numrows)`

Return a BnProjectValue Collection containing a list of project instances meeting the key/value property.

If no instances are found, null is returned.

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

Return a BnProjectValue Collection containing a list of projects based on the supplied key/value pair.

If no projects are found, null is returned.

`Collection` **`getProjectsByPropertyByPK`** `(String key, String value, Collection list, int offset, int numrows)`

Return a BnProjectValue Collection containing a list of projects based on the supplied key/value pair. The returned BnProjectValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no projects are found, null is returned.

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

Return a String Collection containing a list of project names based on the supplied key/value pair.

If no projects are found, null is returned.

`Collection` **`getProjectsByPropertyNamesByPK`** `(String key, String value, Collection list, int offset, int numrows)`

Return a String Collection containing a list of project names based on the supplied key/value pair. The returned String collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no projects are found, null is returned.

`Collection` **`getInstancesActivityTodoListAsPK`** `(String projectName, String nodeName)`

Return a BnNodeLightValue Collection containing a subset of activities model instances assigned to this user.

If no model instances are found, null is returned.

`Collection` **`getInstancesActivityTodoListByPK`** `(String projectName, String nodeName, Collection list, int offset, int numrows)`

Return a BnNodeLightValue Collection containing a subset of activities model instances assigned to this user. The returned BnNodeLightValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no model instances are found, null is returned.

`Collection` **`getInstancesActivityTerminatedAsPK`** `(String projectName, String nodeName)`

Return a BnNodeLightValue Collection containing a subset of activities model instances terminated by this user.

If no model instances are found, null is returned.

`Collection` **`getInstancesActivityTerminatedByPK`** `(String projectName, String nodeName, Collection list, int offset, int numrows)`

Return a BnNodeLightValue Collection containing a subset of activities model instances terminated by this user. The returned BnNodeLightValue collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no model instances are found, null is returned.

`Collection` **`getUserInstancesProjectNodesAsPK`** `(String projectName)`

Return a String Collection containing the names of activities in the ready, anticipable, and executing state for the current user instances.

If no instances are found, null is returned.

An exception is thrown in the following case:

- If projectName does not exist

`Collection` **`getUserInstancesProjectNodesByPK`** `(String projectName, Collection list, int offset, int numrows)`

Return a String Collection containing the names of activities in the ready, anticipable, and executing state for the current user instances. The returned String collection is based on the supplied “list”, starting with “offset” for “numrows”.

If no instances are found, null is returned.

An exception is thrown in the following case:

- If projectName does not exist

Chapter 11. Bonita Entities

Many entry points in 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).

To directly use the internal EJB through the remote or local interfaces (this choice is not recommended), each of these entities may be accessed using its name suffixed by hero.interfaces. Many entry points in 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).

To directly use the internal EJB thru the remote or local interfaces (this choice is not recommended), each of these entities may be accessed using its name suffixed by hero.interfaces.

11.1 Entity Diagrams

11.1.1 Global Diagram

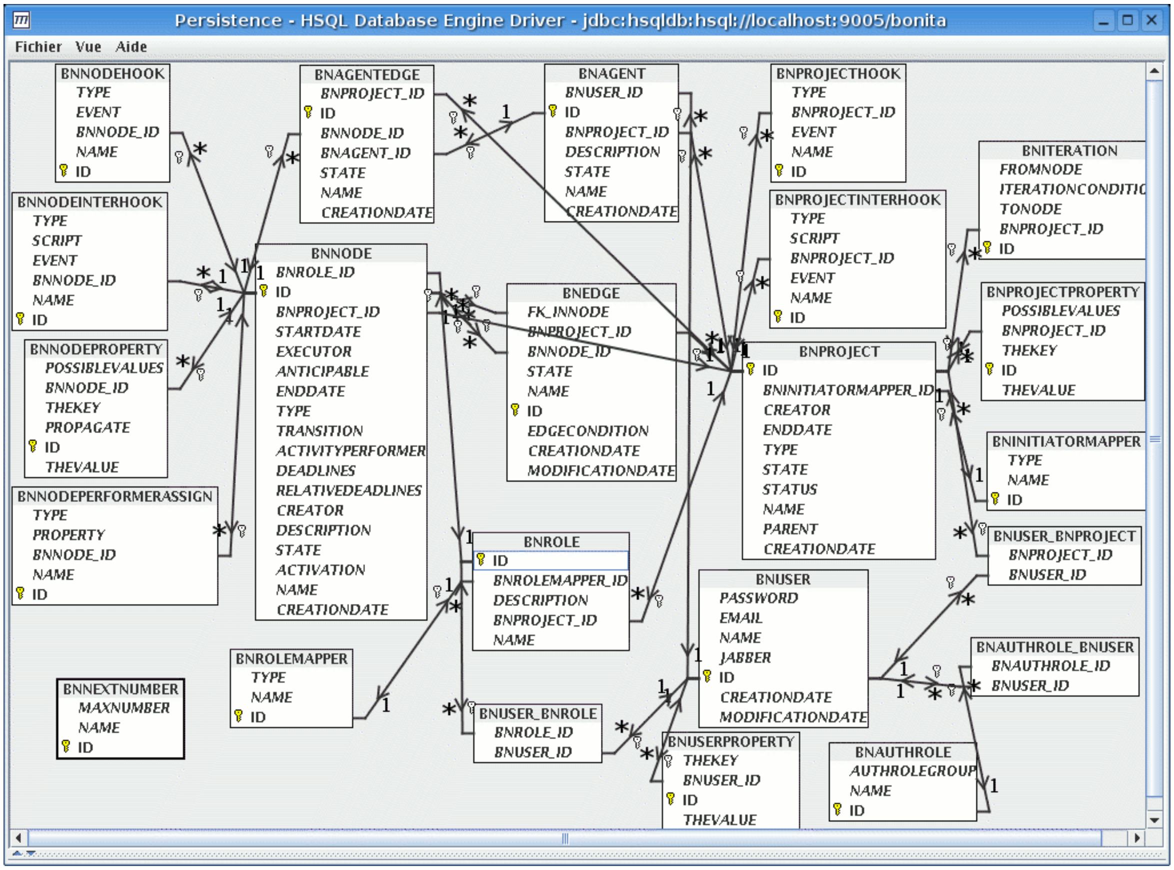


Figure 11-1. Global Diagram

11.1.2 Diagram Focused on Project Entity Relations

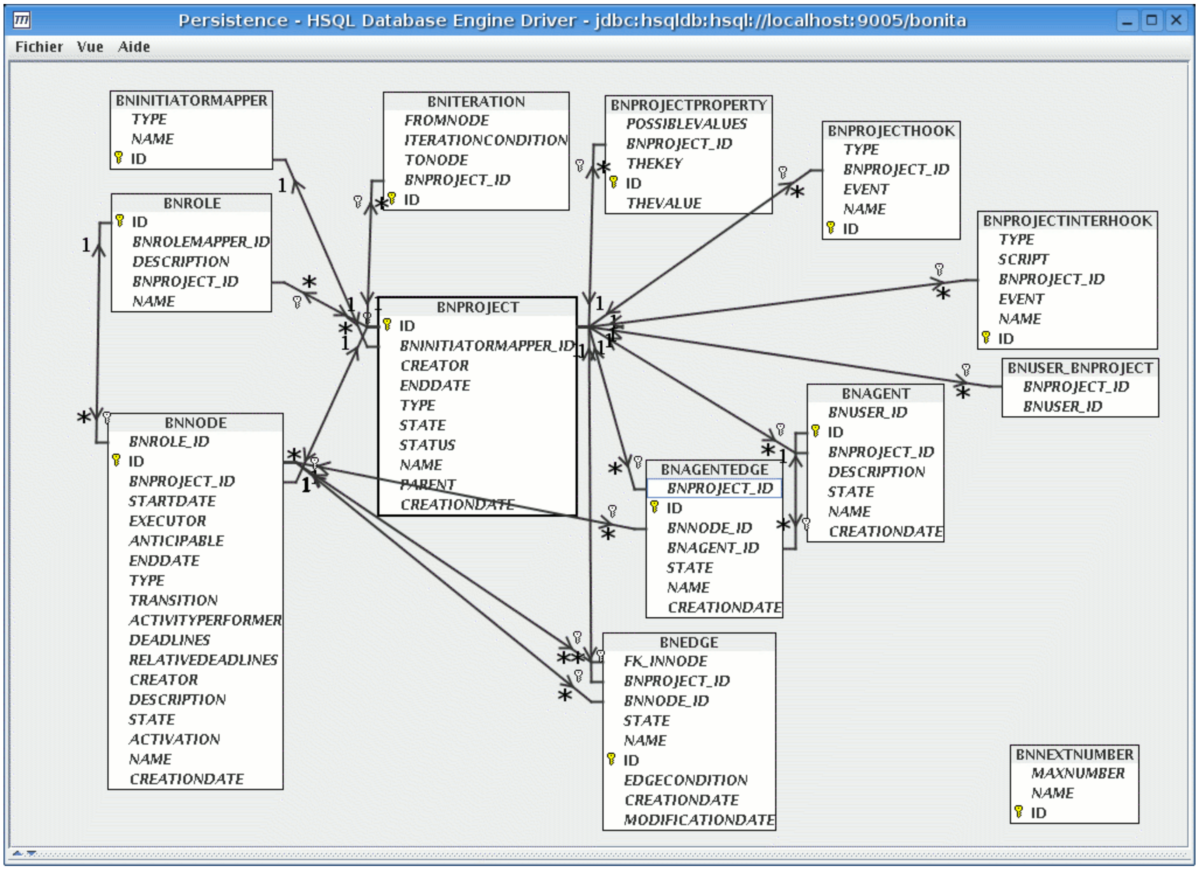


Figure 11-2. Project Entity Diagram

11.1.3 Diagram Focused on Node Entity Relations

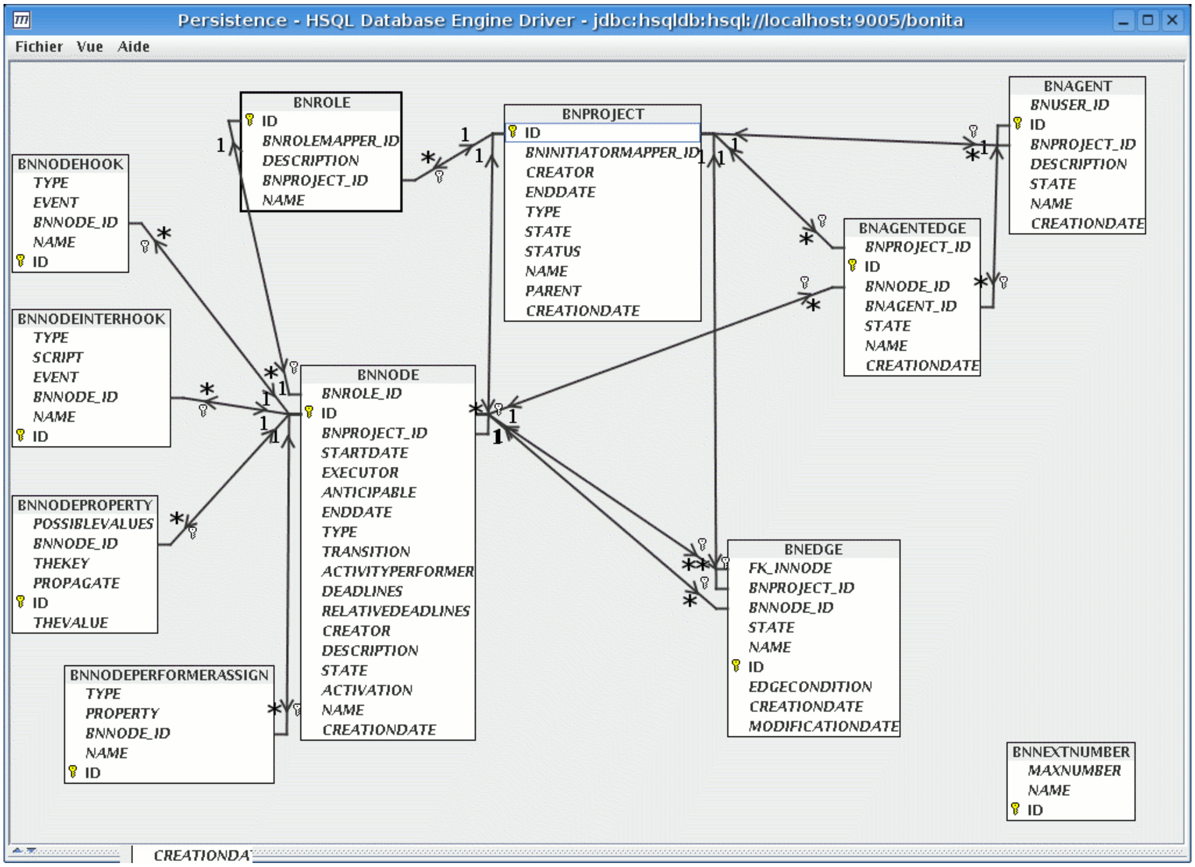


Figure 11-3. Node Entity Diagram

11.1.4 Diagram Focused on User-Role Entities Relations

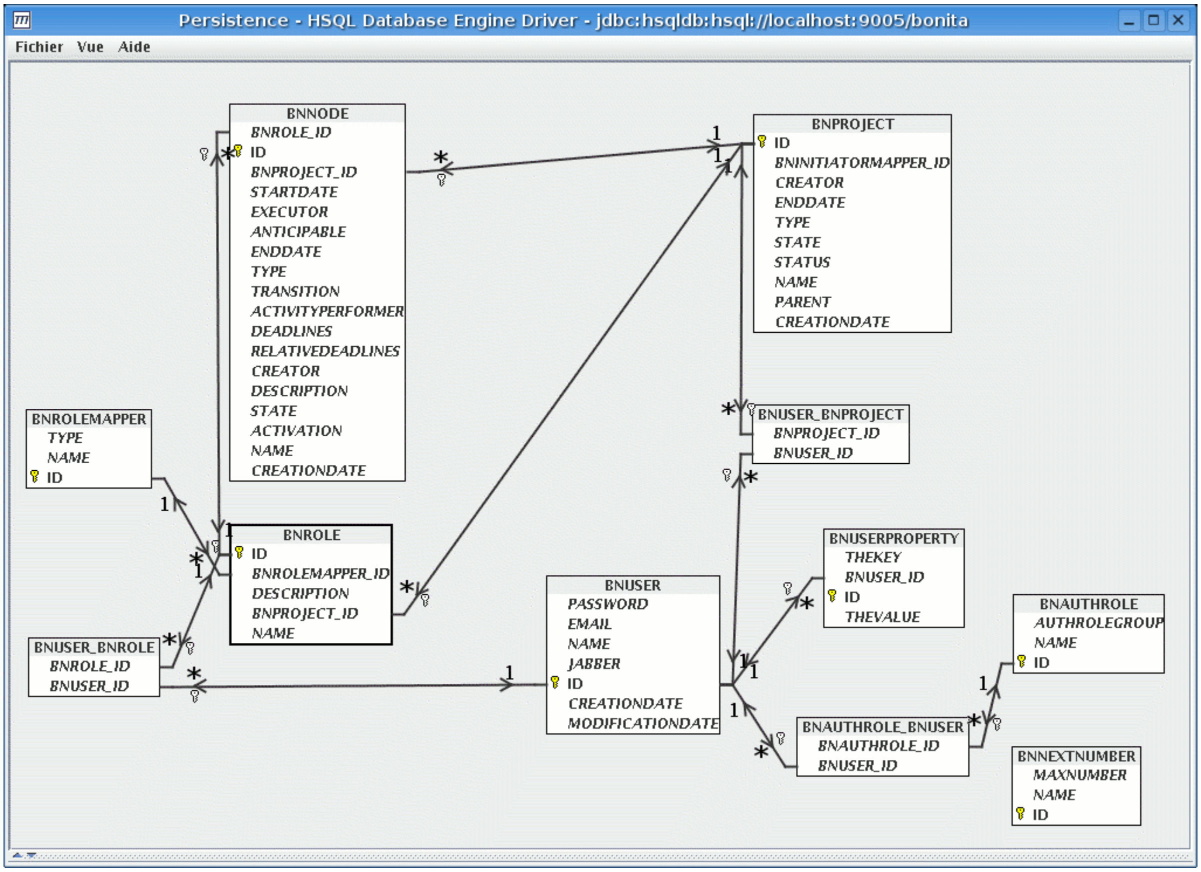


Figure 11-4. User Role Entity Diagram

11.2 Entities Attributes

11.2.1 BnAuthRoleValue

TYPE	ATTRIBUTE	MEANING
String	id	Auth Role ID
boolean	idHasBeenSet	Default = false
String	name;	Auth Role Name
boolean	nameHasBeenSet	Default = false
String	bnRoleGroup;	Auth Role Group Role name
boolean	bnRoleGroupHasBeenSet	Default = false
hero.interfaces.BnAuthRolePK	pk;	Auth Role Primary Key

Table 11-1. BnAuthRoleValue Attributes

11.2.2 BnEdgeValue

TYPE	ATTRIBUTE	MEANING
String	id;	Edge ID
boolean	idHasBeenSet	Default = false
String	name;	Edge Name
boolean	nameHasBeenSet	Default = false
int	state;	Edge State
boolean	stateHasBeenSet	Default = false
String	condition;	Edge Condition
boolean	conditionHasBeenSet	Default = false
java.sql.Date	creationDate;	Date edge created
boolean	creationDateHasBeenSet	Default = false
java.sql.Date	modificationDate;	Date edge modified
boolean	modificationDateHasBeenSet	Default = false
hero.interfaces.BnNodeValue	InBnNode;	Edge Input (from) node
boolean	InBnNodeHasBeenSet	Default = false
hero.interfaces.BnNodeValue	OutBnNode;	Edge Output (to) node
boolean	OutBnNodeHasBeenSet	Default = false
hero.interfaces.BnEdgePK	pk;	Edge Primary Key

Table 11-2. BnEdgeValue Attributes

11.2.3 BnIterationValue

TYPE	ATTRIBUTE	MEANING
String	id;	Iteration ID
boolean	idHasBeenSet	Default = false
String	fromNode;	Iterating from node
boolean	fromNodeHasBeenSet	Default = false
String	toNode;	Iterating to node
boolean	toNodeHasBeenSet	Default = false
String	condition;	Iteration Condition
boolean	conditionHasBeenSet	Default = false
hero.interfaces.BnIterationPK	pk;	Iteration Primary Key

Table 11-3. BnIterationValue Attributes

11.2.4 BnNodeHookValue

TYPE	ATTRIBUTE	MEANING
String	id;	Node Hook Id
boolean	idHasBeenSet	Default = false
String	name;	Node hook name
boolean	nameHasBeenSet	Default = false
String	event;	Node Hook event
boolean	eventHasBeenSet	Default = false
int	type;	Node Hook type
boolean	typeHasBeenSet	Default = false
hero.interfaces.BnNodeHookPK	pk	Node Hook Primary Key

Table 11-4. BnNodeHookValue Attributes

11.2.5 BnNodeInterHookValue

TYPE	ATTRIBUTE	MEANING
String	id	Node InterHook ID
boolean	idHasBeenSet	Default = false
String	name;	Node InterHook name
boolean	nameHasBeenSet	Default = false
String	event	Node InterHook event
boolean	eventHasBeenSet	Default = false
int	type	Node InterHook type
boolean	typeHasBeenSet	Default = false
String	script	Node InterHook script
boolean	scriptHasBeenSet	Default = false
hero.interfaces.BnNodeInterHookPK	pk	Node InterHook Primary Key

Table 11-5. BnNodeInterHookValue Attributes

11.2.6 BnNodePerformerAssignValue

TYPE	ATTRIBUTE	MEANING
String	id	Node Performer Id
boolean	idHasBeenSet	Default = false
String	name	Node Performer name
boolean	nameHasBeenSet	Default = false
int	type;	Node Performer type (i.e. callback)
boolean	typeHasBeenSet	Default = false
String	propertyName	Used with property assignment
boolean	propertyNameHasBeenSet	Default = false
hero.interfaces.BnNodePerformerAssignPK	pk	Node Performer Primary Key

Table 11-6. BnNodePerformerAssignValue Attributes

11.2.7 BnNodePropertyValue

TYPE	ATTRIBUTE	MEANING
String	id;	Node Property Id
boolean	idHasBeenSet	Default = false
String	theKey;	Node Property Key name
boolean	theKeyHasBeenSet	Default = false
String	theValue;	Node Property Value
boolean	theValueHasBeenSet	Default = false
boolean	propagate;	True = propagate
boolean	propagateHasBeenSet	Default = false
hero.interfaces.BnNodePropertyPK	pk;	Node Property Primary Key

Table 11-7. BnNodePropertyValue Attributes

11.2.8 BnNodeValue

TYPE	ATTRIBUTE	MEANING
String	id;	Node Id
boolean	idHasBeenSet	Default = false
int	type;	Node type
boolean	typeHasBeenSet	Default = false
int	state;	Node state
boolean	stateHasBeenSet	Default = false
boolean	anticipable;	Set by setNodeAnticipable
boolean	anticipableHasBeenSet	Default = false
String	name;	Node name
boolean	nameHasBeenSet	Default = false
String	reference	Node SubProcess use
boolean	referenceHasBeenSet	Default = false
String	description;	Node description
boolean	descriptionHasBeenSet	Default = false
String	activityPerformer;	Performer name
boolean	activityPerformerHasBeenSet	Default = false
Date	startDate	Node start date
boolean	startDateHasBeenSet	Default = false

Table 11-8. BnNodeValue Attributes (1 of 2)

TYPE	ATTRIBUTE	MEANING
Date	endDate	Node end date
boolean	endDateHasBeenSet	Default = false
Collection	deadlines	Node deadlines
boolean	deadlinesHasBeenSet	Default = false
Collection	relativeDeadlines	Node relative deadlines
boolean	relativeDeadlinesHasBeenSet	Default = false
String	creator	Node creator
boolean	creatorHasBeenSet	Default = false
String	executor	Node executor
boolean	executorHasBeenSet	Default = false
Date	creationDate	Node creation date
boolean	creationDateHasBeenSet	Default = false
BnRoleValue	BnRole	Node Role value
boolean	BnRoleHasBeenSet	Default = false
BnNodePerformerAssignValue	BnNodePerformerAssign	Node Performer
boolean	BnNodePerformerAssignHasBeenSet	Default = false
BnProjectLightValue	BnProject	Node Light value
boolean	BnProjectHasBeenSet	Default = false
Collection	BnProperties	Node Properties
Collection	BnHooks	Node Hooks
Collection	BnInterHooks	Node Interhooks
BnNodePK	primaryKey	Node Primary Key

Table 11-10. BnNodeValue Attributes (2 of 2)

11.2.9 BnProjectHookValue

TYPE	ATTRIBUTE	MEANING
String	id;	Project Hook Id
boolean	idHasBeenSet	Default = false
String	name;	Hook name
boolean	nameHasBeenSet	Default = false
String	event;	Hook event
boolean	eventHasBeenSet	Default = false
int	type;	Hook type
boolean	typeHasBeenSet	Default = false
hero.interfaces.BnProjectHookPK	pk;	Project Hook Primary key

Table 11-9. BnProjectHookValue Attributes

11.2.10 BnProjectInterHookValue

TYPE	ATTRIBUTE	MEANING
String	id;	Project InterHook Id
boolean	idHasBeenSet	Default = false
String	name;	Project InterHook name
boolean	nameHasBeenSet	Default = false
String	event;	Project InterHook event
boolean	eventHasBeenSet	Default = false
int	type;	Project InterHook type
boolean	typeHasBeenSet	Default = false
String	script;	Project Interhook script
boolean	scriptHasBeenSet	Default = false
hero.interfaces.BnProjectInterHookPK	pk;	Project InterHook Primary Key

Table 11-10. BnProjectInterHookValue Attributes

11.2.11 BnProjectPropertyValue

TYPE	ATTRIBUTE	MEANING
String	id;	Property ID
Boolean	idHasBeenSet	Default = false
String	theKey;	Property Key
Boolean	theKeyHasBeenSet	Default = false
String	theValue;	Property value
Boolean	theValueHasBeenSet	Default = false
Collection	possibleValues	Possible values (allowed)
boolean	possibleValuesHasBeenSet	Default = false
hero.interfaces.BnProjectPropertyPK	pk;	Property Primary Key

Table 11-11. BnProjectPropertyValue Attributes

11.2.12 BnProjectValue

TYPE	ATTRIBUTE	MEANING
String	id;	Project Value ID
boolean	idHasBeenSet	Default = false
String	parent;	Parent name
boolean	parentHasBeenSet	Default = false
String	name;	Project name
boolean	nameHasBeenSet	Default = false
String	version	Project version
boolean	versionHasBeenSet	Default = false
String	status	Project status
boolean	statusHasBeenSet	Default = false
String	type	Project type
boolean	typeHasBeenSet	Default = false
String	creator;	Project creator
boolean	creatorHasBeenSet	Default = false
int	state;	Project state
boolean	stateHasBeenSet	Default = false
java.util.Date	creationDate;	Project creation date
boolean	creationDateHasBeenSet	Default = false
java.util.Date	endDate	Project end date
boolean	endDateHasBeenSet	Default = false
Collection	BnUsers	Project users
Collection	BnRoles	Project roles
BnInitiatorMapperValue	BnInitiatorMapper	Initiator mapper
boolean	BnInitiatorMapperHasBeenSet	Default = false
Collection	BnNodes	Project nodes
Collection	BnEdges	Project edges
Collection	BnAgents	Project agents
Collection	BnAgentEdges	Project agent edges
Collection	BnProperties	Project properties
Collection	BnIterations	Project iterations
Collection	BnHooks	Project hooks
Collection	BnInterHooks	Project interhooks
hero.interfaces.BnProjectPK	pk;	Project Primary key

Table 11-12. BnProjectValue Attributes

11.2.13 BnRoleMapperValue

TYPE	ATTRIBUTE	MEANING
String	id;	Role Mapper ID
boolean	idHasBeenSet	Default = false
String	name;	Role Mapper name
boolean	nameHasBeenSet	Default = false
int	type;	Role Mapper type
boolean	typeHasBeenSet	Default = false
hero.interfaces.BnRoleMapperPK	pk;	Role Mapper Primary Key

Table 11-13. BnRoleMapperValue Attributes

11.2.14 BnRoleValue

TYPE	ATTRIBUTE	MEANING
String	id;	Role value ID
boolean	idHasBeenSet	Default = false
String	description;	Role value description
boolean	descriptionHasBeenSet	Default = false
String	name;	Role value name
boolean	nameHasBeenSet	Default = false
hero.interfaces.BnRoleMapperValue	BnRoleMapper;	Role value mapper
boolean	BnRoleMapperHasBeenSet	Default = false
hero.interfaces.BnRolePK	pk;	Role value Primary Key

Table 11-14. BnRoleValue Attributes

11.2.15 BnUserPropertyValue

TYPE	ATTRIBUTE	MEANING
String	id;	User Property ID
boolean	idHasBeenSet	Default = false
String	theKey;	User Property key
boolean	theKeyHasBeenSet	Default = false
String	theValue;	User Property value
boolean	theValueHasBeenSet	Default = false
hero.interfaces.BnUserPropertyPK	pk;	User Property Primary Key

Table 11-15. BnUserPropertyValue Attributes

11.2.16 BnUserValue

TYPE	ATTRIBUTE	MEANING
String	id;	User ID
boolean	idHasBeenSet	Default = false
String	name;	User name
boolean	nameHasBeenSet	Default = false
String	password;	User password
boolean	passwordHasBeenSet	Default = false
String	email;	User email
boolean	emailHasBeenSet	Default = false
String	jabber;	User jabber name
boolean	jabberHasBeenSet	Default = false
java.sql.Date	creationDate;	User creation date
boolean	creationDateHasBeenSet	Default = false
java.sql.Date	modificationDate;	User modification date
boolean	modificationDateHasBeenSet	Default = false
Collection	BnProjects	User projects
Collection	BnRoles	User roles
Collection	BnAuthRoles	User authorized roles
hero.interfaces.BnUserPK	pk;	User Primary Key

Table 11-16. BnUserValue Attributes