BONITA
Application Programming Interface

* This is a preliminary version of Bonita Application Programming Interface. You can find more information of Bonita's API in your $BONITA_HOME/javadoc installation directory.

Index

Introduction

BONITA is a workflow system featuring a lot of innovative features like activities that can start in anticipation, awareness infrastructure allowing users to be notified of any events occurring during the execution in a given process , or automatic activation of user's code according to a defined activity life cycle.

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

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

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

.

1 Concepts

1.1 Terminology

- A **process** is a set of activities. In BONITA, the term project is also used.
- An **activity** is an atomic unit of work. In BONITA, activities are also termed **Node*s.***
- **A transition** is a dependency expressing an order constraint between two activities. In BONITA, transitions are also termed **Edge**s.

1.2 Process

1.2.1 Life Cycle

BONITA has a very simple process life cycle

- A process is **initial** once it has been created. As soon as the process is in this state, it can be controlled using both User API & Project API. The User API allows to monitor the execution of the process. Whenever the first activity has been started using the User API, the process goes to **executing** state. The execution of the process is performed by the BONITA enactment engine, under control of applications thru the use of the User API.
- A process is **executing** as soon as the first activity has started. While being executing, the process definition still can be modified using the Project API. When all activities are terminated, the process stays in state **executing**. It still can be modified , for example new activities can be added.
- A process is **terminated** once it has been explicitly terminated by an application thru the User API. In **terminated** state, the process definition cannot be modified any more.

To allow the reuse of process definition across several process instances, BONITA builds on this simple life cycle to provide process cloning and process model instantiation.

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

Just after the cloning operation:

- The process clone has the same set of activities than the original one. All activities are in initial state, and without any properties being defined. Each activity is allocated to the same role than in the original process. All activities have the same hooks (see Hooks description in 1.3 below) than the ones in the original process.
- The process clone has no properties defined.
- No users are associated to the process clone.
- The process clone can be controlled without any restriction thru the User and Project APIs.

A process model is a specific definition of a process that can be instantiated many times. A process model keeps track of all its instances, that is all instances of such a process can be retrieved thru the User API.

Just after the instantiation operation:

- The process instance has the same set of activities than the process model, each activity allocated to the same role than in the model. All activities are in initial state, and have the same properties than in the model, without any associated value. All activities have the same hooks and the same transition conditions than the ones in the original process.
- The process properties are the same than the model's one, but are not initialised.
- The users associated to the process are the same than in the process model, and have the same associated roles.
- The process instance can be controlled without any restriction thru the User and Project APIs.

1.2.2 Relationship to Users

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

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

Users assuming the Admin role can modify the definition of the process.

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

1.2.3 Process names

Names are given to processes at creation time. There are no restrictions on the number of characters used to name process.

Process instances are named automatically be BONITA, which derives the instance name from the model name as follows:

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

1.3 Activities

1.3.1 Activities basics

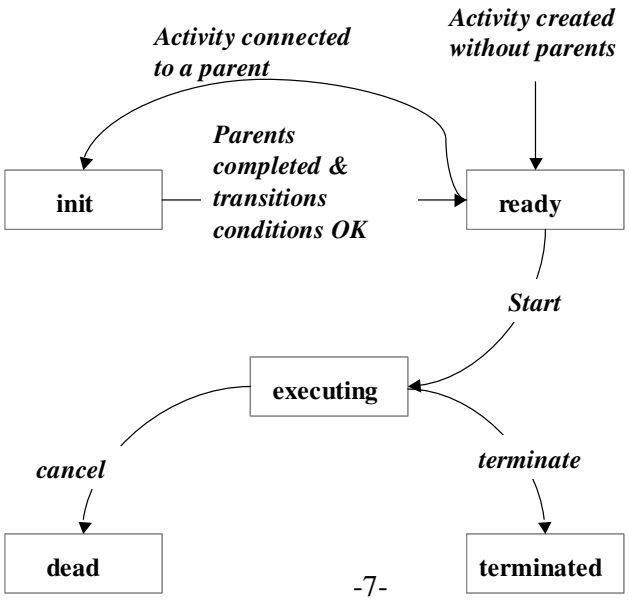
The activity is the basic unit of work within a process.

- Execution of an activity can be automatic; in this case the BONITA enactment engine will start it as soon as the applicable transitions from preceding activities are successfully evaluated.
- Alternatively, the execution of an activity can be manual that is the BONITA enactment engine will not start an activity until some application has explicitly started it thru the User API.

The life cycle of an activity is as follows.

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

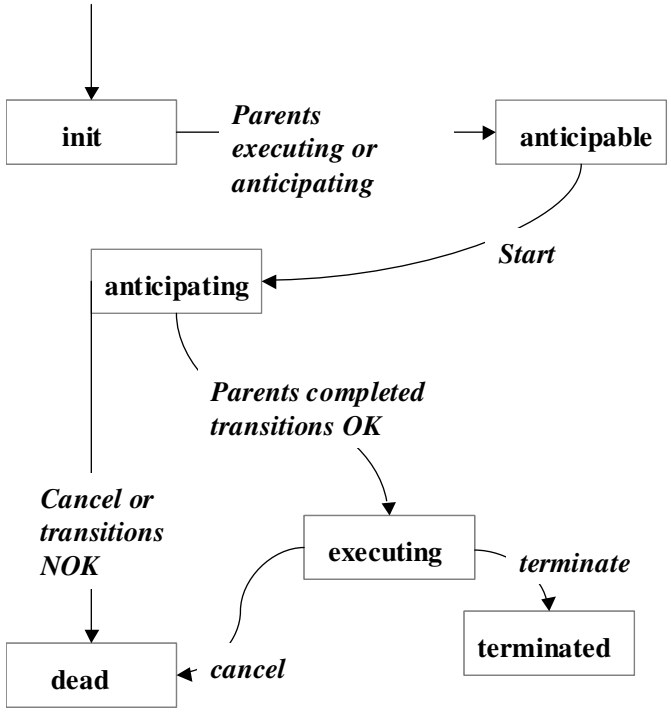
The activity life cycle is figured below for activities that cannot be anticipated.



Recall that for automatic activities, BONITA will automatically

- transition the state from ready to executing ,
- launch the executing hook
- terminate the activity whenever the executing hook has complete

The activity life cycle is figured below for activities that can be anticipated.



Recall that for automatic activities, BONITA will automatically

- transition the state from anticipable to anticipating ,
- launch the anticipating hook
- transition the state from anticipating to executing whenever all the parents complete
- terminate the activity whenever the executing hook has completed

1.3.2 Concept of Hooks

BONITA has the concept of Hooks, which are user defined logic that can be triggered at some defined points in the life of the activity. These defined points are

- Before Start hook is called just before after the activity starts. The Before Start hook is not considered to be in the same transaction than the activity. The Before Start hook is not triggered for activities that can be anticipated
- After Start hook is called just after the activity has started. It is considered to be in the same transaction than the activity.
- Cancel hook is called before cancelling an activity
- Before Terminate hook is called just before after the activity terminates. The Before Terminate hook is considered to be in the same transaction than the activity.

- After Terminate hook is called just after the activity has terminated. It is not considered to be in the same transaction than the activity.
- Anticipating hook is called when an automatic activity is started, only if the activity is anticipable. It is considered to be in the same transaction than the activity.

1.3.3 Relationship to Users

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

1.4 User

BONITA manages users in a specific base.

This base allows to store properties for a given user. Properties are (key, value) pairs where both key and values are java.lang.String variables. The application can set and retrieve properties using the User interface. BONITA makes use of specific user properties in order to store the User preferences.

1.4.1 Relationship to processes

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

Two scenarios allow to associate a User with a process.

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

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

1.4.2 Authentication scenario

At the moment, User authentication is performed against BONITA specific database. It is also possible to authenticate against an LDAP directory.

1.5 Role

BONITA manages roles on a per Process basis. This allows to have different semantics associated to the same role name in the scope of two different processes.

Activities are associated to roles, that is each activity can be taken over only by a user assuming a given role. There is a single role associated to each activity.

Roles are associated to Users, where a User can assume several roles in the scope of a given process.

2 Project interface

2.1 Principle

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

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

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

2.2 Initiating the ProjectSessionBean

2.2.1 Initiating with the fresh instance creation option

Void initProject (`java.lang.String projectName)`

The Project interface is initialised with the given *projectName*. All subsequent interface methods will deal with the corresponding project.

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

2.2.2 Initiating with the clone project creation option

Void initProject (`java.lang.String oldProject, java.lang.String newProject)`

The Project interface is initialised after oldProject was cloned. This interface is initialised with the given *newProject* project name. All subsequent interface methods will deal with the corresponding *newProject*.

2.2.3 Initiating with the instantiate project creation option

Void instantiateProject (j`ava.lang.String projectName)`

The Project interface is initialised after new project instance was created. This interface is initialised with the new project instance name (see process names description in 1.2.3). All subsequent interface methods will deal with the corresponding project instance.

2.3 Managing project

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

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

2.3.1 Project attributes

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

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

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

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

2.3.2 Getting the name of a project

```
java.lang.String  getName();
```

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

2.3.3 Getting the name of a project's creator

```
java.lang.String  getCreator();
```

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

2.3.4 Properties

```
void  setProperty (java.lang.String key, java.lang.String value)
```

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

```
java.util.Collection  getProperties();
```

Returns all the properties existing for this project. Properties are returned as BnProjectPropertyValue

```
void  deletteProperty(java.lang.String key);
```

Delete a property of an existing project.

2.4 Managing users

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

```
java.util.Collection  getUsers();
```

Users are returned as Strings

2.4.2 Adding a user to a project

```
void  addUser(java.lang.String username);
```

2.4.3 Checking whether a user is part of a project

```
boolean  containsUser(java.lang.String username);
```

2.5 Managing roles

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

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

2.5.1 Declaring a new role in the project

```
void  addRole java.lang.String roleName, java.lang.String description);
```

2.5.2 Allocating a role to a User

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

```
void  setUserRole (java.lang.String userName, java.lang.String roleName);

void  unsetUserRole (java.lang.String userName, java.lang.String roleName);
```

2.5.3 Getting the list of roles that an User can assume in the scope of a project

```
java.util.Collection  getUserRoles (java.lang.String userName);
```

2.5.4 Associating an activity with a role

Only a single role can take over a given activity.

2.6 Edge (transitions between activities)

2.6.1 Adding an edge to an activity

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

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

```
java.lang.String  addEdge(java.lang.String in, java.lang.String out);
```

The two activities named in and out will be connected by a new eddge. The method returns the name of the newly created edge.

```
java.lang.String  addEdge(java.lang.String name,java.lang.String in,
java.lang.String out);
```

The two activities named in and out will be connected by a new edge. The newly created edge will be named according to the name passed as input parameter.

2.6.2 Deleting an edge

```
Void  deleteEdge(java.lang.String name);
```

The edge named with the parameter name will be deleted.

2.6.3 Getting connected activities from an edge

```
java.lang.String  getEdgeInNode(java.lang.String edgeName) ;
```

Get back the name of the inbound node of the given edgeName.

```
java.lang.String  getEdgeOutNode(java.lang.String edgeName) ;
```

Get back the name of the outbound node of the given edgeName.

2.6.4 Setting a condition on an edge

```
Void  setEdgeCondition(java.lang.String edge, java.lang.String condition);
```

The condition is passed as a string expressing a condition in a scripting language.

2.6.5 Getting the condition on an edge

```
java.lang.String  getEdgeCondition(java.lang.String edge);
```

2.6.6 Getting all existing edges in a project

```
java.util.Collection    getEdgesNames();
```

Returns all the existing edges in the project.

2.6.7 Getting all existing edges for an activity

```
java.util.Collection    getNodeInEdges();
```

Returns all the existing edges inbound for a given node .

```
java.util.Collection    getNodeOutEdges();
```

Returns all the existing edges outbound for a given node .

2.6.8 Reading an edge as a Java Object

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

2.6.9 Changing the state of an Edge

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

2.7 Hooks

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

Must document in a central place the different possible scripting strategies

Hooks can be coded in a scripting language, or as java library. Therefore, the hook interface is divided in two sets. Script hooks are called interactive Hooks, hence all calls related to them will contain “Inter” in their name.

Hooks can be defined at the project level. Such hooks will be activated for every activity contained in the project.

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

2.7.1 Adding a hook to a project

```
Void  addHook    (java.lang.String hookName,    java.lang.String eventName,
int hookType)

Void  addInterHook  (java.lang.String hookName,  java.lang.String eventName,
int hookType, java.lang.String value)
```

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

2.7.2 Deleting a hook from a project

```
Void  deleteHook (java.lang.String hookName)

Void  deleteInterHook (java.lang.String hookName)
```

The hook with name hookName will be deleted.

2.7.3 Adding a hook to an activity

```
Void  addNodeHook   (java.lang.String nodeName,   java.lang.String hookName,
java.lang.String eventName, int hookType)

Void  addNodeInterHook                          (java.lang.String nodeName,
java.lang.String hookName,     java.lang.String eventName,     int hookType,
java.lang.String value)
```

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

3 User interface

3.1 Principle

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

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

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

3.2 Getting the name of the User

```
Java.lang.String getUser()
```

Returns the name of the User who created the Interface as a java.lang.String.

3.3 Getting the list of existing properties for the User

Java.util.Collection getUserProperties ()

Returns the properties defined for a given User. Properties are returned as a collection of BnUserProperty .

3.4 Setting a property for the User

Void setUserProperty (java.lang.String key, java.lang.String value)

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

3.5 Deleting a property of the User

Void deleteUserProperty (`java.lang.String key)`

Delete the property whose name is *key* .

3.6 Getting the list of projects for the user

`java.util.Collection getProjectListNames ()`

Returns the list of projects in which the current user is registered. Projects name are returned as java.lang.String.

3.7 Getting the list of activities for a given project

`java.util.Collection geActivityList (java.lang.String projectName)`

Returns all user activities from the specified project . Only activities in **executing** or **anticipating** state are given back. Activities are returned back as BnNodeLocal objects.

3.8 Getting the list of activities waiting to start for a given project

Void getToDoList ((j`ava.lang.String projectName)`

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

3.9 Getting information about an activity

`hero.interfaces.BnNodeValue` getNode (j`ava.lang.String projectName, java.lang.String nodeName)`

3.10 Starting an activity

Void startActivity (java.lang.String projectName, java.lang.String nodeName)

Starts the activity that has the given name in the scope of the process with the given name.

3.11 Terminating an activity

Void terminateActivity (java.lang.String projectName, java.lang.String nodeName)

Terminates the activity that has the given name in the scope of the process with the given name.

3.12 Cancelling an activity

Void cancelActivity (java.lang.String projectName, java.lang.String nodeName)

Cancels the activity that has the given name in the scope of the process with the given name.

3.13 Terminating a process

Void terminate (java.lang.String projectName)

Terminate the process that has the given name.

3.14 Removing a process

Void removeProcess (java.lang.String projectName)

Remove the process that has the given name.