BONITA Workflow Cooperative System
Application Programming Interface

(Version 2.0)

Christophe Loridan
Miguel Valdés Faura

BULL R&D

* 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

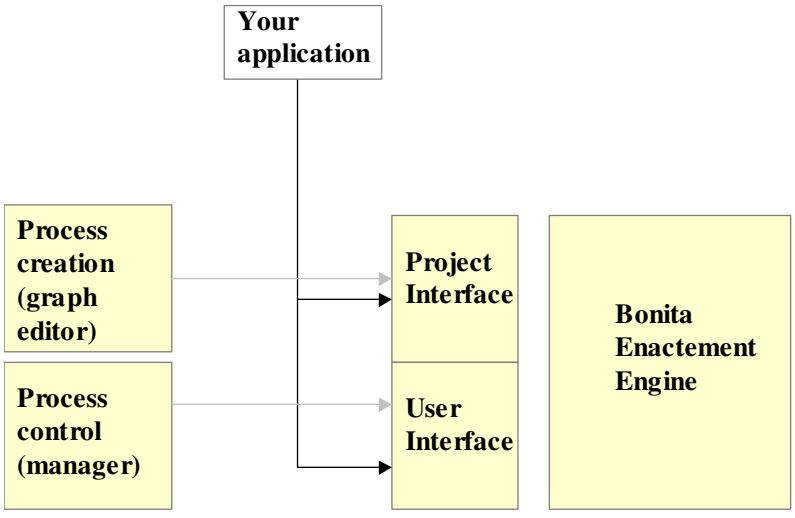
BONITA is a workflow system featuring a lot of innovative features like activities that can start in anticipation, awareness infrastructure allowing users to be notified of any events occurring during the execution in a given process , or automatic activation of user's code according to a defined activity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

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

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

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

The drawing hereunder shows the different components of BONITA



1 Concepts

1.1 Terminology

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

1.2 Process

1.2.1 Life Cycle

BONITA has a very simple process life cycle

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

1.2.2 Ad'hoc processes

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

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.

1.2.3 Instances processes

Obviously, there are usage scenario where the reuse of process definition is of key importance ; in these scenario, one spends a long-time to define carefully a generic process model, that will be instantiated in the same way a lot of time. We call those processes administrative processes (we also say instance processes). They are based on model-instance workflow paradigm. In this kind of workflows, the Project API must be used to define the workflow model. When the process definition is done, the workflow users are enabling to instantiate the previous workflow model via Project API. Once model instance are created, workflow participants can access to the User API to obtain their todolist, to execute assigned activities, and so on.

So, either cooperative or administrative workflows use the same component definition API, that is Project API. The main difference between those workflow types concerns the execution. Cooperative workflows are ready to be executed and modified from the creation. In the other hand, administrative workflows need to be instantiated before. The term process model is used to talk about Bonita projects defined on the context of administrative workflow use case.

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.

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

1.2.4 Relationship to Users

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

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

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

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

1.2.5 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.

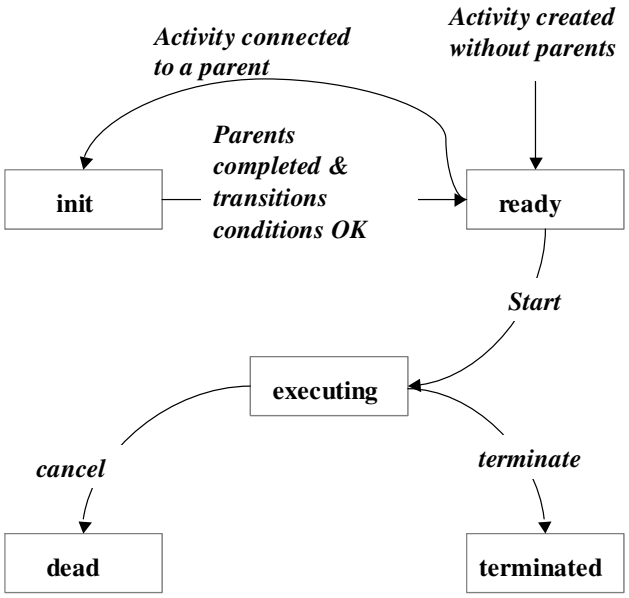
- Execution of an activity can be automatic; in this case the BONITA enactment engine will start it as soon as the applicable transitions from preceding activities are successfully evaluated.

- 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 (so is the first activity of the workflow process). In the second one, a normal activity has parent activities that have all terminated successfully, and whose transition condition to the activity has been successfully evaluated.
- **Initial :** This the state of an activity waiting for some processing to complete before being ready to run. In case of normal activities, at least one of the parent activities is still executing. In case of activity that can be anticipated, at least one of the parent activities has not started.
- **Anticipable:** This is the state of an activity that can be started, without waiting for its parent's activity to complete. All the parents' activities must be started however.
- **Anticipating:** A previously anticipable activity that has been started. Automatic activities are automatically transitioned from anticipable to anticipating, manual activities must be explicitly started. An anticipating activity cannot be terminated until all its parent activity has themselves terminated, and the transition conditions have been successfully evaluated.
- **Executing:** An activity being executed.
- **dead:** An activity that has been cancelled. All the depending activities will be automatically cancelled. Cancellation occurs in two cases: explicit cancellation, or unsuccessful evaluation of one of the inwards transition condition.
- **Terminated**: An activity that has been successfully terminated.

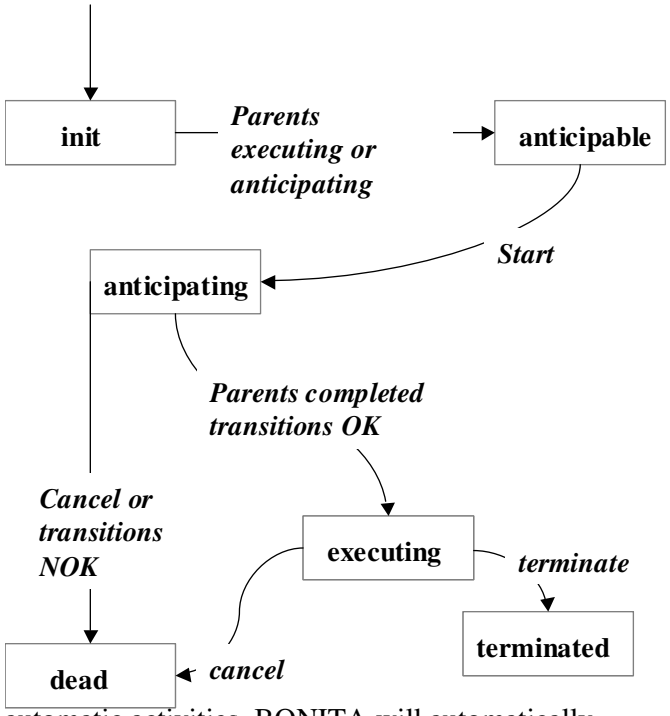
The activity life cycle is figured below for activities that cannot be anticipated.



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 the activity starts. The Before Start hook is not considered to be in the same transaction than the activity. The Before Start hook is not triggered for automatic activities that cannot be anticipated.
- After Start hook is called just after the activity has started. It is considered to be in the same transaction than the activity. The After Start hook is not triggered for automatic activities that cannot be anticipated.

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

Fault management

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

Consider the following simple scenario: an application calls the terminate Activity statement on "Activity1"; this triggers the execution of a before Terminate hook which raises an exception; the exception will be caught by the application.

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

Examples above show you two error scenarios related to transactional hooks execution. You should be aware that Hooks can be executed in a transactional or in a non-transactional context.

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

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

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

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

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

1.3.3 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.

1.3.4 Activity and transactions

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

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

BONITA will abort the transaction in two cases:

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

You should therefore be aware that some Hook types are executed in a transactional context.

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

1.4 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.
BONITA make the distinction between Users and Participants. The User base is the repository of all users that make use of the workflow system. Now for each process, one

should declare the Participants, that is all the Users that are allowed to play some role in the given process.

1.4.1 Relationship to processes

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

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

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

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

1.4.2 Authentication scenario

At the moment, User authentication is performed against BONITA specific database. It is also possible to authenticate against an LDAP directory.
You will find below an example of how to code the authentication of the admin user, using the “TestClient” login context proposes with Bonita.

Code example :

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


...

public class MyWorkFlowClass {

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

          ...
}
```

1.5 Roles

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.

1.6 Bonita LDAP Configuration (for JOnAS)

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

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

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

1.6.1 Installation

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

1.6.2 Configuration

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

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

1.6.3 What do this ldap Import?

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

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

1.6.4 How you can use it?

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

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

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

1.7 Mappers feature: automatic filling in of the bonita groups

1.7.1 Introduction

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

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

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

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

1.7.2 Mappers types

1.7.2.1 LDAP mapper:

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

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

- No checking that the distinguished names (dn) for the users found in the groups are compatible with the LDAP tree containing the users defined in the JOnAS LDAP realm configuration.

1.7.2.2 Custom mapper

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

- Take a look at sample class above and implements your custom mapper logic in a new java file.

- Add your .java file into this directory and then launch "ant" task from your $BONITA_HOME directory.

- After that, you can start JOnAS application server.

1.7.2.3 Properties mapper

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

1.8 Performer Assignment

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

1.8.1 Introduction

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

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

When this functionality is added, the user is notified (mail notification) that the activity is ready to be started.

The users of the groups (role in Bonita) associated to the activity can see the activity but cannot start and terminate it.

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

Furthermore, we can assign an activity to the **initiator of the instance.** It needs only the use of a *properties mapper* (as described above).

1.8.2 Description of these performer assignments

1.8.2.1 callback performer assignment

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

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

- Take a look at sample class above and implements your performer assignment logic in a new java file.

- Add your .java file into this directory and then launch "ant" task from your $BONITA_HOME directory.

- Start JOnAS application server.

1.8.2.2 Properties performer assignment

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

2 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.

You will find below examples of code using this interface. These are extracts of the Sample1 process proposed in the "samples" directory of Bonita. The sample1xxx classes implement the user guide workflow example (Order Processing and Customer Service).
You can run this example by using then ant tasks "sample1-create-process-model" (Model creation), "sample1-admin-wf" (user administration and project instantiation), "sample1-running-session" (Process execution).

2.2 Creating the ProjectSessionBean

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

Code example :

```
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();


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

        /* Project Session Bean Creation using local Interface
        ProjectSessionLocalHome projecth =
        (ProjectSessionLocalHome)hero.interfaces.ProjectSessionUtil.getLocalHome();
        ProjectSessionLocal projectsession = projecth.create();
        */

          ...
}
```

2.3 Initiating the ProjectSessionBean

2.3.1 Initiating the Session Bean

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

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

2.3.2 Initiating with the fresh instance creation option

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

Same as before, but when the corresponding *projectName* does not exist.
A new empty project is then created and given this name.

2.3.3 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 then deal with the corresponding *newProject*.

2.3.4 Initiating with the instantiate project creation option

```
Void  instantiateProject (java.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.

2.3.5 Code example

```
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();

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

          // Customer Service project creation
          prjSession.initProject("Customer Service");
                ... Customer Service Project Definition (See the following sections) ...

          // Customer Service project cloning into New Customer Service Project
          try {
          prjSession.initProject("Customer Service","New Customer Service");
          }catch(Exception e){System.out.println(e) ;} // Maybe project does not exists

         // Customer Service project Instantiation
          try {
          String instName = prjSession.instantiateProject("Customer Service ");
          }catch(Exception e){System.out.println(e) ;} // Maybe project does not exists
          ...
}
```

2.4 Managing project

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

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

2.4.1 Project attributes

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

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

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

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

2.4.2 Getting the name of a project or an instance

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

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

```
java.lang.String   getProjectNameOfInstance(java.lang.String instanceName)
```

Returns the project name of the instance instanceName.

2.4.3 Getting the name of the parent project

```
public java.lang.String getParent()
```

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

2.4.4 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.

2.4.5 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

```
java.util.Collection getPropertiesKey()
```

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

```
BnProjectPropertyValue getProperty(java.lang.String key)
```

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

```
void  deleteProperty(java.lang.String key);
```
Delete a property of an existing project.

2.4.6 Project details

```
public BnProjectValue getDetails()
```
Returns project information: project attributes, nodes, edges, hooks, properties...

2.4.7 Code example

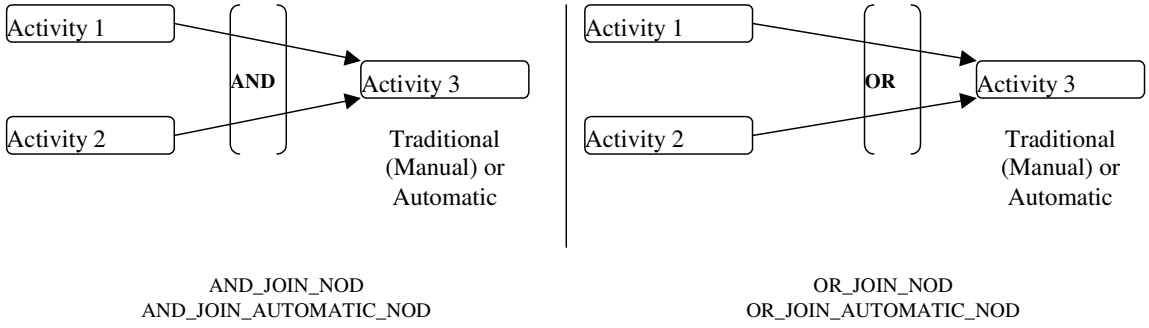
```
Scenario (To Be Implemented)
        InitProject(P1)
        GetName
        InstanciateProject(P1)
        getParent
        getCreator
        setProperty(…)
        setProperty(…)
        setProperty(…)
        collection getProperties : println
        collection getPropertiesKet : iterator getProperty : println
        deleteProperty
        getProperties
        getDetails
```

2.5 Managing activities

TBCompleted

2.5.1 Types of activities

Activity type can be either one of the following :



AND_JOIN_NOD
AND_JOIN_AUTOMATIC_NOD

OR_JOIN_NOD
OR_JOIN_AUTOMATIC_NOD

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

Here are the values associated with the types constants :

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

2.5.2 Activities states

See the "Activities basics " section of this document.

The values associated with the main activities states constants are :

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

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

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

2.5.3 Creating activity

```
void addNode(java.lang.String name, int nodeType)
```

Add a node to the project. This method creates a node with the corresponding node type and assign to it the hero.interfaces.Constants.InitialRole role.

2.5.4 Creating SubProcess activity

```
void addNodeSubProcess(java.lang.String name, java.lang.String projectName)
```

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

2.5.5 Configuring activity

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

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

`void` **`setNodeAnticipable`**`(java.lang.String name)`

Set the node in anticipable mode.

`void` **`setNodeAutomatic`**`(java.lang.String name)`

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

`void` **`setNodeDeadline`**`(java.lang.String name, long date)`

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

`void` **`setNodeDescription`**`(java.lang.String name,java.lang.String description)`

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

`void` **`setNodeProperty`** `(java.lang.String nodeName, java.lang.String key, java.lang.String value)`

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

`void` **`setNodeProperty`**`(java.lang.String nodeName, java.lang.String key, java.lang.String value, boolean propagate)`

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

`void` **`setNodeTraditional`**`(java.lang.String name)`

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

`void` **`setNodeType`**`(java.lang.String name, int type)`

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

2.5.6 iterating activities

`void` **`addIteration`**`(java.lang.String from, java.lang.String to, java.lang.String condition)`

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

2.5.7 Getting information about nodes in the project

`java.lang.Object` **`getNodes`**`()`

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

`java.util.Collection` **`getNodesNames`**`()`

Returns all node names of the project as a String Collection.

2.5.8 Code example

Scenario (To Be Implemented)
- AddNode N1, N2, N4
- AddNode N3 (subProcess)
- SetNode DeadLine, Desc
  - N2 : automatic
  - 3 properties
- Iterate N1-N2
- GetNodes
- GetNodesNames

2.5.9 Getting information about a specific node

`BnNodeValue` **`getNode`**`(java.lang.String projectName,java.lang.String nodeName)`

Get Node Value from a specific project

`java.lang.String` **`getNodeDeadline`**`(java.lang.String nodeName)`

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

`java.lang.String` **`getNodeDescription`**`(java.lang.String name)`

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

`java.lang.String` **`getNodeExecutor`**`(java.lang.String name)`

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

`java.util.Collection` **`getNodeProperties`**`(java.lang.String nodeName)`

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

`BnNodePropertyValue` **`getNodeProperty`**`(java.lang.String nodeName, java.lang.String key)`

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

`BnNodePropertyValue` **`getNodeProperty`**`(java.lang.String nodeName, java.lang.String key)`

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

`java.lang.String` **`getNodeRoleName`**`(java.lang.String nodeName)`

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

`int` **`getNodeState`**`(java.lang.String name)`
Returns the state of the node.

`int` **`getNodeType`**`(java.lang.String name)`
Returns the type of the node.

`BnNodeValue` **`getNodeValue`**`(java.lang.String name)`
Returns the node Value. Returns node information.

`boolean` **`getNodeAnticipable`**`(java.lang.String name)`
Returns if the node is set to be executed in anticipated mode.

2.5.10 Deleting activity

`void` **`deleteNode`**`(java.lang.String name)`
Delete a node from the project. If this node is in executing, terminated or cancelled state, the method throws an execption

`void` **`deleteNodeProperty`**`(java.lang.String nodeName, java.lang.String key)`
Delete a property of a node. Deletes the node property associated to this key

2.5.11 Code example

Scenario (To Be Implemented)
GetNodeValue, Type, …
Deleting
GetNodesNames

2.6 Managing users

2.6.1 Getting the list of all bonita registered users

`java.util.Collection` **`getAllUsers`**`()`
Return all registered users. Get user names of Bonita System

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

`java.util.Collection` **`getUsers`**`();`
Returns all users of the project. Users are returned as Strings.

2.6.3 Adding a user to a project

`void` **`addUser`**`(java.lang.String username);`
Add a user to this project (This user must exist at bonita database)

2.6.4 Checking whether a user is part of a project

`boolean` **`containsUser`**`(java.lang.String username);`
Test if the project contains this user

2.6.5 Code example

Scenario (To Be Implemented)
getUsers
addUser
containsUser

2.7 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.

2.7.1 Declaring a new role in the project

```
void  addRole java.lang.String roleName, java.lang.String description);
```
Add a role to the project. Creates a role within this project. The role is specific of this project

2.7.2 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);
```

2.7.3 Getting the list of roles that an user can assume

```
java.util.Collection  getUserRoles (java.lang.String userName);
```
Returns all the roles available for this user (independently of any project) as a collection of BnRoleLOcal objects

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

```
java.util.Collection getRoles()
```
Returns all roles of the current project as a collection of BnRoleLocal objects. These roles have been associated with the nodes included in the project.

```
java.util.Collection getRolesNames()
```
Returns the names of all the roles of the current project as a collection of String objects.

```
java.util.Collection getUserRolesInProject(java.lang.String userName)
```
Returns the roles of this user in the current project as a collection of BnRoleValue

```
java.util.Collection getUserRolesInProjectNames(java.lang.String userName)
```
Returns the role names of the user in the current project as a String Collection.

2.7.5 Associating an activity with a role

Only a single role can take over a given activity.

```
void setNodeRole(java.lang.String name, java.lang.String role)
```
Sets the role of an activity. Changes the role of this activity

2.8 Edge (transitions between activities)

2.8.1 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.

2.8.2 Deleting an edge

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

The edge named with the parameter name will be deleted.

2.8.3 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.

2.8.4 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.

2.8.5 Getting the condition on an edge

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

2.8.6 Getting all existing edges in a project

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

Returns all the existing edges in the project.

2.8.7 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 .

2.8.8 Reading an edge as a Java Object

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

Get the edge value.

2.8.9 Changing the state of an Edge

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

Set the edge state

2.9 Hooks

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

Must document in a central place the different possible scripting strategies

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

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

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

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

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

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

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

2.9.1 Managing a hook at the project level

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

Creating

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

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

```
Void  addInterHook  (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.

Deleting

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

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

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

Managing

```
void setInterHookValue(java.lang.String hook, java.lang.String value)
```

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

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

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

```
java.util.Collection getHooks()
```

Returns all the hooks of the project as a collection of ProjectHooksValue**.**

```
java.util.Collection getInterHooks()
```

Returns the interactive hooks of the project as a collection of ProjectInterHookValue .

```
String getInterHookValue(java.lang.String hook)
```

Returns the inter hook value script.

2.9.2 Managing a hook to an activity level

Creating

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

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

```
Void  addNodeInterHook  (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.

Deleting

`Void` **`deleteNodeHook`** `(java.lang.String hookName)`
Delete a node hook.

`Void` **`deleteNodeInterHook`** `(java.lang.String hookName)`
Delete a node interHook. The hook or the interHook with name hookName will be deleted from the node.

Managing

`java.util.Collection` **`getNodeHooks`**`(java.lang.String nodeName)`
Returns the Node hooks of the project as a NodeHookValue collection.

`java.util.Collection` **`getNodeInterHooks`**`(java.lang.String nodeName)`
Return all the Interactive Node hooks of the project as a NodeInterHookValue collection.

`BnNodeInterHookValue` **`getNodeInterHook`**`(java.lang.String nodeName, java.lang.String interHook)`
Returns all the node inter hook data associated to the hook of name « interHook » at the node « nodeName).

`String` **`getNodeInterHookValue`**`(java.lang.String node, java.lang.String hook)`
This method returns the hook script associated to the hook with name « hook » of this node

`Void` **`hookEvent`**`(java.lang.String nodeName, java.lang.String event)`
Execute the hook of node « nodeName » associated with the precised event.

2.10 Mappers

`void` **`addRoleMapper`**`(java.lang.String roleName, java.lang.String mapperName, int mapperType)`
Add an existing mapper to the role « roleName ». This type of mapper use a Java file loaded at run time.

`void` **`deleteRoleMapper`**`(java.lang.String roleName)`
Delete a role mapper.

`java.util.Collection` **`getRoleMappers`**`()`
Returns all the role mappers of the project as a collection of BnRoleMapperValue.

2.10.1 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,"mapper1",Constants.Mapper.LDAP);
         pss.addRoleMapper(role2,"mapper2",Constants.Mapper.PROPERTIES);

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

2.11 Performer assignment

(TBD : types and property list)

2.11.1 Addition of a performer assignment to a node

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

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

2.11.2 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");
```

3 User Registration Interface

3.1 Principle

TBCompleted

3.2 Creating the UserRegistrationBean

TBCompleted

Code example :

```
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();

          ...
}
```

3.3 Creating Users

`void` **`userCreate`**`(java.lang.String name, java.lang.String password, java.lang.String email)`

Creates user account. You have to call this method after "create" call (this API is an stateles session bean).

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

Creates user account with an instant messaging address. You have to call this method after "create" call (this API is an stateles session bean).

3.4 Creating Roles

`void` **`roleCreate`**`(java.lang.String name, java.lang.String roleGroup)`

Creates a new authorization role to the system. This kind of role is used to control the user access to different APIs. You have to call this method after "create" call (this API is an stateles session bean).

3.5 Defining Users

`void` **`setUserProperty`**`(java.lang.String userName, java.lang.String key, java.lang.String value)`

Set a new property to the user. User properties will be used to define user preferences. You have to call this method after "create" call (this API is an stateles session bean).

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

Set a new authorization role to the user. You have to call this method after "create" call (this API is an stateles session bean).

3.6 Deleting Users

`void` **`deleteUser`**`(java.lang.String userName)`

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

4 User Session interface

4.1 Principle

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

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

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

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

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

4.2 Creating the UserSessionBean

TBCompleted : User : authentified user.

Code example :

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

          ...
}
```

4.3 Setting User Information

```
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*.

```
void setUserMail(java.lang.String userName, java.lang.String mail)
```

Set the mail of this user into Bonita database.

4.4 Getting User Information

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

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

```
java.lang.String getUserPassword()
```

Returns the user password

```
java.lang.String getUserMail(java.lang.String userName)
```

Returns the mail of this user from Bonita database.

```
java.util.Collection getUserProperties()
```

Returns User properties as a BnUserPropertyValue Collection

4.5 Getting the list of projects for the User

```
java.util.Collection getProjectList()
```

Returns Workflow processes associated to this user as a Collection of BnProjectLightValue objects.

```
java.util.Collection getProjectListNames()
```

Returns project list names for this user as a String Collection.

```
java.util.Collection getProjectsByProperty(java.lang.String key,
java.lang.String value)
```

Returns Workflow projects from a property as a BnProjectValue Collection.

```
java.util.Collection getProjectsByPropertyNames(java.lang.String key,
java.lang.String value)
```

Returns Workflow projects from a property as a String Collection.

4.6 Getting the list of instances for the User

`java.util.Collection` **`getInstancesList`**`()`

Returns user instances list as a Collection of BnProjectLightValue objects. This method is equivalent to getProjectList but it only returns the current instances of the user.

`java.util.Collection` **`getInstancesListNames`**`()`

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

`java.util.Collection` **`getProjectInstances`**`(java.lang.String projectName)`

Returns Workflow instances of this project as a BnProjectValue Collection.

`java.util.Collection` **`getProjectInstancesNames`**`(java.lang.String projectName)`

Returns workflow instances names of this project as a String Collection.

`java.util.Collection` **`getInstancesByProperty`**`(java.lang.String key, java.lang.String value)`

Returns Workflow instances from a property as a BnProjectValue Collection.

`java.util.Collection` **`getInstancesByPropertyNames`**`(java.lang.String key, java.lang.String value)`

Returns a list of project instances from a property, as a String Collection.

4.7 Managing the project for the User

`void` **`removeProject`**`(java.lang.String projectName)`

Delete a Workflow project

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

`void` **`terminate`**`(java.lang.String projectName)`

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

4.8 Getting the list of activities for the User

`java.util.Collection` **`getActivityList`**`(java.lang.String projectName)`

Obtains all user activities from specific project (executing and anticipating state)

`java.util.Collection` **`getActivityListAllInstances`**`()`

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

`java.util.Collection` **`getActivityListByProperty`**`(java.lang.String key, java.lang.String value)`

Obtains executing user activities matching with property value (executing and anticipating state activities) as a BnNodeValue Collection.

4.9 Getting Information on User activity

`BnNodeValue` **`getNode`**`(java.lang.String projectName,java.lang.String nodeName)`

Returns Node Value from a specific project .

4.10 Getting the ToDo list for the User

`java.util.Collection` **`getToDoList`**`(java.lang.String projectName)`

Obtains all user activities from specific project (ready and anticipable state) as a String Collection.

`java.util.Collection` **`getToDoListAllInstances`**`()`

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

`java.util.Collection` **`getToDoListByProperty`**`(java.lang.String key, java.lang.String value)`

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

4.11 Managing activities for the User

`void` **`startActivity`**`(java.lang.String projectName, java.lang.String nodeName)`

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

`void` **`terminateActivity`**`(java.lang.String projectName, java.lang.String nodeName)`

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

`void` **`cancelActivity`**`(java.lang.String projectName, java.lang.String nodeName)`

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

4.12 Getting the list of existing properties for the User

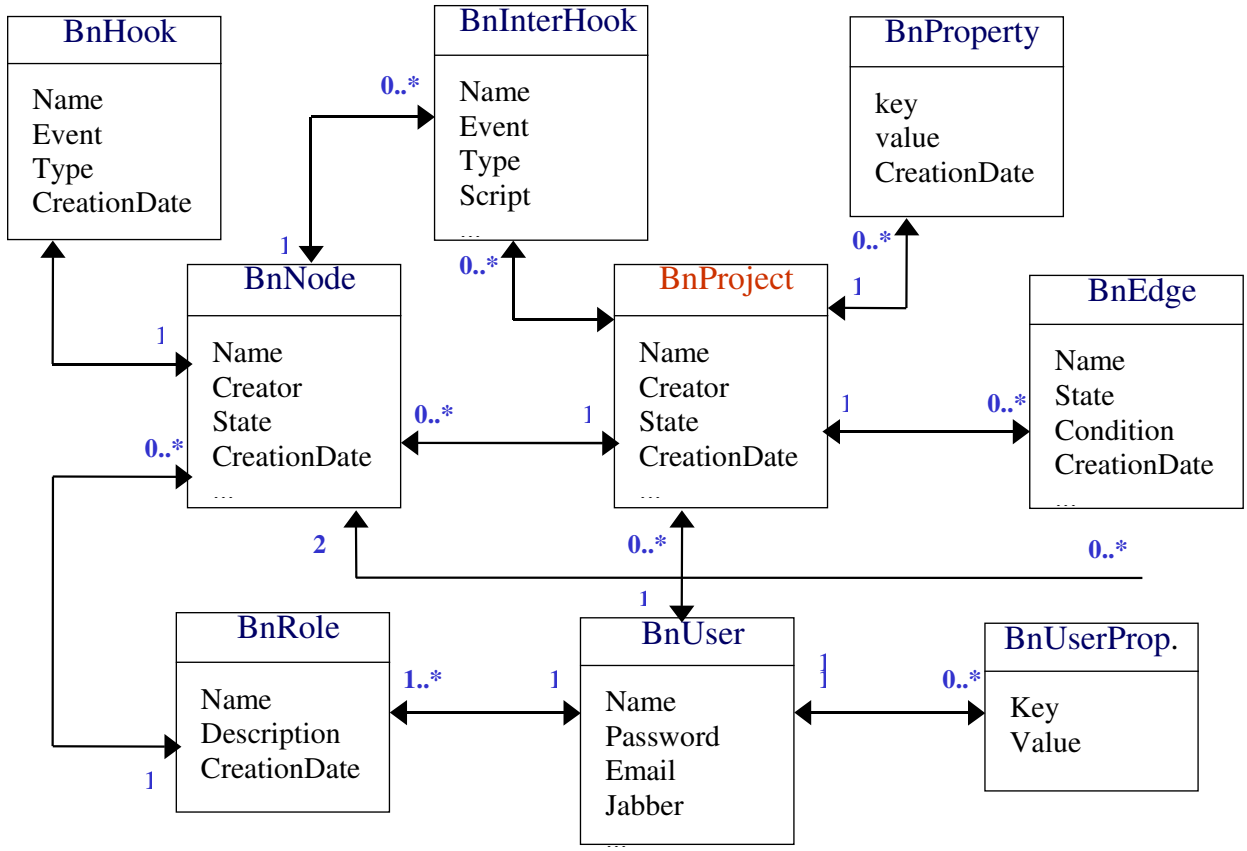
`Java.util.Collection` **`getUserProperties`** `()`

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

5 Bonita Entities

To have an idea of which kind of data can be accessed thru the bonita classes, you will find below the list of attributes of these classes. For further information, refer to the code in the bonita/build/generate/hero/interfaces directory.

5.1 Diagram



5.2 Entities Attributes

5.2.1 BnAuthRoleValue

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

RolePK		

5.2.2 BnEdgeValue

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

5.2.3 BnInstanceValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
java.lang.String	**name;**	
boolean	**nameHasBeenSet**	
java.lang.String	**creator;**	
boolean	**creatorHasBeenSet**	
java.lang.String	**parent;**	
boolean	**parentHasBeenSet**	
int	**state;**	
boolean	**stateHasBeenSet**	
java.util.Date	**creationDate;**	
boolean	**creationDateHasBeenSet**	
java.util.Date	**modificationDate;**	
boolean	**modificationDateHasBeenSet**	
hero.interfaces.BnProjectValue	**javaTree;**	
boolean	**javaTreeHasBeenSet**	
Collection	**BnUsers**	
Collection	**BnRoles**	

Collection	**BnNodes**	
Collection	**BnProperties**	
hero.interfaces.BnInstancePK	**pk;**	

5.2.4 BnIterationValue

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

5.2.5 BnNodeHookValue

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

5.2.6 BnNodeInterHookValue

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

5.2.7 BnNodePerformerAssignValue

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

5.2.8 BnNodePropertyValue

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

5.2.9 BnNodeValue

TYPE	ATTRIBUTE	MEANING
int	**id;**	
boolean	**idHasBeenSet**	
int	**type;**	
boolean	**typeHasBeenSet**	
int	**state;**	
boolean	**stateHasBeenSet**	
boolean	**anticipable;**	
boolean	**anticipableHasBeenSet**	
java.lang.String	**name;**	
boolean	**nameHasBeenSet**	
java.lang.String	**description;**	
boolean	**descriptionHasBeenSet**	
java.lang.String	**activityPerformer;**	
boolean	**activityPerformerHasBeenSet**	
hero.entity.NodeState	**transition;**	

boolean	**transitionHasBeenSet**	
hero.entity.EdgeState	**activation;**	
boolean	**activationHasBeenSet**	
java.util.Date	**startDate;**	
boolean	**startDateHasBeenSet**	
java.util.Date	**endDate;**	
boolean	**endDateHasBeenSet**	
java.util.Date	**deadline;**	
boolean	**deadlineHasBeenSet**	
java.util.Date	**creationDate;**	
boolean	**creationDateHasBeenSet**	
java.util.Date	**modificationDate;**	
boolean	**modificationDateHasBeenSet**	
hero.interfaces.BnUserLightValue	**Creator;**	
boolean	**CreatorHasBeenSet**	
hero.interfaces.BnUserLightValue	**Executor;**	
boolean	**ExecutorHasBeenSet**	
hero.interfaces.BnRoleValue	**BnRole;**	
boolean	**BnRoleHasBeenSet**	
hero.interfaces.BnNodePerformerAssignValue	**BnNodePerformerAssign;**	
boolean	**BnNodePerformerAssignHasBeenSet**	
hero.interfaces.BnProjectLightValue	**BnProject;**	
boolean	**BnProjectHasBeenSet**	
Collection	**BnProperties**	
Collection	**BnHooks**	
Collection	**BnInterHooks**	
hero.interfaces.BnNodePK	**pk;**	

5.2.10 BnProjectHookValue

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

5.2.11 BnProjectInterHookValue

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

5.2.12 BnProjectPropertyValue

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

5.2.13 BnProjectValue

TYPE	ATTRIBUTE	MEANING
Int	id;	
boolean	idHasBeenSet	
int	instanceNs;	
boolean	instanceNsHasBeenSet	
java.lang.String	parent;	
boolean	parentHasBeenSet	
java.lang.String	name;	
boolean	nameHasBeenSet	
java.lang.String	creator;	
boolean	creatorHasBeenSet	
int	state;	
boolean	stateHasBeenSet	
java.util.Date	creationDate;	
boolean	creationDateHasBeenSet	
java.util.Date	modificationDate;	

boolean	modificationDateHasBeenSet	
Collection	BnUsers	
Collection	BnRoles	
Collection	BnInstances	
Collection	BnNodes	
Collection	BnEdges	
Collection	BnAgents	
Collection	BnAgentEdges	
Collection	BnProperties	
Collection	BnIterations	
Collection	BnHooks	
Collection	BnInterHooks	
hero.interfaces.BnProjectPK	pk;	

5.2.14 BnRoleMapperValue

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

5.2.15 BnRoleValue

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

5.2.16 BnUserPropertyValue

TYPE	ATTRIBUTE	MEANING
int	id;	
boolean	idHasBeenSet	
java.lang.String	theKey;	

boolean	theKeyHasBeenSet	
java.lang.String	theValue;	
boolean	theValueHasBeenSet	
hero.interfaces.BnUserPropertyPK	pk;	

5.2.17 BnUserValue

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