Users and Groups

The methods and properties in this chapter allow you to manage the *Directory*, *Group*, and *User* objects. These objects are used to handle your application's security access system. The whole system is detailed in the **Data Security and Access Control** chapter.

Directory

This section describes the properties and methods available for a *Directory* object.

By default, the *Directory* object is built upon the directory file of the solution (named *solutionName*.waDirectory).

Note: In future versions of Wakanda, it will be possible to plug the Directory object into an external directory like a LDAP catalog.

internalStore

Description

The **internalStore** property contains the entire Wakanda users and groups directory as a datastore object. This internal datastore is built on the solution's directory file.

You can use this property to explore the solution's current directory (see example).

This property is only available for native Wakanda directories.

Example

In the following example, we can select all the users whose name starts with "P":

```
var pusers = directory.internalStore.User.query( "name = :1" , "P@" );
// User is one of the datastore classes in internalStore
```

addGroup( )

Group **addGroup**(String *name* [, String *fullName*])

Parameter	Type	Description
name	String	Name of the new group
fullName	String	Full name of the new group
Returns	Group	New group

Description

The **addGroup( )** method creates a new group in the solution's *Directory* and returns it as a *Group* object. The group will be created with the properties you passed in *name* and *fullName* and will automatically be assigned an **ID**.

- In *name*, pass the group's **name** property. This parameter is mandatory for the group to be created and must follow Wakanda's **Naming Conventions**.
- In *fullName*, pass the group's **fullName** property.

Note: If you try to create a group with a name that already exists in the datastore, an error is generated.

Once a group has been created, you can use the **putInto( )** method (for users) or **putInto( )** method (for groups) to include users or groups into the group.

Keep in mind that the group will be created in the solution's current directory, but will not be saved on disk until you call the **save( )** method on the directory.

Example

Create the "dev" group in the current directory:

```
var newGroup = directory.addGroup("dev" , "Developers");
directory.save();
```

addUser( )

User **addUser**(String *name* [, String *password*[, String *fullName*]])

Parameter	Type	Description
name	String	Name of the user
password	String	User's password
fullName	String	User's full name
Returns	User	Newly created user

Description

The **addUser( )** method creates a new user in the solution's *Directory* and returns it as a *User* object. The user will be created with the properties you passed in *name*, *password*, and *fullName*. Once created, the user will automatically be assigned an **ID**.

- In *name*, pass the user's **name** property. This parameter is mandatory for the user to be created and must follow Wakanda's **Naming Conventions**.

- In *password*, pass the user's password. It can be changed afterwards using the **setPassword( )** method. Note that password comparisons are case-sensitive.
- In *fullName*, pass the user's **fullName** property.

Note: If you try to create a user with a name that already exists in the datastore, an error is generated.

You may want to use the **putInto( )** method to add the new *User* to one or more groups in order to define the user's access rights.

Keep in mind that the user will be created in the open directory, but will not be saved on disk until you call the **save( )** method on the directory.

Example

We want to create a user named "Henry" in our solution:

```
var newUser = directory.addUser("Henry", "123", "Henry Charles");
directory.save(); // do not forget to save the changes
```

filterGroups( )

Array **filterGroups**(String *filterString* [, Boolean | String *isQuery*])

Parameter	Type	Description
filterString	String	String to filter group names (starts with), or query to apply to the internalStore directory
isQuery	Boolean, String	false or omitted = filterString is a 'group name starts with' filter (default), true or "query" = filterString is a query in internalStore
Returns	Array	Array of groups

Description

The **filterGroups( )** method returns the groups matching the *filterString* query in the directory.

filterString can contain either a predefined "group name starts with" query or a custom query to apply to the **internalStore** directory, which is the datastore that manages your current solution's users and groups:

- To define a "group name starts with" query, simply pass the letters to filter in the *filterString* parameter and omit the *isQuery* parameter (or pass **false**).
- To define a custom query in the **internalStore** datastore, write the query string in the *filterString* parameter and pass **true** or "query" in the *isQuery* parameter. You can query on any *Group* attribute. For more information about query strings, please refer to the **Defining Queries (Server-side)** section.

The corresponding groups are returned in an array of groups. If no group is found based on the *filterString* parameter, an empty array is returned.

Example

We want to get all the groups containing the string "dev":

```
var devG = directory.filterGroups("name = '@dev@'", true);
```

filterUsers( )

Array **filterUsers**(String *filterString* [, Boolean | String *isQuery*])

Parameter	Type	Description
filterString	String	String to filter user names (starts with), or query to apply to the internalStore directory
isQuery	Boolean, String	false or "not query" or omitted = filterString is a 'user name starts with' filter (default), true or "query" = filterString is a query in internalStore
Returns	Array	Array of users

Description

The **filterUsers( )** method returns the users matching the *filterString* query for the directory or specific group.

filterString can contain either a predefined "user name starts with" query or a custom query to apply to the **internalStore** directory, which is the datastore that manages your current solution's users and groups:

- To define a "user name starts with" query, simply pass the letters to filter in the *filterString* parameter and omit the *isQuery* parameter (or pass **false** or "not query").
- To define a custom query in the **internalStore** datastore, write the query string in the *filterString* parameter and pass **true** or "query" in the *isQuery* parameter. You can query on any *User* attribute. For more information about query strings, please refer to the **Defining Queries (Server-side)** section.

The corresponding users are returned in an array of users. If no user whose name meets the *filterString* is found, an empty array is returned.

Example

We want to get the users whose name starts with "john":

```
var arOlds = directory.filterUsers("john");
```

Example

We want to get the users whose password is not defined:

```
var arEmpty = directory.filterUsers("password is null || password = ''", "query");
```

Example

We want to get the users belonging to groups with the name starting with "admin":

```
var arAdmins = directory.filterUsers("groups.name = 'admin@'", true);
```

group( )

Group | Null **group**(String *name*)

Parameter	Type	Description
name	String	Name or ID of the group
Returns	Group, Null	Group object with the specified name or ID, or null if not found

Description

The **group( )** method returns a *Group* object containing the group corresponding to the name (or ID) you passed in the *name* parameter.

The name is used to identify the group and is given by the developer when it is created. The ID is an internal unique identifier automatically generated by Wakanda when the group is created.

The method returns Null if there is no *Group* with the given name or ID in the directory.

Example

To access a group of the current directory:

```
var myGroup = directory.group( "Accounting" ); // creates the group object
var members = myGroup.getUsers(); // gets the list of users (all levels) of the group
```

save( )

Boolean **save**([String | File *backup*])

Parameter	Type	Description
backup	String, File	Path or reference for a backup file of the Directory
Returns	Boolean	true if the directory was saved successfully, false otherwise

Description

The **save( )** method saves all changes made to the open solution directory. Changes are recorded in the solution's directory file, named *solutionName*.waDirectory.

You must call this method after you add, modify, or delete a user or a group programmatically.

If you pass the *backup* parameter, the solution directory is saved at the specified destination. This option is useful to save a copy of the directory for backup purposes or to use in a "try/cacth" structure in case of a write error in the current directory. You can pass either a full path to a file on your disk, or a reference to a *File* object in *backup*.

The method returns **true** if the directory was saved successfully, and **false** if an error occurred.

Example

This example changes the user's password and store it in the directory:

```
var user = directory.user("ed");
if (user != null)   // if the user exists in the directory
{
    user.setPassword("sjk16d");   // only the HA1 key will be stored in the directory
}
directory.save(); // save the directory
directory.save("c:/wakanda/backups/myDir.waDirectory"); //keep a copy
```

user( )

User | Null **user**(String *name*)

Parameter	**Type**	**Description**
name	String	Name or ID of the user
Returns	Null, User	User object with the specified name or ID, or null if not found

Description

The **user( )** method returns an *User* object containing the user corresponding to the name (or ID) you passed in the *name* parameter.

The name is a property of a user, used to log in the application, along with the password (see **name**).The ID is an internal unique identifier that you may you in some cases (see **ID**).

The method returns Null if there is no *User* with the given name or ID in the directory.

Example

This example accesses a user in the current directory:

```
var myUser = directory.user( "phil" ); // creates the user object
var toDisplay = myUser.ID + ", " + myUser.name; // returns two properties in the variable
```

Group

This section describes the properties and methods available for a *Group* object.

Group objects, which are defined in the solution's directory file (named *solutionName*.waDirectory), can contain users or other groups. *Group* objects are used to define permissions to your application's resources.

You can get a *Group* object using the **group( )** method or **addGroup( )** method of the **Directory** class.

name

Description

The **name** property contains the name of the *Group*. The **name** of a group is used to identify the group in your application, it must follow the general **Naming Conventions** of Wakanda.

ID

Description

The **ID** property contains the internal ID of the *Group*, which is a UUID that is automatically assigned by Wakanda when the group is created. The **ID** cannot be changed. If the group is deleted, its **ID** is never reused.

fullName

Description

The **fullName** property contains the full name of the *Group*. The **fullName** property value represents the actual name of the group (i.e., "Developer Group"), compared to the **name**, which is an ID (i.e., "dev1"). The **fullName** can be used to display the current group name on an interface page, for example.

filterUsers( )

Array **filterUsers**(String *filterString* [, Boolean | String *isQuery*])

Parameter	Type	Description
filterString	String	String to filter user names (starts with), or query to apply to the internalStore directory
isQuery	Boolean, String	false or "not query" or omitted = filterString is a 'user name starts with' filter (default), true or "query" = filterString is a query in internalStore
Returns	Array	Array of users

Description

The **filterUsers( )** method returns the users matching the *filterString* query for the directory or specific group.

filterString can contain either a predefined "user name starts with" query or a custom query to apply to the **internalStore** directory, which is the datastore that manages your current solution's users and groups:

- To define a "user name starts with" query, simply pass the letters to filter in the *filterString* parameter and omit the *isQuery* parameter (or pass **false** or "not query").
- To define a custom query in the **internalStore** datastore, write the query string in the *filterString* parameter and pass **true** or "query" in the *isQuery* parameter. You can query on any *User* attribute. For more information about query strings, please refer to the **Defining Queries (Server-side)** section.

The corresponding users are returned in an array of users. If no user whose name meets the *filterString* is found, an empty array is returned.

Example

We want to get the users whose password has not been defined:

```
var arEmpty = group.filterUsers("password is null || password = ''", true);
```

getChildren( )

Array **getChildren**([Boolean | String *level*])

Parameter	Type	Description
level	Boolean, String	true or "firstLevel" = get only first-level groups, false or "allLevels" or omitted = get groups including subgroups (default)
Returns	Array	Array of groups belonging to the group

Description

The **getChildren( )** method returns an array of the subgroups belonging to the *Group*.

Groups can be nested to create a hierarchy of inherited permissions in Wakanda.
By default, if you omit the *level* parameter or if you pass **false** or "allLevels" in this parameter, the method returns all the groups belonging to *Group,* including first-level groups (groups that are directly assigned to a group) and groups that belong to groups belonging to a group (i.e., subgroups) at all sublevels. For more information about group hierarchy in Wakanda, please refer to section **Users and Groups**.

If you pass **true** or "firstLevel" in the *level* parameter, only the groups directly assigned to a group are returned. Sublevel groups are ignored.

Example

We want to get both first-level and nested groups:

```
var g = directory.group("dev");
var x1 = g.getChildren(true); // get only groups assigned to a group
var x2 = g.getChildren("allLevels"); // get all groups including nested groups
// x1 <= x2
```

getParents( )

Array **getParents**([Boolean | String *level*])

Parameter	Type	Description
level	Boolean, String	true or "firstLevel" = get only first level groups, false or "allLevels" or omitted = get parent groups at all levels (default)
Returns	Array	Array of groups to which the group belongs

Description

The **getParents( )** method returns an array of the groups to which either *User* or *Group* belongs.

Groups can be nested to create a hierarchy of inherited permissions in Wakanda.
By default, if you omit the *level* parameter or if you pass **false** or "allLevels" in this parameter, the method returns all the parent groups of the *Group* or *User*. First-level groups (groups to which the *Group* or *User* is directly assigned) and parent groups of the parent groups at all levels are included in this case. For more information about group hierarchy in Wakanda, please refer to the **Users and Groups** section.

If you pass **true** or "firstLevel" in the *level* parameter, only the groups to which the *Group* or *User* is directly assigned are returned. Higher level groups are ignored.

getUsers( )

Array **getUsers**([Boolean | String *level*])

Parameter	Type	Description
level	Boolean, String	true or "firstLevel" = get only first-level users, false or "allLevels" = get users including subgroup users
Returns	Array	Array of users in the group

Description

The **getUsers( )** method returns an array of users belonging to the *Group*.

By default, if you omit the *level* parameter or if you pass **true** or "allLevels" in this parameter, the method returns all the users belonging to the group. First-level users (users who are directly assigned to a group) and users who belongs to groups that are assigned to the group (i.e., users in subgroups) at all sublevels are included in this case. For more information about group hierarchy in Wakanda, please refer to the **Users and Groups** section.

If you pass **true** or "firstLevel" in the *level* parameter, only those users who are directly assigned to a group are returned. Sublevel groups are ignored.

Example

We want to get both first-level and all level users:

```
var g = directory.group("finance");
var x1 = g.getUsers("firstLevel"); // get only users assigned to the group
var x2 = g.getUsers(); // get all users including those from nested groups, for example "account"
// x1 <= x2
```

putInto( )

void **putInto**(String | Array *groupList*)

Parameter	Type	Description
groupList	String, Array	List or array of groups

Description

The **putInto( )** method adds *Group* to the group(s) you passed in the *groupList* parameter. You assign a group to another group to define a hierarchy of access rights in the datastore. A group can be added to one or several other groups.

Several syntaxes are accepted for the *groupList* parameter:

- A string list of group names or IDs:

```
aGroup.putInto("sales", "finance", "admin"); // list of group names
aGroup.putInto("HDIKF56FD4XX...","SDFDFFD4XX..."); // list of group IDs;
```

- A list of *Group* objects:

```
var group1 = directory.group("finance");
var group2 = directory.addGroup("account");
 aGroup.putInto( group1 , group1 ); // list of group objects
```

  Note: You can mix group names, IDs, or references.

- An array of groups, containing either strings, group references, or both:

```
var arDev= directory.filterGroups("dev"); //array of group names
aGroup.putInto( arDev );
```

If you pass an invalid group name or reference in *groupList*, an error is generated.
If the *Group* is already included in a destination group, Wakanda just ignores the call (no error is generated).

remove( )

void **remove**( )

Description

The **remove( )** method removes the *User* or *Group* from the solution's *Directory*. The user or group reference is also removed from the groups to which it was assigned.

Keep in mind that the reference will be removed from the solution's open directory, but the change will not be saved on disk until you call the method on the directory.

removeFrom( )

void **removeFrom**(String | Array *groupList*)

Parameter	Type	Description
groupList	String, Array	List or array of groups

Description

The **removeFrom( )** method removes the *Group* from the group(s) you passed in the *groupList* parameter. Once removed from a group, the *Group* and its users lose all the assigned access rights in the datastore.

Several syntaxes are accepted for the *groupList* parameter:

- A string list of group names or IDs:

```
aGroup.removeFrom("sales", "finance", "admin"); // list of group names
aGroup.removeFrom("HDIKF56FD4XX...","SDFDFFD4XX...") // list of group IDs;
```

- A list of *Group* objects:

```
var group1 = directory.group("finance");
var group2 = directory.group("account");
 aGroup.removeFrom( group1 , group2 ) // list of group objects
```

  Note: You can mix group names, IDs, or references.

- An array of groups, containing either strings, group references, or both:

```
var arDev= directory.filterGroups("dev"); //array of groups names
aGroup.removeFrom( arDev );
```

If you pass an invalid group name, ID, or reference in *groupList*, an error is generated.
If the *Group* was not included in a listed group, Wakanda just ignores it (no error is generated).

Session

This section describes the properties and methods available for a *ConnectionSession* object.

ConnectionSession objects are returned by the **currentSession( )** method.

ConnectionSession objects handle the actual access privileges of a running user session on the server. These privileges can differ temporarily from the user privileges defined at the Directory level because of the "promote" mechanism. This mechanism allows a function to be executed with the privileges of a specific group. When such a function is executed from within a user session, the *ConnectionSession* privileges differ from the user privileges.

There is one session defined per thread, which means that the same user can run different sessions with different privileges.

For more information about the "promote" mechanism, refer to the **Assigning Group Permissions** section.

user

Description

The **user** property returns the *User* who runs the session on the server. The *null* value is returned when no user is logged for the session; in other words, when a "guest" session is running.

belongsTo( )

Boolean **belongsTo**(String | Group *group*)

Parameter	Type	Description
group	String, Group	Group to check for current session membership
Returns	Boolean	true if the current session belongs to the group, false otherwise

Description

The **belongsTo( )** method returns **true** if the current session belongs to the *group*.

If the current session does not have membership in the *group*, **belongsTo( )** returns **false** but does not generate an error (you have to send a permission error yourself). If you want to get a permission error, use **checkPermission( )** instead.

You can pass in *group* either:

- a group **name** (string)
- a group **ID** (string)
- a *Group* object

This method is useful to check membership on-the-fly for "promoted" functions.

Example

```
// we want to check that the session is run under the "Management" group

var session = ds.currentSession();
var isIn = session.belongsTo("Management");
if (isIn)
    {...;}
```

checkPermission( )

Boolean **checkPermission**(String | Group *group*)

Parameter	Type	Description
group	String, Group	Group to check for current session membership
Returns	Boolean	true if the current session belongs to the group, false otherwise

Description

The **checkPermission( )** method returns **true** if the current session belongs to the *group* and throws an error if **false**.

If the current session does not have membership in the *group*, **checkPermission( )** returns **false** and generates a permission error that you can handle in your code. If you do not want to get an exception in case of a permission error, use **belongsTo( )** instead.

You can pass in *group* either:

- a group **name** (string)
- a group **ID** (string)
- a *Group* object

This method is useful to check membership on-the-fly for "promoted" functions.

promoteWith( )

Number **promoteWith**(Group | String *group*)

Parameter	Type	Description
group	Group, String	Group into which to "promote" the current session
Returns	Number	Promoted session token

Description

WARNING: For security reasons, this method should not be public.

The **promoteWith( )** method temporarily promotes the current session into the *group*. All actions initiated in the session will be executed with the access rights associated with the group, in addition to those of the current user. The session will be "promoted" until the end of the hosting thread (that is, the end of the script execution) or until is executed.

Note: For more information about the "promote" action, please refer to the section.

You can pass in *group* either:

- a group (string)
- a group (string)
- a *Group* object

The **promoteWith( )** method returns a token number for the promoted session. This number can be passed to the method afterwards. The method returns 0 if no promotion was actually done (for example, when the user already belongs to the *group*, or when their access rights are higher than those of the *group*).

unPromote( )

void **unPromote**(Number *token*)

Parameter	Type	Description
token	Number	Session token

Description

WARNING: For security reasons, this method should not be public.

The **unPromote( )** method stops the temporary promotion set for the current session using the method. After this method is called, all actions initiated in the session will be executed with the standard access rights of the current user (if any).

In *token*, pass the session reference as returned by the method.

User

This section describes the properties and methods available for a *User* object.

User objects are based on "users" defined in your solution's directory file (named *mySolution*.waDirectory). Users are logged in the datastore using either the **loginByPassword( )** or the **loginByKey( )** method, available in the **Application** level of Wakanda.

You can get a *User* object using the **user( )** method or **addUser( )** method of the **Directory** class.

Note: You can also obtain a User object by calling the currentUser( ) method in the Application class.

name

Description

The **name** property contains the name of the *User*. The **name** property value is required for a user to log into the application along with the password.

fullName

Description

The **fullName** property contains the full name of the *User*. The **fullName** property value represents the actual name of the user (i.e., "John Smith"), compared to the **name**, which is the ID (i.e., "jsmith"). The **fullName** can be used to display the current user name on your interface pages, for example.

ID

Description

The **ID** property contains the internal ID of the *User*. The **ID** property value is a UUID that is automatically assigned by Wakanda when the user is created and cannot be changed. If the user is deleted, its **ID** is never reused.

getParents( )

Array **getParents**([Boolean | String *level*])

Parameter	Type	Description
level	Boolean, String	true or "firstLevel" = get only first-level groups, false or "allLevels" or omitted = get parent groups at all levels (default)
Returns	Array	Array of groups to which the user belongs

Description

The **getParents( )** method returns an array of the groups to which either *User* or *Group* belongs.

Groups can be nested to create a hierarchy of inherited permissions in Wakanda.
By default, if you omit the *level* parameter or if you pass **false** or "allLevels" in this parameter, the method returns all the parent groups of the *Group* or *User*. First-level groups (groups to which the *Group* or *User* is directly assigned) and parent groups of the parent groups at all levels are included in this case. For more information about group hierarchy in Wakanda, please refer to the **Users and Groups** section.

If you pass **true** or "firstLevel" in the *level* parameter, only the groups to which the *Group* or *User* is directly assigned are returned. Higher level groups are ignored.

putInto( )

void **putInto**(String | Array *groupList*)

Parameter	Type	Description
groupList	String, Array	List or array of groups

Description

The **putInto( )** method adds the *User* to the group(s) you passed in the *groupList* parameter. You add a user to a group to assign access rights in the datastore. A user can be added to one or several groups.

Several syntaxes are accepted for the *groupList* parameter:

- A string list of group names or IDs:

```
aUser.putInto("sales", "finance", "admin"); // list of group names
aUser.putInto("HDIKF56FD4XX...","SDFDFFD4XX...") // list of group IDs;
```

- A list of *Group* objects:

```
var group1 = directory.group("finance");
var group2 = directory.addGroup("account");
 aUser.putInto( group1 , group1 ) // list of group objects
```

Note: You can mix group names, IDs, or references.

- An array of groups, containing either strings, group references, or both:

```
var arDev= directory.filterGroups("dev"); //array of group names
aUser.putInto( arDev );
```

If you pass an invalid group name, ID, or reference in *groupList*, an error is generated.
If the *User* is already included in a destination group, Wakanda just ignores the call (no error is generated).

Example

In the following example, a group and a user are created. Then, the user is put in the new group as well as in two other existing groups:

```
var newUser = directory.addUser("john", "abc123" , "John DEACON");
var newGroup = directory.addGroup("Consulting"); // creates a new group
newUser.putInto("account" , "finance" , newGroup ); // add the user to 3 groups
```

remove( )

void **remove**( )

Description

The **remove( )** method removes the *User* or *Group* from the solution's *Directory*. The user or group reference is also removed from the groups to which it was assigned.

Keep in mind that the reference will be removed from the solution's open directory, but the change will not be saved on disk until you call the **save( )** method on the directory.

removeFrom( )

void **removeFrom**(String | Array *groupList*)

Parameter	Type	Description
groupList	String, Array	List or array of groups

Description

The **removeFrom( )** method removes the *User* from the group(s) you passed in the *groupList* parameter. Once removed from a group, a user loses all the access rights in the datastore defined by the group(s).

Several syntaxes are accepted for the *groupList* parameter:

- A string list of group names or IDs:

```
aUser.removeFrom("sales", "finance", "admin"); // list of group names
aUser.removeFrom("HDIKF56FD4XX...","SDFDFFD4XX...") // list of group IDs;
```

- A list of *Group* objects:

```
var group1 = directory.group("finance");
var group2 = directory.group("account");
 aUser.removeFrom( group1 , group2 ) // list of group objects
```

Note: You can mix group names, IDs, or references.

- An array of groups, containing either strings, group references or both:

```
var arDev= directory.filterGroups("dev"); //array of groups names
aUser.removeFrom( arDev );
```

If you pass an invalid group name, ID, or reference in *groupList*, an error is generated.
If the *User* was not included in one of the groups in *groupList*, Wakanda just ignores it (no error is generated).

setPassword( )

void **setPassword**(String *password*)

Parameter	Type	Description
password	String	New user password

Description

The **setPassword( )** method allows you to change the password for the *User*.
In *password*, pass the new password for the user. Remember that passwords are case-sensitive.