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

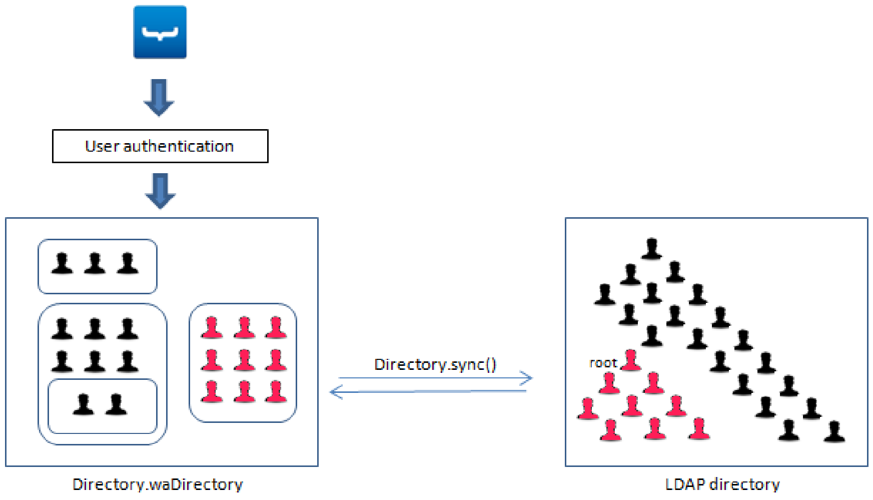
This section describes the properties and methods available for a *Directory* object.

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

You can also plug an external LDAP catalog to the *Directory* object using the **sync( )** method, that can be combined with a custom RemoteDirectory.json file (see **Using a LDAP remote directory**).

Using a LDAP remote directory

Wakanda allows you to create a local replication of a LDAP directory using the **sync( )** method (note that you can also pass a configuration object directly as parameter to the **sync( )** method). Once defined, this local replication can be used just like the Wakanda regular *Directory*. Remote users and groups can be authenticated in the same way as Wakanda directory users.



Note: A blog entry in the Wakanda Web site gives a detailed overview about how to synchronize Wakanda Directory's with remote LDAP directory.

Creating a RemoteDirectory.json file

To define a LDAP based remote directory in Wakanda, you need to create a **RemoteDirectory.json** file. This file must be stored in the **solution folder** of your application. When the solution is started, Wakanda will automatically synchronize the local directory with the remote LDAP directory.

During the session, you will be able to update the local replication of the directory using the **directory.sync( )** method.

The **RemoteDirectory.json** file should contain a single object, with the following attributes:

Attribute	Description
"url"	LDAP Server url (ldap:// or ldaps://)
"uuidAttribute"	binary attribute name
"userBind"	user for binding; Wakanda will use that user for binding with LDAP server
"password"	user password
"baseDN"	DN entry from where wakanda begin searching users and groups
"priority"	"remote" or "local" first authentication, in case of user name conflict

For example:

```
{
   "url": "ldap://192.168.222.23",
   "uuidAttribute": "objectGUID",
   "userBind": "CN=Administrator,CN=Users,DC=LDAPTest,DC=local",
   "password": "!123@",
   "baseDN": "OU=Test Unit Node,DC=LDAPTest,DC=local",
   "priority" : "remote"
}
```

Note: Alternatively, you can pass directly a LDAP definition object as parameter to the **directory.sync( )** *method. For more information, please refer to the description of the method.*

Configuring the remote directory

Once you have synchronized the remote LDAP directory with the local Directory, you can control access rights for remote users exactly as for local users.

However, some conflicts may arise between local and remote names. To avoid conflicts, you can use the following methods and attributes:

- **directory.getRemoteGroupByDN( )** to return a remote group through its DN
- then, **group.setAlias( )** to rename the group locally and **directory.getRemoteGroupByAlias( )** to work with local names (alias)
- in case of conflict between a local user name and a remote user name, the "priority" attribute of the remote configuration object allows you to define which user to authenticate first: "remote" means LDAP users, "local" means Directory users.
- you can configure the root entry from where Wakanda begins the search on users and groups into remote directory: the "baseDN" attribute of the remote configuration object allows you do set a DN from which the search will be started.

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 and execute any kind of query (see example).

This property is only available for native Wakanda directories.

Example

In the following example, we can select all the users whose password field is empty:

```
var pusers = directory.internalStore.User.query("password = null || password =''" );
// User is one of the datastore classes in internalStore
// password is one of the User class attributes
```

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
```

computeHA1( )

String **computeHA1**(String *userName* , String *password* [, String *realm*])

Parameter	Type	Description
userName	String	User name

password	String	User password
realm	String	Authentication realm
Returns	String	Digest HA1 hash value

Description

The **computeHA1( )** method returns the HA1 key resulting from the combination of *userName*, *password* and (optionally) *realm* parameters using a hash function.

HA1 keys can be used to store passwords with a high level of security. These keys result from applying a hash function that encrypts data so that it is impossible to retrieve the original information, i.e. the user password. Wakanda stores HA1 encrypted password keys in the solution's directory. This method allows you to set up a customized user management system, using a datastore class for example, and to benefit from the secure hash function to store and compare password values.

In *userName* and *password*, pass the string values to encrypt.

You can also pass a customized string value in *realm*. This value will be combined with the other parameters to generate the HA1 key. If you omit this parameter, the default "Wakanda" string is used as realm value. This value is used by Wakanda to generate password keys in the standard .waDirectory file.

***Note**: This method is case sensitive for every parameter.*

Example

This example shows how to use this method to set up a custom user management and authentication system. Users are stored as entities in a "User" datastore class. You defined a **HA1Key** storage attribute and a **password** calculated attribute.

- Here is the code for the **password** calculated attribute:

```
password :
        {
            onSet:function(value)
            {
                this.HA1Key = directory.computeHA1(this.name, value);
                  // we only store the HA1 key
            },
            onGet:function()
            {
                return "****"; //could also return Null
            }
        }
```

***Note**: You could check the password compliance with some security rules (not blank, number of characters, etc.)*

- To validate a login request, you create a function *validatePassword* for the "User" datastore class (applied to the entity):

```
entityMethods :
        {
            validatePassword:function(name, password)
            {
                var ha1 = directory.computeHA1(name, password);
                return (ha1 === this.HA1Key); // true if validated, false otherwise
            }
        }
```

With this authentication strategy, you need to pay attention to the following cases:

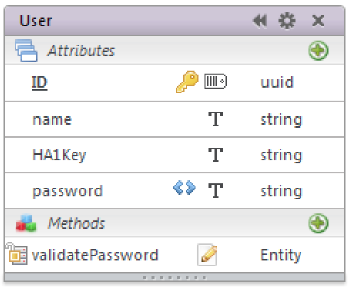
- if users want to change their name, they have to change their password as well;
- since you cannot control the code processing sequence, such a following instruction could fail because the engine may process the 'password' attribute before the 'name' attribute:

```
var u = new ds.User({name:"jones", password:"123"});
```

To avoid these cases, example 2 provides an alternative strategy that uses an uuid attribute instead of the name to compute the HA1 key.

Example

In this example, as in example 1 the custom security strategy is based on a "User" datastore class with a **HA1Key** storage attribute and a **password** calculated attribute. However, this time we will also use an ID attribute of the "uuid" type with the **Autogenerate** option:



User

Attributes

ID	uuid
name	string
HA1Key	string
password	string

Methods

validatePassword	Entity

- Here is the code for the **password** calculated attribute:

```
password :
        {
            onSet:function(value)
            {
                this.HA1Key = directory.computeHA1(this.ID, value);
                  // we use the ID to compute the HA1 key
            },
            onGet:function()
            {
                return "****"; //could also return Null
            }
        }
```

Note that we use the "autogenerated" **ID** attribute to calculate the HA1 key rather than the user name. This strategy has two advantages compared to the previous example:
  - as the ID attribute is automatically generated if not existing, it will be available at any time, even if the 'password' attribute is accessed before the 'name' attribute when the user is created.
  - the user will be allowed to change his login name without having to reset its password.

- To validate a login request, you define the *validatePassword* function for the "User" datastore class (applied to the entity):

```
            entityMethods :
            {
                validatePassword:function(password) //only use the password
                {
                    var ha1 = directory.computeHA1(this.ID, password);
                    return (ha1 === this.HA1Key); // true if validated, false otherwise
                }
            }
```

filterGroups( )

Array **filterGroups**(String *filterString*)

Parameter	Type	Description
filterString	String	String to filter group names (starts with), or "" to get all names
Returns	Array	Array of groups

Description

The **filterGroups( )** method returns all groups whose name starts with *filterString* in the *Directory*. The method returns groups defined at all levels in the *Directory*.

Pass the characters to be used with the "group name starts with" query in the *filterString* parameter. If you don't want to filter the group names, pass an empty string in *filterString*.

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 whose names contain the string "dev":

```
var devG = directory.filterGroups("*dev"); //groups contain "dev"
```

filterUsers( )

Array **filterUsers**(String *filterString*)

Parameter	Type	Description
filterString	String	String to filter user names (starts with), or "" to get all user names
Returns	Array	Array of users

Description

The **filterUsers( )** method returns all users whose names starts with *filterString* in the *Directory*. The method returns users at all levels in the *Directory*.

Pass the characters to be used with the "user name starts with" query in the *filterString* parameter. If you don't want to filter the user names, pass an empty string in *filterString*.

The 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");
```

getLoginListener( )

String **getLoginListener**( )

Returns	String	Name of the loginListener function

Description

The **getLoginListener( )** method returns the name of the *loginListener* function set by **setLoginListener( )** for the solution, if any. If no *loginListener* has been set, the function returns an empty string.

getRemoteGroupByAlias( )

Group | Null **getRemoteGroupByAlias**(String *alias*)

Parameter	Type	Description
alias	String	Local alias (name) of a group in remote LDAP directory
Returns	Group, Null	Group object with the specified alias, or null if not found

Description

The **getRemoteGroupByAlias( )** method returns the local *Group* object referencing the remote group with the alias (i.e. the local name) you passed in the *alias* parameter.

The alias is the local name of a remote group that you defined through the **getRemoteGroupByDN( )** method. The remote group must belong to the remote LDAP directory referenced using the **directory.sync( )** and/or in the solution's *RemoteDirectory.json* file (for more information, please refer to the **Using a LDAP remote directory** section). The method returns *Null* if the *alias* group is not found.

This method only select local copies of groups based on remote groups. Using this method, you can make sure to select the local copy of a remote group if a local group with the same name also exists in the Directory.

Example

You want to reference a remote group using its local name:

```
var grp1 = directory.getRemoteGroupByDN("CN=Group1,CN=Groups,OU=Unit Node,DC=LDAPTest,DC=local");
grp1.setAlias("IT_Systems");
var grp2 = directory.getRemoteGroupByAlias("IT_Systems");
//grp1 === grp2
```

getRemoteGroupByDN( )

Group | Null **getRemoteGroupByDN**(String *dn*)

Parameter	Type	Description
dn	String	Distinguished name of a group in remote LDAP directory
Returns	Group, Null	Group object with the specified DN, or null if not found

Description

The **getRemoteGroupByDN( )** method returns a local *Group* object referencing the remote group that corresponds to the unique Distinguished Name (DN) you passed in the *dn* parameter.

The DN group must belong to the remote LDAP directory referenced using the **directory.sync( )** and/or in the solution's *RemoteDirectory.json* file (for more information, please refer to the **Using a LDAP remote directory** section). The method returns Null if there is no group with the given DN in the remote LDAP directory.

Example

You want to create a new group in the directory based on a remote group defined by its DN:

```
var grp1 = directory.getRemoteGroupByDN("CN=Group1,CN=Groups,OU=Test Node,DC=LDAPTest,DC=local");
```

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
```

hasAdministrator( )

Boolean **hasAdministrator**( )

Returns	Boolean	True if the solution is running in controlled access mode, false otherwise

Description

The **hasAdministrator( )** method returns **true** if the solution is currently running under the *controlled admin access mode*, and **false** if it is under the free access mode.

The controlled access mode is activated as soon as there is at least one user with a password (or several users with or without passwords) added to the Admin group. In this mode, administration features, such as server start/stop, access to admin page, and debug, are restricted.

Controlled access and free access modes are discussed in the **Configuring Admin Access Control** section.

Example

You want to perform some actions depending on the current access status of the solution:

```
if(directory.hasAdministrator())
    ... ; // actions in controlled access mode only
```

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
```

setLoginListener( )

void **setLoginListener**(String *loginListener* [, Group | String *group*])

Parameter	**Type**	**Description**
loginListener	String	Name of function to execute when server receives a login request
group	Group, String	Group into which to "promote" the listener execution

Description

The **setLoginListener( )** method allows you to set a *loginListener* function to handle login requests for your Wakanda solution. It can add users dynamically to the Wakanda session and groups. Dynamic users only exist during the session; they are not stored in the solution's directory.

Like a standard request handler, a login listener function is automatically called by the server when a login request is received. The function receives user information (name, password or key, and optional infos) and must authenticate them by any custom means: you can use or call a datastore class, an external directory, another database, a Kerberos challenge (on Windows), etc. Depending on the function result, the request can be accepted, refused, or passed to the regular Wakanda login. For more information, refer to the **Authenticating Users** section.

Optionally, you can designate a group in the *group* parameter where you can pass either:

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

When this parameter is used, the *loginListener* function will be "promoted" to the specified *group* during its execution. This means that it will run with the group's access rights, allowing it for example to access data in protected areas of the application, such as log files or entities. Of course, once the *loginListener* function has finished executing, the user is logged with the privileges of their own group.
By default, if the *group* parameter is omitted, the *loginListener* function is executed with no specific access rights.

In *loginListener*, pass the name of the function to execute when a login request is received in custom mode.

The *loginListener* function must be written as follows:

```
function LoginListener(userName, passwordOrKey, secondIsAKey[, infos]);
```

Parameters

The *loginListener* function is called with the following parameters:

Name	**Type**	**Description**
userName	String	Name entered by the user
passwordOrKey	String	Password entered by the user if logged with **login( )** or Key associated to the user if logged with **loginByKey( )**
secondIsAKey	Boolean	**false** if the user logged with **login( )** or **true** if the user logged with **loginByKey( )**

infos	Object	Advanced login information, used in case of **Negociate** authentication mode (see). Provides the following attributes: *login* Windows login (ex: "jSmith") *realm* Windows domain name (ex: "wakUS") *method* Authentication mode ("Negociate" for the NTLM/Kerberos authentication mode)

On the client side, if the user was logged using the **loginByKey( )** method, the ***secondIsAKey*** returns **true**. In this case, the second parameter received is not a password but the hash key resulting from a computation of the password on the client side, sent by **loginByKey( )**. The key can result from any custom hash process, it does not necessarily need to be a SHA-1 key.
This feature allows you to use the same function on the client side and on the server side to compute the key:

- on the client, you let the user enter their name and password, then you compute a key from the password. You can then call the **loginByKey( )** function with the name and key.
- on the server, the *loginListener* function is called. You can check the *secondIsAKey* parameter and, if it is **true**, instead of comparing the password (that is not sent in that case), you compare the received key parameter with the local key that you have precomputed and stored in your user information.

Regarding Admin access, the *loginListener* function will be called only if the **Controlled Admin Access Mode** is activated -- that is, if the *.waDirectory* file of the solution contains, in the "Admin" group, at least one user with a password. This activation is necessary, even if you do not use the Directory file to identify your admin users. For more information, please refer to the **Configuring Admin Access Control** section.

Returned Value

- If the authentication is accepted, the *loginListener* function must return a "user" object containing the following attributes:

```
return{
    ID: //a UUID string referencing the user. It can be any UUID but must not be an existing user ID,
    name: //a string which will be the user name attribute,
    fullName: //a string which will be the user full name attribute,
    belongsTo: //an array of UUID strings, or
               //an array of group names referencing the groups the user must belong to
    storage: // a Storage object which is the sessionStorage property of the user session
};
```

  ***Note:** You can add any additional attributes you may need. For example, in the case of a NTLM/Kerberos authentication, you can add the login and the realm if necessary (see example).*

- If the authentication is refused, the *loginListener* function must return an "error" object containing the following attributes:

```
return{
    error: //the error number (passed as a number),
    errorMessage: //the text of the error
};
```

- If you want to pass the authentication to the Wakanda automatic login, the *loginListener* function must return *false*:

```
return false; //to continue the login using the standard process (use the internal directory)
```

Where to write the code?

- The **setLoginListener( )** method needs to be called only once. It may be a good idea to call it in the bootstrap of the main project (either a file in a "bootStraps" folder at the project root, or a file with the bootstrap role, see **getItemsWithRole( )**). Bootstrap code is automatically executed when the project is launched.

- The *loginlistener* function must be implemented in a **required.js** file that is located at the same level as the *.waSolution* file. Since this **required.js** file is evaluated for each JavaScript context created at the solution level, the *loginlistener* function will be available everywhere in the JavaScript code (see **required.js File**).

***Note**: You* cannot *use a **required.js** file that is located at the project level with the* **setLoginListener( )** *method.*

Example

We want to use a custom datastore class of the "MyDirectory" application to authenticate login requests, except for users existing in the directory of the solution.

- In the "bootstrap" of the "MyDirectory" project (or of any launched project of the solution), we define a loginListener:

```
directory.setLoginListener("myLogin", "Admin");
    // we want the listener to be run with the administrators group's privileges
```

- In a "required.js" file created next to the *.waSolution* file, we write the "myLogin" function:

```
function myLogin(userName, password)
{
        //get a reference to the current datastore of the MyDirectory application
        //this is necessary because the myLogin function can be called from any project of the solution
        //we 'share' the custom datastore class between all the projects of the solution
    var dsDir = solution.getApplicationByName("MyDirectory").ds;

    var p = dsDir.People({name:userName}); //look for the user in the People datastore class
    if (p == null) //if the user name does not exist in our datastore class
        return false; //let Wakanda try to find it in the solution's directory
    else // the user name is known
    {
        if (p.password == password) //this is given to keep the example simple
                    //we should have a more secured challenge here, for example
                    //by storing and comparing a hash key
        {
            var theGroups = [];
            switch (p.accessType){
                case 1:
                    theGroups = ['Internal'];
                    break;
                case 2:
                    theGroups = ['Administrator'];
                    break;
                case 3:
                    theGroups = ['Manager'];
                    break;
                case 4:
                    theGroups = ['Employee'];
                    break;
            }
            var connectTime = new Date();
            return {
                ID: p.userID,
                name: p.name,
                fullName:"guest "+p.name,
                belongsTo: theGroups,
                storage:{
                    time: connectTime,
                    access: "Guest access"
                        //in the user session, sessionStorage.access
                        //will contain "Guest access"
                }
            };
        }
        else
            return { error: 1024, errorMessage:"invalid login" }
    }
};
```

Example

We want to use a custom datastore class to authenticate login requests. We have defined different custom datastore classes depending on the projects.

- In the "bootstrap" of any project of the solution, we define a loginListener:

```
directory.setLoginListener("myLogin", "Admin");
    // we want the listener to be run with the administrators group's privileges
```

- In the solution's "required.js" file, we write the "myLogin" function:

```
function myLogin(userName, password) {
    var user = null;
    var result = false;
    //we want to use a specific user list per project
    //we look for the user in the datastore class depending to the project name
    if (application.name == "Employees") {
        user = application.ds.EmployeeUser( {name: userName} );
    }
    else if (application.name == "Accounting") {
        user = application.ds.AccountingUser( {name: userName} );
    }

    if (user != null) {
        if (user.password = password) {

            var theGroups = [];
            switch (user.accessType){
                case 1:
                    theGroups = ['Internal'];
                    break;
                case 2:
                    theGroups = ['Administrator'];
                    break;
                case 3:
                    theGroups = ['Manager'];
                    break;
                case 4:
                    theGroups = ['Employee'];
                    break;
            }
            var connectTime = new Date();
            return {
                ID: user.userID,
                name: user.name,
                fullName:user.name,
                belongsTo: theGroups,
                storage:{
                    time: connectTime,
                    access: "Guest access"
                        //in the user session, sessionStorage.access
                        //will contain "Guest access"
                }
            };
        }
        else
            result = {
                error: 1024,
                errorMessage:"invalid login"
            };
        }
    }
    return result;
};
```

Example

We want to use SSO and authenticate automatically our Wakanda users with their Windows credentials.

- The authentication type for the solution is set to "negociate".
- In the "bootstrap" of the "MyDirectory" project (or of any launched project of the solution), we define a loginListener:

```
directory.setLoginListener("myLogin");
```

- In a "required.js" file created next to the *.waSolution* file, we write the "myLogin" function:

```
function myLogin(userName, password, passIsKey, infos) {
    if(userName!="" && password=="") {  //in case of negociate method,
                //the login listener is called with a 'windows_login@ad_domain' userName
                //and without password value
        if("method" in infos) {
            if(infos.method=="Negotiate") { //we have been authenticated by Windows
                return {
                        //userName is 'login@realm'
                    ID: userName,
                    name: userName, //you can use any name
                    belongsTo: ["WinUsers"], //set the user to the appropriate Wakanda group

                    login: infos.login, //embedding additional attributes
                    realm: infos.realm,
                    method: infos.method
                };
            }
        }
    }
    return false; //otherwise, go to the internal directory
}
```

sync( )

void **sync**([Object *remoteLDAP*])

Parameter	Type	Description
remoteLDAP	Object	Remote directory configuration

Description

The **sync( )** method synchronizes the local Wakanda directory (*.waDirectory* file) with a remote LDAP directory.

This method creates a new local replication of the remote LDAP directory or updates the existing local replication. In case of an updated, any modification done to the remote LDAP directory is then applied to the local image.

Remote LDAP directory users can be authenticated just like local directory users. The local image will be kept locally during the whole solution session.

This method can be called from the active bootstrap of the solution, so that the local directory is synchronized at solution startup. You should also call it each time a modification is applied to the remote LDAP directory, to update the local replication.

To define the target remote directory, you can:

- either pass a configuration object in the *remoteLDAP* parameter of the **sync( )** method,
- or use a *RemoteDirectory.json* file located in the solution's folder (in this case, you pass **sync( )** without parameter). For more information, please refer to the **Using a LDAP remote directory** section.

If passed, the *remoteLDAP* object parameter should contain the following attributes:

Attribute	Description
"url"	LDAP Server url (ldap:// or ldaps://)
"uuidAttribute"	binary attribute name
"userBind"	user for binding; Wakanda will use that user for binding with LDAP server
"password"	user password
"baseDN"	DN entry from where wakanda begin searching users and groups

"priority"	"remote" or "local" first authentication, in case of user name conflict

For example:

```
{
   "url": "ldap://192.168.222.23",
   "uuidAttribute": "objectGUID",
   "userBind": "CN=Administrator,CN=Users,DC=LDAPTest,DC=local",
   "password": "!123@",
   "baseDN": "OU=Test Unit Node,DC=LDAPTest,DC=local",
   "priority" : "remote"
}
```

Example

In the bootstrap file, you want to synchronize your directory and return an error in case of failure (you have set a *RemoteDirectory.json* file in the solution's folder):

```
try{
    directory.sync();
}
catch(e) {
    console.log(e)
};
```

Example

You want to synchronize your directory with a LDAP directory passed as parameter:

```
var remoteLDAP = {
    "url": "ldap://192.168.222.22",
    "uuidAttribute": "objectGUID",
    "userBind": "CN=Administrator,CN=Users,DC=LDAPTest,DC=local",
    "password": "wakadmin#.1",
    "baseDN": "OU=Test Unit Node,DC=LDAPTest,DC=local",
    "priority" : "remote"
};
directory.sync(remoteLDAP);
```

user( )

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

Parameter	Type	Description
name	String	Name or ID of the user
Returns	User, Null	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 use 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.

filterChildren( )

Array **filterChildren**(String *filtrerString* [, Boolean | String *level*])

Parameter	Type	Description
filtrerString	String	String to filter group names (starts with) or "" to get all group names
level	Boolean, String	true or "firstLevel" = get only first-level groups, false or "allLevels" or omitted = get children groups at all levels (default)
Returns	Array	Array of groups belonging to the group

Description

The **filterChildren( )** method returns an array of the subgroups belonging to the *Group*, filtered using the *filterString* parameter. This parameter applies a 'starts with' query to the group names. To retrieve all the groups belonging the *Group*, pass an empty string in *filterString*.

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

To get first-level children groups of the "Managers" group whose names contain "top":

```
var myGroup = directory.group( "Managers" ); // creates the group object
var children = myGroup.filterChildren("@top","firstLevel"); // gets the first level filtered children groups
```

filterParents( )

Array **filterParents**(String *filterString* [, Boolean | String *level*])

Parameter	Type	Description
filterString	String	String to filter group names (starts with) or "" to get all group names
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 or group belongs

Description

The **filterParents( )** method returns an array of the groups to which the *User* or the *Group* belongs, filtered using the *filterString* parameter. This parameter applies a 'name starts with' query to the groups or users. To retrieve all the parent groups for the *User* or the *Group*, pass an empty string in *filterString*.

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 *User* or *Group*. First-level groups (groups to which the *User* or the *Group* 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 *User* or the *Group* is directly assigned are returned. Higher level groups are ignored.

If no group whose name meets the *filterString* is found, an empty array is returned.

filterUsers( )

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

Parameter	Type	Description
filterString	String	String to filter user names (starts with), or "" to get all user names
level	Boolean, String	true or "firstLevel" = get only first-level users, false or "allLevels" or omitted = get users at all levels (default)
Returns	Array	Array of users

Description

The **filterUsers( )** method returns an array of the users that belong directly or indirectly to the *Group*, filtered using the ***filterString*** parameter. This parameter applies a 'name starts with' query to the users. To retrieve all the users for the *User*, pass an empty string in ***filterString***.

By default, if you omit the ***level*** parameter or if you pass **false** or "allLevels" in this parameter, the method returns all the users directly assigned to the *Group* as well as any users belonging to one of its subgroups at all levels 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 users directly assigned to the *Group* are returned. Lower level users are ignored.

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 all the users whose name starts with "customer_":

```
var arEmpty = group.filterUsers("customer_");
```

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 , 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 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 **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 *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).

setAlias( )

void **setAlias**(String *alias*)

Parameter	Type	Description
alias	String	New local alias (name) of a group in remote LDAP directory

Description

The **setAlias( )** method sets a local name (alias) to the *Group* object corresponding to a remote group from a LDAP directory.

The alias can be set only for remote groups, this method does not have effect on local directory groups. Remote groups must belong to the remote LDAP directory referenced using the **directory.sync( )** and/or in the solution's *RemoteDirectory.json* file (for more information, please refer to the **Using a LDAP remote directory** section).

The alias can be used by the **getRemoteGroupByAlias( )** method. Setting a local alias for remote groups is necessary to differentiate between groups with same name and different DN or local and remote groups with the same name.

Example

You want to reference a remote group and put it in an existing local group named "Account":

```
var rgp = directory.getRemoteGroupByDN("CN=Account,CN=Groups,OU=Account Node,DC=LDAPTest,DC=local");
rgp.setAlias("remote_Account");
remote_Account.putInto("Account")
```

Permissions

The **Permissions** class object can be accessed through the application.**permissions** property.

This object contains a single method that returns available information on defined permissions for the project.

findResourcePermission( )

Object | Undefined **findResourcePermission**(String *type*, String *resource*, String *action*)

Parameter	Type	Description
type	String	Type of the resource
resource	String	Name of the resource
action	String	Action associated with the resource
Returns	Object, Undefined	JSON object describing the resource

Description

The **findResourcePermission( )** method returns a JSON object describing the permission defined for the specified ***type***, ***resource*** and ***action***. For example, this method allows you to check that a **Group** is assigned to a permission.

In the parameters, pass strings defining the permission to find:

- ***type***: type of the resource. It can be "model", "dataClass", "method", "attribute", "module", or "service".
- ***resource***: name of the resource. It depends on the resource type but can be, for example, "Model", "Model.*{DatastoreClassName}*.*{MethodName}*", "myModule", "myService"...
- ***action***: action(s) allowed. It depends on the resource type and can be "read", "create", "update", "remove", "describe", "execute", "promote", "executeFromClient", or "upload".

Note*: For more information on permission definition, please refer to the project's* **Permissions** *section.*

If the target resource exists, the method returns a JSON object containing each attribute of the permission definition. In particular, it returns the group **ID** and any custom property defined for the permission in the .waPerm file.

The method returns *undefined* if the target resource was not found.

Example

Assuming the following permission is defined in the .waPerm file:

```
<allow type="module" resource="myModule" action="executeFromClient" groupID="0CD5A2B4253CE940AXXXXXXXXXXXXXXX"
customProperty="toBeChecked"/>
```

If you execute the following code:

```
var p = application.permissions.findResourcePermission( 'module', 'myModule', 'executeFromClient');
```

the value of *p* will be:

```
{
    type: "module",
    resource: "myModule",
    action: "executeFromClient",
    groupID: "0CD5A2B4253CE940AXXXXXXXXXXXXXXX",
    customProperty: "toBeChecked"
}
```

Session

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

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

Default Session

By default, if a user runs a session without being logged, a default user session is used. In this case, the **currentSession( )** method returns a valid session with a user whose name is "default guest" and ID is 00000000000000000000000000000000. This default session supports standard session features.

Starting with Wakanda 10, all default guest connections share the same session on the server. This feature allows saving server memory.

To "force" a guest user connection to open a separate session on the server, usually for testing needs, you just need to execute a **WAF.directory.logout( )** method while connecting the guest user.

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.

If the session is run by a non-logged user, a default user session is used. The **user** property returns a *User* whose **name** is "default guest" and **ID** is 00000000000000000000000000000000.

Example

To get the user running the current session on the server:

```
var curSession = currentSession();
var curUser = curSession.user;
```

storage

Description

The **storage** property returns the *Storage* object associated with the current user session. This property gives you a direct access to the **sessionStorage** application property for the session.

Example

```
var mySession = currentSession(); //gets the current session
var myStorage = mySession.storage;
    // myStorage is equivalent to sessionStorage for the session
```

ID

Description

The **ID** property contains the internal ID of the session on the server. The **ID** property value is a UUID that is automatically assigned by Wakanda to the session when it is opened on the server.

The session **ID** is stored in the cookie sent to the user (WASID).

expiration

Description

The **expiration** property returns the current expiration date and time of the user session.

The expiration date and time of the session is the moment when the user session will be closed automatically on the server if no user query or action is performed within the thread. On the other hand, the **expiration** date is postponed for a new **lifeTime** duration as soon as a user action is performed in the thread.

lifeTime

Description

The **lifeTime** property returns the default lifetime of a user session in seconds. By default, the lifetime is 3600, i.e. one hour.

It can be set when the user session is opened using the **loginByPassword( )** or **loginByKey( )** method on the server.

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 = 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.

forceExpire( )

void **forceExpire**( )

Description

The **forceExpire( )** method makes the user session expire.

Once the method is called, the session expires on the server. This may take some extra time to be accomplished because the server waits until any tasks running in the thread are finished.

Since the same user can run several different sessions, this method can be used, for example, to limit the number of sessions opened by a single user.

Example

You want to force the expiration of all user sessions other than the current one (except if it's a default guest session):

```
var curSession = currentSession();
var user = curSession.user;
if (user.ID != "00000000000000000000000000000000") //default guest sessions
{
    var sessions = getUserSessions(user);
    sessions.forEach(function(item) {
        if (item.ID != curSession.ID) {
            item.forceExpire();
        }
    });
}
```

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

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 **unPromote( )** is executed.

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

You can pass in *group* either:

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

The **promoteWith( )** method returns a token number for the promoted session. This number can be passed to the **unPromote( )** 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*).

Example

```
var token = currentSession().promoteWith('GroupName');
 ... //Code needing elevated access
currentSession().unPromote(token); //back to the initial access rights
```

unPromote( )

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

Parameter	Type	Description
token	Number	Session token

Description

The **unPromote( )** method stops the temporary promotion set for the current session using the **promoteWith( )** 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 **promoteWith( )** 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 by:

- using the **user( )** method or **addUser( )** method of the **Directory** class.
- 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.

By default, Wakanda assigns "default guest" as **name** property to non-logged users.

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 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.

By default, Wakanda assigns "00000000000000000000000000000000" as **ID** property to non-logged users, i.e. "default guest" users.

storage

Description

The **storage** property returns the ***Storage*** object associated with the *User*. This object is automatically available for each user defined in the solution and is maintained as long as the Wakanda server is alive (it is not stored after the server shuts down). You can use it for example to write initialization data at startup or to count the number of times a user logs in or out.

Note that this property is user-related and not session-related. The **user.storage** object is available even when the user is not logged.

Example

We want to store the number of times each user connects to the application. In a custom login method, we increment the user log account (see full example in the **loginByPassword( )** method description):

```
model.Person.methods.login = function(userName, password)  //  the function gets name and password
    {
        var result =loginByPassword(userName, password, 60*60); // session is  created in case of success
        if (result) // user identified successfully
            {
                var user = currentUser(); //gets the user running the session
                var logs = user.storage.logs; //gets the user log count
                if (logs == null) // logs key does not exist, it is the first login
                    logs = 0; // initialization

                var newLog = logs + 1; // incrementation  otherwise
                user.storage.logs = newLog; //stores the new log
            };
        return result; // result is sent to the client
    }
```

filterParents( )

Array **filterParents**(String *filterString* [, Boolean | String *level*])

Parameter	Type	Description
filterString	String	String to filter group names (starts with) or "" to get all group names
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 or group belongs

Description

The **filterParents( )** method returns an array of the groups to which the *User* or the *Group* belongs, filtered using the *filterString* parameter. This parameter applies a 'name starts with' query to the groups or users. To retrieve all the parent groups for the *User* or the *Group*, pass an empty string in *filterString*.

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 *User* or *Group*. First-level groups (groups to which the *User* or the *Group* 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 *User* or the *Group* is directly assigned are returned. Higher level groups are ignored.

If no group whose name meets the ***filterString*** is found, an empty array is returned.

Example

We want to get the parent groups of the "john" user containing the "admin" string at all levels:

```
var theUser = directory.user( "john" ); // get the user
var arrGrps = theUser.filterParents("@admin", "allLevels"); // filter group names containing "admin"
```

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).

Keep in mind that the User 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.

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.

Keep in mind that the password will be set in the solution's directory, but the change will not be saved to disk until you call the **save( )** method for the directory.