# Data Security and Access Control

Access control and data protection are important issues for Web applications.

Wakanda provides you with a complete security system that allows you to protect your Web applications. By combining the following three principles that define Wakanda's security system, you can set up security levels that are tailored to your needs and the desired restrictions for your Web applications:

- Set up **users** who can connect to your application and define which **group**(s) they belong to.
- Define **access groups with permissions** for the various application resources (datastore models, classes, class methods, administration features, RPC calls, etc.).
- **Authenticate** users for your web applications through a standard or custom interface.

# Users and Groups

The access control system in Wakanda applications is based on the concept of **users** and **groups**.

Each user can belong to one or more groups. You can then assign access rights to each group for the different aspects of your Wakanda application. When a user logs in and is authenticated by the system, he or she automatically has access to all the application resources associated with the group(s) to which he or she belongs.
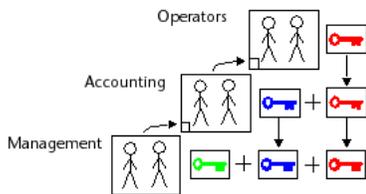
For example, if you place the user *John Smith* in the "Accounting" group and then allow this group to create entities in the "Invoice" class, when John Smith logs on, he can create invoices in the application (either from a Page or any client request). A user who does not belong to the "Accounting" group cannot create invoices. If this user sends an HTTP request to create an invoice, a 401 error is returned by the server.

You can also adapt your Web application's interface by showing or hiding various widgets according to the access rights of the user.

You can place groups inside other groups to set up a hierarchical access system. The principle is that included groups inherit access rights from its encompassing parent group. In such a system, the group with the lowest access rights contains other group(s) with higher access rights. For example, you can create a hierarchy with three groups so that access rights are assigned correctly to the users in each group:

- "Operators": Provides basic rights, for example entering new data only. Users belonging only to this group can enter new data but cannot edit or delete existing entities. In addition to individual users, you can include in this group all higher level groups containing users that you also want to be able to enter new data, in addition to their own access rights. In our example, we include the "Accounting" group.
- "Accounting": Gives data maintenance rights, for example accessing and modifying data. Users belonging only to this group can access and modify existing data, as well as enter new data, which is inherited from the "Operators" group - but they cannot delete entities. In addition to individual users, you can include in this group any higher level groups containing users that you also want to be able to enter and edit data, in addition to their own access rights. In our example, we include the "Management" group.
- "Management": Gives data deletion rights. Users belonging only to this group can delete entities as well as enter new data, and access and modify existing data, which are inherited respectively from the "Operators" and "Accounting" groups. This group contains only the highest-level users, no other group is included inside.

You can easily control the access rights assigned to each group according to the user's level of responsibility. The following diagram illustrates how access rights are inherited for the different groups in our example:



In the Wakanda Studio Directory editor, the following settings are defined:



## Setting Up Users and Groups

Users and groups are defined for a Wakanda solution and are saved in a file whose extension is **.waDirectory**. This file is located at the same level as your Wakanda solution. The directory is shared among all the projects in your solution, and is created for each new solution that you create by default. The new directory does not contain any users; however, there are two groups created by default: Admin and Debugger.

There are four ways that you can set up users and groups in this file:

- Using Wakanda's Directory editor (see in the **Solution Manager**).
- Writing XML in Wakanda's Directory format (see below).
- Using the various methods and properties of the **Users and Groups** API. Changes can then be saved using the **directory.save( )** method.
- Using a directory of external users defined in LDAP or ActiveDirectory, for example. In this way, you can interface a Web application with an existing business directory. To do this, you create local groups (in the Directory file) that refer to the external groups.
  *Note: In the current version of Wakanda, this feature has not yet been implemented.*

## Managing the Wakanda Directory File

Wakanda's Directory file contains a complete diagram of your solution's users and groups in XML format. In Wakanda Studio, you can set up and modify your solution's directory using the following editor:
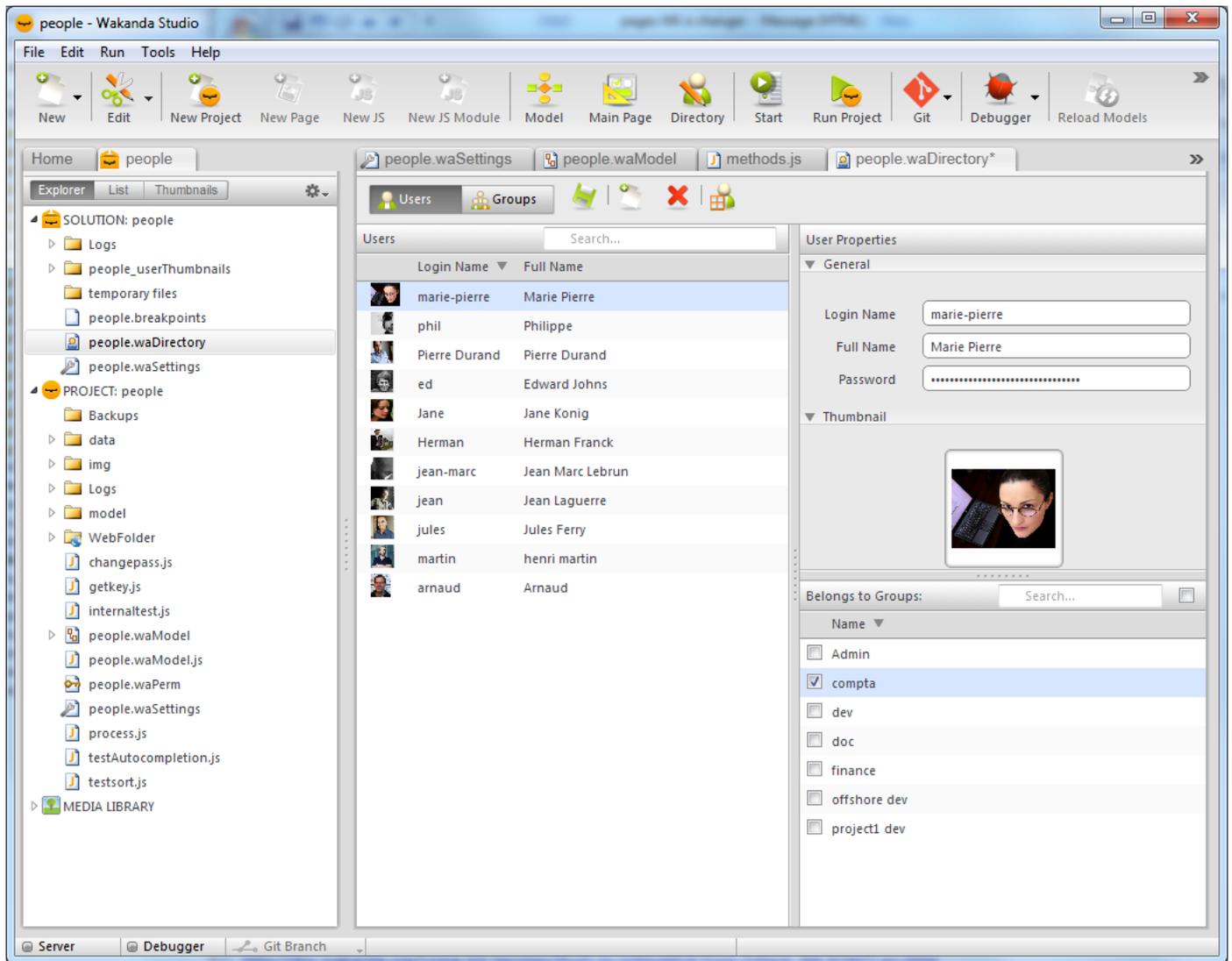
For more information about this editor, refer to the section in the **Wakanda Studio Reference Guide**.

Remind that can also create, modify, or remove users and groups at runtime using the methods and properties from the **Directory**, **Group** and **User** classes.

Here are a raw extract from our example Directory file:

```
<user name="john" ID="758F13625DFFEA45A3597E788E5A6F4A" password="DJKNDNDNSK..." fullName="John Smith">
    <belongsTo group="accounting"/>
</user>
```

The properties for the *user* element are the following:

- *name*: the user identifier to be used when logging in.
- *id*: the user's unique ID (UUID). Wakanda automatically manages this ID and ensures that it will not be reused. This UUID can also be used to identify a user.
- *password*: a hash key (HA1 specifications) used to validate the user's password. The user's password is never saved directly in this file. To ensure maximum security, the password's validation is a result from a challenge using this hash key as well as other information.
- *fullName*: the user's name that can be displayed in the application's interface by the Login Dialog widget or your own custom interface.
- *belongsTo*: lists the group(s) to which a user belongs. You can also use the *include* keyword at the group level (see below).

```
<group name="finance" ID="3C9E1D3CC0C7BD43AECED581002B0AC6" fullName="Finance People">
    <include user="ed"/>
    <include user="John Smith"/>
    <include group="international sales"/>
    <belongsTo group="marketing"/>
</group>
```

The properties for the *group* element are the following:

- *name*: group identifier.
- *id*: group's unique ID (UUID). Wakanda automatically manages this ID and ensures that it will not be reused. This UUID can also be used to identify a group.
- *fullName*: name of the group that can be displayed in the application's interface by the Login Dialog widget or your own custom interface.
- *include*:  each entry defines the name of a user or of another group belonging to the group.
- *belongsTo*: each entry defines the name of a group to which this group belongs (alternative to *include*).

**Default Directory File**

By default, when you create a new solution, Wakanda creates a **.waDirectory** file that contains a single empty group: "Admin". This group is designed to contain high-level users who have access to your Web application's administration and debugging features:

- Activate the Admin Access Control when it contains at least one user with a password or two users without a password

- Connect to a solution with Wakanda Studio or the Server Administration page and full access to development features
- Access to the Debugger
- Ability to execute a server-side JavaScript file

*Note: You can rename the "Admin" group in your directory.*

While the "Admin" group contains no user with a password, any user who connects to a solution using either Wakanda Studio or the Server Administration page will have full access. Internally, the user connects with "default guest" access rights. If, however, you have more than one user without passwords in the "Admin" group, the Admin Access Control is activated.

If you want to control the admin access to your solution, which is strongly recommended, you must **activate the Admin Access Control**. For more information, refer to the **Configuring Admin Access Control** section.

### Handling Users and Groups at Runtime

On the server, you can obtain information about the currently logged in user to display his/her full name and to check if he/she belongs to a specific group.

- In the global application object, the **currentUser( )** method returns a *User* type object indicating the current session's logged in user's name. To display the name of the currently logged in user on the client, you can create a function that queries the server and returns the user name.

- In the global application object on the server, there are three classes you can use to modify or obtain information:
    - **Directory**
    - **Group**
    - **User**
    Note that the methods and properties available for the *Directory* object can be limited in the case of an external LDAP directory.

- In custom queries (see **Using placeholders in query strings**), you can use the **$userID** or **$userName** placeholders to indicate the current user ID or name. This feature can be used to link user data to users. You can add an attribute in the datastore class that stores the current user ID. This attribute will be automatically filled in when an entity is added, for example in the datastore class's **onInit** event:

```
onInit:function()
{
    var theOwner = currentUser(); // get the user who creates the entity
    this.owner = theOwner.ID; // store the ID of the user who creates the entity in the owner attribute
}
```

You will then be able to access data relevant to this user using a query such as:

```
ds.Lead.query("owner = :$userID") // return leads created by the user
```
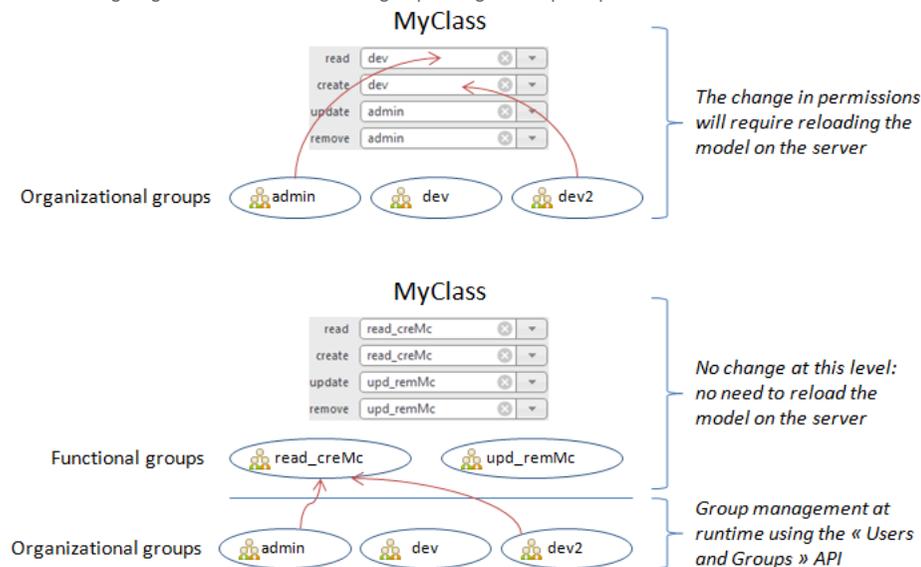
### Functional Groups vs Organizational Groups

When you define user groups in Wakanda, you will usually start by creating *organizational* groups; in other words, groups that reflect the organization of users: "Accounting", "Production", "Management" and so on. You can then easily assign appropriate permissions to groups depending on their status in the organization. However, when permissions are set at the model level (e.g. for a class), each time you assign or change a permission, Wakanda will ask you to reload the model on the server. In a production context, this could penalize the application fluidity if you need to change permissions on a regular basis.

To avoid this issue, a good solution consists of creating a *functional* group for each permission you want to assign and then to include organizational groups within these functional groups. For example, for "read" permission on the "Invoice" dataclass, you can create and assign an "Invoice_read" group. Then, you can put any relevant organizational group (Accounting, Management, etc.) into each functional group. If you want to change permissions for a resource, you just need to modify the contents of the associated functional group.

With this configuration, permissions at the model level will not actually change, so you no longer need to reload the model. Using the **Users and Groups** API, you can modify groups and also promote or unpromote sessions at runtime without having to reload the model or the solution (as long as you do not **save( )** the directory changes).

The following diagram shows the functional group configuration principle:





*Note: Functional groups are only useful when managing groups at runtime, using the API. If you change groups in the Directory file using Wakanda Studio or save the changes to disk, the solution needs to be restarted, which means that functional groups are useless in this case.*

## Assigning Group Permissions

In the Datastore Model Designer or in the *Permissions.waPerm* file of your project, you can assign access rights and permissions to the groups for the various resources in your project.

When a logged user (or a guest access if the user is not logged in) attempts to execute an action and does not belong to the group that has appropriate access rights, a permission error is usually generated (status code 401). In the case of missing Read permissions on attributes, no error is returned but a **null** value is sent.

### Resources

You can assign specific permissions to the following resources in your project:

- Model
- Datastore classes
- Attributes
- Class methods
- RPC modules and functions
- Services
- Admin access.

Settings can be set either at a general level or at lower levels. For example, access permissions for class methods can be set at the level of the datastore model or at the level of each datastore class or method. When it is a general setting, each lower level inherits from the level higher up (shown in italic in the editor). This setting can be overridden at each lower level. In the following example taken from a datastore class, permissions are inherited from the Model level except for the **Remove** action, that is overriden for this class and assigned to the *Admin* group:



The permissions for the Admin group is discussed in the **Configuring Admin Access Control** chapter.

### Changing and Saving Permissions

Permission settings for a project are saved in a file whose extension is **.waPerm** located in your project's folder.

To assign permissions to datastore models, classes, and class methods, you can select the group name from the dropdown menu next to each action in the Properties tab of the Datastore Model Designer:



*Note: You can only assign one group to an object, which is why it is important to set up access groups in such a way that the most "powerful" users belong to all the groups at the higher levels in the hierarchy (see Users and Groups).*

You can also assign permissions by adding elements manually in the **.waPerm** file, which is in XML format. You define a permission in the following way:

```
<allow action={actionName} groupID={groupUUID}|groupName={groupName} resource={resourceName} type={resourceType}/>
```

For example:

```
<allow action="execute" groupID="ED38AEAF24598447B101956417588640" resource="people.Person.addPerson" type="method"/>
```

or:

```
<allow action="execute"  groupName="Manager"  resource="people.Person.addPerson" type="method"/>
```

You can use either the *groupID* or the *groupName* attribute to assign a permission to a group. Using the *groupID* is safer because this UUID will never change, even if the group is renamed. You need to open the .waDirectory file to get it, or use the **ID** property. The *groupName* is more 'readable' and you can get it directly from the Wakanda Studio Directory editor; however, if the group is renamed, you must not forget to update the permission settings accordingly.

### Permission Actions

Permissions are actions that the assigned group is allowed to carry out for a specific resource. For example, if you select the "Accounting" group for the "create" action of a datastore class, only members of the "Accounting" group are allowed to create entities in that class. The available permissions and their effects vary according to the type of resource.

*Warning: Not assigning a group to an action sets it free for all users. See implications in the **Implicit Permission Accesses to the Model** paragraph.*

### Datastore Class

The following permissions are available for a datastore class. They can be assigned at the global Model level (and then apply to all datastore classes) and/or at each datastore class level (and then apply to this datastore class only).

*Note: Some permissions grant implicit access to other permissions. For more information, please refer to the **Implicit Permission Accesses to the Model** paragraph.*

- **Read**: allows displaying the class's entities (**Describe** access is automatically granted to the group).
- **Create**: allows creating the class's entities.
- **Update**: allows displaying and updating the class's entities (**Read** and **Describe** accesses are automatically granted to the group).
- **Remove**: allows deleting the class's entities.
- **Describe**: allows accessing the model's or a specific datastore class's description (i.e. classes, attributes and properties, also called *catalog*). If the user (or guest if user is not logged in) tries to access the model or a class description without appropriate access rights, a permission error (status code 401) is returned: a login dialog is displayed and the requested page is not sent by Wakanda unless an appropriate access is provided.

  Basically, the **Describe** permission is required for **$catalog**, **$catalog/$all** and **$catalog/{datastoreClass}** REST requests. Since by default, Wakanda sends a **$catalog/$all** request before loading any page, if this permission is set for the model or one of the datastore class, no page can be displayed if the user is not logged and does not belong to the appropriate group. This setting has two main effects:
    - "force" all users to log in before using your application (in this case, all logged users must belong to the **Describe** group)
    - disallow any REST **$catalog** request that tries to access your application model.

  You can also define which datastore classes will be loaded on a page using the **WAF.Catalog** meta tag (see **Using WAF Data Access Only** paragraph). In this case, you can assign **Describe** permissions depending on the datastore class usage. For example, you can assign restricted **Describe** permissions to datastore classes that need an Admin access.

**Attributes**

The following permissions are available for an attribute:

- **Read**: displays the attribute value
- **Create**: displays and adds the attribute value in a new entity
- **Update**: displays and updates the attribute value in a modified entity

*Note: With respect to efficiency, permissions on attributes only work for REST requests, thus excluding RPC and HTTPRequestHandler requests.*

When a user does not have the permission to read an attribute, no error is generated but a **null** value is returned for this attribute to the user when they try to read the entity.

When a user does not have the permission to create or update an attribute and this attribute is created or edited client-side and sent back in a REST update request (PUT or POST), then an access error is generated and the whole entity is not saved.

For example, if you want to control access to the salary attribute, you can write in the *.waPerm* file:

```
<allow action="update" groupName="account" groupID="7421AAD5B99D2C4889F27C278E2C301B" resource="Model.Employee.salary" ty
<allow action="read" groupName="accessread" groupID="F981662341428E4DA8525B7D3E8DFE81" resource="Model.Employee.salary" t
```
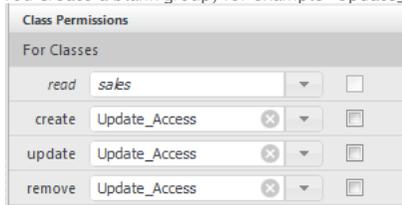
**Class Methods**

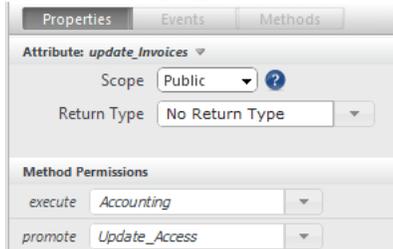For class methods, the following permissions are available:

- **Execute**: executes the method
- **Promote**: executes the method with specific access permissions

  Execute any datastore class method in the model with specific access permissions. For example, when you grant **Update** permission to a group for a datastore class, you do not have control over the update operations actually performed by the users. Even if you limit these operations through the interface, a user in this group can still modify data by sending requests to the server (for example, REST requests).

  In order to maintain complete control over the user's actions, you can associate a group (without users) to the **Promote** action so that operations that you want to control (adding and/or deleting entities, for example) are handled by code in a datastore class method. Only users in the group assigned the Execute action can run the method. But when this method is executed, it acquires the **Promote** group permissions (and not those of the **Execute** group). Of course, you must associate appropriate permissions (**Create** and **Delete**) with the **Promote** action group. With this setup, users cannot perform actions directly on data and are obliged to pass through the authorized methods.

  For example, if you want the members of the "Accounting" group to be able to create, modify, and delete invoices in a controlled framework:
  1. You create a blank group, for example "Update_Access", that you associate with the **create**, **update**, and **remove** actions for the *Invoice* class.



  2. You create a class method (named, for example, **updateInvoices**) that contains functions to add and modify invoices according to your business logic.
  3. You associate the **Execute** action to the Accounting group and the **Promote** action to the Update_Access group.



  When a user in the Accounting group executes the method, this method acquires the rights of the Update_Access group so that it can be executed.
  *Note: Usually, a group used for a **Promote** action should not have any users since it is a "technical" group.*

**RPC Modules and Functions**

For RPC modules and functions, the following permissions are available:

- **ExecuteFromClient**: RPC module or function is available and can be executed from the client.
- **Promote**: executes the method with specific access permissions (see "Class Methods")

For example:

```
<!-- Publish the whole "myRPC" module only for one group -->
<allow type="module" resource="myRPC"  action="executeFromClient"  groupID="3C9E1D3CC0C7BD43AECED581002B0AC6"/>
```

For more information about RPC methods settings, refer to the **Configuring CommonJS Modules for RPC** section.

### Services

You can control user access to the "upload" service using the permission settings, which is primarily used by the **File Upload** widget.

The **upload** service allows users of this group to use the "upload" service, which includes access rights to the associated "verify" and "upload" actions. A permission error is generated if a user not having the appropriate permission tries to upload a file.

For example, if you only want members of the "accessManager" group to upload files, you'd write the following:

```
<allow type="service" action="upload" resource="upload" groupName="accessManager" groupID="39000000000000000000000000000000
```

If you are in **Free Admin Mode**, you can write the following in your project's "Permissions.waPerm" file, which will allow anyone connecting to your project to use the "upload" service:

```
<allow type="service" action="upload" resource="upload" groupName="admin" groupID="01000000000000000000000000000000" temp
```

### Admin Access

Admin access is controlled through a specific group named "Admin" (which Wakanda creates by default for each new solution). Admin Access Control is activated by adding a user with a password to this group or at least two users without passwords.

Only users belonging to the "Admin" group can connect to Wakanda Server using Wakanda Studio or the **Wakanda Server Administration** interface.

For more information, refer to the **Configuring Admin Access Control** section.

Only users belonging to the "Admin" group are allowed to use the **Debugger** as well as run a server-side JavaScript file ( ▶ button).

### Implicit Permission Accesses to the Model

Keep in mind that when you do not assign a group to a permission property, this means that it is not restricted and therefore all users are granted access rights for that particular action.

Also, you have to pay attention to the fact that permissions are logically embedded, and therefore call a chain of implicit permission accesses:

- **Read** access group is granted **Describe** permission automatically.
- **Update** and **Remove** access groups are granted **Read** and **Describe** permissions automatically.

The following graphic shows the interactions between access rights in Wakanda:



Since Wakanda always grants the highest access rights to the user, you may need to define several permissions to make sure to control a specific access right. The following scenarios are provided to help illustrate this principle.

### Sample Scenario: Defining Read-Only Permissions

Suppose you want to give read-only access rights to the *finance* group. If you only assign **Read** permissions to the group in your model and leave the other actions untouched, all users will nevertheless be able to modify entities because the **Update** and **Remove** accesses remains free. If you want to define read-only access rights, you must assign groups to the **Update** permissions beforehand.

*Even though the* finance *group is assigned to the read permission,* **all users** *have read/write access rights because the Update and Remove accesses are not restricted:*



*Now only users in the* finance *group have* **read-only** *access rights:*



Of course, in this example, the *offshore dev* group should be included in the *finance* group, so that it has basic **Read** and **Describe** accesses.

**Sample Scenario: Controlling the Describe Access**

You want to activate the **Describe** permission so that guest users cannot access the catalog. Since **Read**, **Update**, and **Remove** groups are granted **Describe** access automatically, you need to fill in these permissions in order to control the **Describe** access.

*All users can create new entities; all other access rights are activated:*



If, for example, the **Remove** permission was left untouched, everybody would be able to read and thus to describe the datastore class.

**Forcing Temporary Permissions**

The **Force temporary permissions** option temporarily modifies the inheritance of permissions associated with a resource or a group of resources for testing or functional implementation without changing the overall access rights of your project. When this option is checked for an action at one level, all lower-level objects take on the access rights of this level and the individual setting is ignored.



For example, if you want the "Test" group to be able to create entities in the datastore model temporarily, you check the **Force temporary permissions** option at the datastore model level for the **create** action. You can then be sure that no user can perform this action unless they belong to this group.

**Permissions and Inheritance**

Since usually derived datastore classes do not have the same permission settings as parent classes, permissions are not inherited from parent classes.

When you extend a datastore class, its permission settings are not automatically reported to the derived class. You need to define permissions for each derived class.
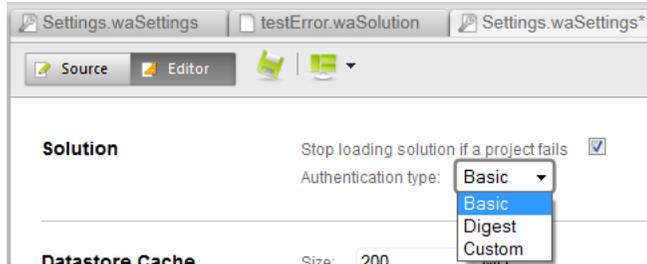
## Authenticating Users

Authentication is the process by which a user logging in to a Wakanda application attempts to identify himself/herself after sending his/her username and password. The user is identified by an existing account that has been previously defined either in the solution's Directory (the **.waDirectory** file) or in a custom user list. Wakanda offers several options for both the user **authentication** and **identification** phases:

- **User Authentication** (how a user logs in to a solution):
  - **Basic**: Allows you to take advantage of the built-in browser authentication features.
  - **Digest**: Allows you to take advantage of the built-in browser authentication features and encrypt data.
  - **Custom**: Offers you the most flexibility to design a custom authentication interface and process.

  The authentication mode is defined at the solution level.

- **User Identification** (how information provided by the user is controlled):
  - **Automatic**: based on the .waDirectory file.
  - **Customized**: based on a login listener function.

  The two identification modes can also be mixed.

### Setting the Authentication Type

The **Authentication type** parameter defines the way users will be able to enter their login information for a Wakanda solution.

You set the authentication type in your solution's .waSettings file. In Wakanda Studio, the Settings editor allows you to set the **Authentication type** property:



*Note: For more information about the .waSettings file, refer to the **Solution Settings File** chapter.*

To change the authentication type, select a new option, save your solution's .waSettings file, and restart the server.

The available authentication types are described below.

*Note: The Authentication type property is not taken into account for Web **Wakanda Server Administration** connections, which are always executed in Basic mode.*

### Basic Authentication

In Basic authentication mode, the way the user identifiers are passed to the server is based on the browser's built-in authentication features. The first time a browser sends a request to a protected resource in the Wakanda application, the following sequence is run:

- The server returns a 401 error accompanied by an authentication request.
- The user must enter his/her name and password in the browser's standard authentication dialog and click **OK**:
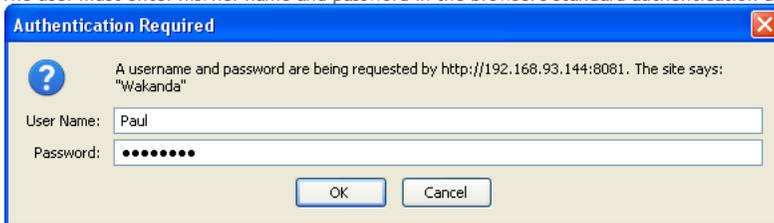


The user should then be identified on the server (see below).

In Basic mode, the name and password entered by the user are not encrypted when sent in HTTP requests, which means that security is less than optimum since they could be intercepted by a third-party. However, the risk of the username and password information being intercepted is low if the authentication sequence is sent in an HTTPS request (TLS-SSL), which is highly recommended.

### Digest Authentication

The Digest authentication type is quite similar to the Basic type: a user has to enter his/her identifiers in an authentication dialog.

Digest mode, however, offers higher security than Basic because authentication information is handled by a one-way hashing that makes the contents virtually impossible to decrypt.

*Note: Using either of Basic or Digest authentication modes is transparent to the user.*

### Custom Authentication (Default)

In the "custom" authentication mode, you as the Web developer design the interface, manage the data entry, and send the user identifiers.

Login identifiers are generally displayed at the top of the web application pages, thus allowing users to browse pages either logged in or not. To change from one mode to the other, your user clicks on a **Log in** or **Log out** button.



You have three different ways to manage and send user information to the server:

- The easiest way is to use the built-in Wakanda **Login Dialog** widget. For more custom integration, you can use its dedicated API (**Login Dialog**).
- If you want to use your own widgets or to have better control over the authentication process, you can design a Page that uses methods from the **Directory Class** WAF library.
- Design a Page that calls a datastore class method or an RPC function directly, which lets you customize the whole authentication process.

### Identifying Users

In Wakanda, you have two ways of identifying users and either granting or rejecting their access to your solution:

- **automatic identification**: the evaluation of user information is based on the .waDirectory file of your solution.
- **customized identification**: the evaluation of user information is based on a login listener function that can perform any challenge you want.

The following diagram summarizes the various authentication sequences:
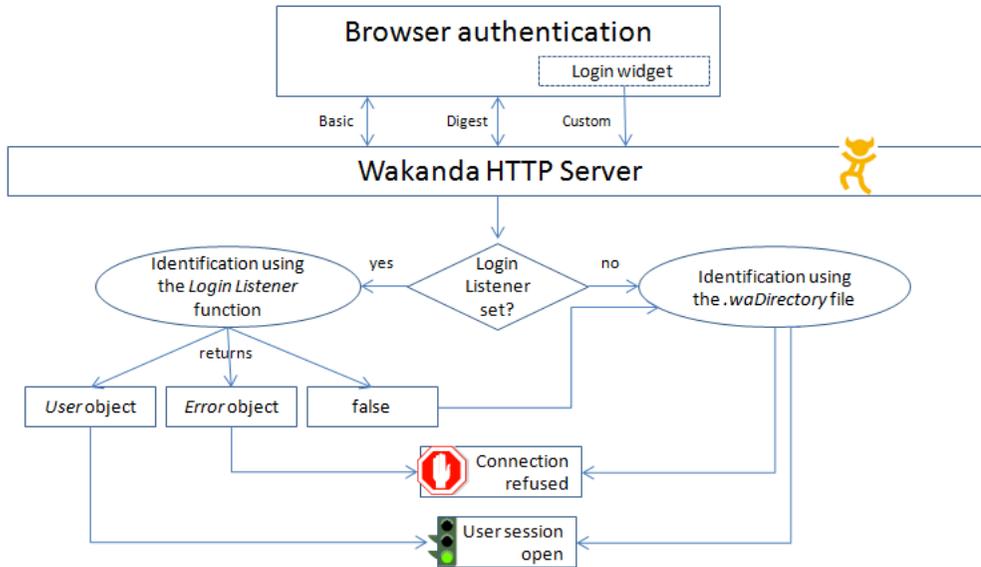


### Automatic Identification

Automatic identification means that the user identifiers will be evaluated by Wakanda server using your solution's directory (which is the **{solution name}.waDirectory** file). By default, if you use the **Login Dialog** widget, no code is necessary. Wakanda Server  automatically compares the received login information for the users defined in your solution's directory and either accept or reject the connection.

Automatic identification is used by default in Wakanda solutions: you just have to define users in the **.waDirectory** file. Automatic identification is used until a login listener function is set. In this case, you'll have to manage a customized identification sequence (see the authentication diagram).

*Note: You can also switch to automatic identification from a login listener function if it returns false (see below).*

- If you use the **Login Dialog** widget in your Page, identification is completely automatic; the entire identification process is managed by the widget.
- If you use a custom client-side login interface, you will have to write specific code using the **Directory Class** WAF library to compare user identifiers with the solution's directory.
  For an example, refer to the **login( )** method.
- You could also design a custom interface and call server-side code to compare user identifiers in the solution's directory. On the server, you call the **loginByPassword( )** method and pass to it the values entered by the user. The method returns true if the identification was performed successfully. Otherwise, it returns false. In this mode, you must use an SSL connection to guarantee a high-level of security.
  For an example, refer to the **loginByPassword( )** method.

### Identification using a Login Listener function

You can set a **Login Listener** in your Wakanda solution, using the **setLoginListener( )** method. In this case, all user login requests sent to the solution are handled by the listener function (even when the **Controlled Admin Access Mode** is active, login requests from Wakanda Studio or Server Administration to access administration features, such as starting/stopping the server). This feature allows you to manage the identification step based on a custom user directory or a remote server.

When the authentication request is sent to the server, it triggers a call to the Login Listener function installed by the **setLoginListener( )** method and passes the user name and password as parameters to this function. The function can validate the login using any custom code, an HTTP request, etc. After processing, the Login Listener function returns one of the following:

- **On success**: an object containing the authenticated user information (including the ID, name, full name, and group list). In this case, Wakanda dynamically creates a user in the solution and logs the user into the solution with the appropriate access rights. The user is created for the session and is not saved in the solution's directory.
- **In case of error**: an object containing two attributes: *error* and *errorMessage*. Access to Wakanda Server is denied.
- **In case of a *False* value**: the authentication is passed to the regular Wakanda login process. This allows you to combine custom and automatic identifications.

For more information and examples, refer to the **setLoginListener( )** method.

### Successful Authentication

Whatever the authentication and identification process, if the user login is validated, the following occurs:

- Code 200 is returned by the server,
- A user session is opened locally on the server, and
- A cookie is sent to the browser.

The user can then work in their session with the permissions granted from their associated group(s). By default, an inactive user session expires after 15 minutes.

If authentication fails (incorrect name and/or password), you can send a message back to the client to request that his/her information be entered again.

### Logging Out

For a user to log out from Wakanda Server, you can do one of the following depending on how you have set up your authentication system.

- If you defined a custom authentication and use the **Login Dialog**, you benefit from an automatic logout feature.
- If you use your own widgets, you can call the **logout( )** client-side Directory method, or even the **logout( )** server-side method.

- If you used Basic or Digest authentication, you do not have a dedicated interface to log out; therefore, you have to:
  - Close and reopen the Studio application from Wakanda Studio.

- Close and reopen the browser, or delete the user session cache.
  In some cases, when different authentication modes are used jointly, you may also need to delete the cookies to log out.

## Configuring Admin Access Control

**Free Admin Access vs Controlled Admin Access**

By default, the Wakanda Admin Access Control **is not activated** in newly created solutions, and therefore Wakanda Server runs in *free admin mode*.

**Free Admin Mode**

In *free admin mode*, all administration features, which include starting and stopping Wakanda Server, reloading model(s), debugging and running files, available in Wakanda Studio or through the Web **Wakanda Server Administration**, can be accessed without any restrictions.

The *free admin mode* is designed to simplify the administration of Wakanda during the early stages of development. However, once your web applications are online or when you want to control user access, we strongly recommend that you activate the **Controlled Admin Access Mode** (see below how to activate this mode).

*Note: In* free admin mode, *all users are automatically added to the Admin group during their work sessions. They are removed from the Admin group when the* controlled admin access mode *is activated.*

**Controlled Admin Access Mode**

The controlled access mode allows only users who are in the "Admin" group to have access to the following administration features:

- Start Solution/Shutdown Server (see **Toolbar**. (*If the authenticated user is not in the "Admin" group, the server can still be started with restrictive access and cannot be stopped.*)
- Reload models (see **Toolbar**. in the GUI Designer chapter and **Toolbar** in the Datastore Model Designer chapter).
- **Wakanda Server Administration** page (authentication must be done in the Web browser).
- Run a server-side JavaScript file ▶ (see **Toolbar** in the Code Editor chapter).
- Use the **Debugger**.

Defined login listener functions, if any, are called to identify admin users (see **Authenticating Users** and **setLoginListener( )**).
*Warning*: *To identify users for admin access using a login listener function, you must install the required.js file at the same level as the .waSolution file.*

To activate the Controlled admin access mode, you need to:

1. **Add at least one user with a password to the "Admin" group in the solution's Directory.**
   *Note: Controlled mode is also activated if you add two or more users without passwords to the "Admin" group.*
   For more information about adding users to groups, refer to the **Users and Groups** section.
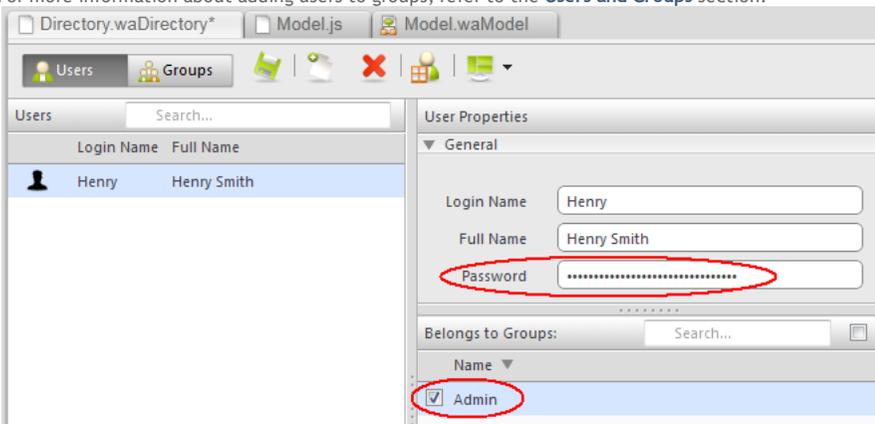


2. **Restart Wakanda Server**.

To disable the controlled admin access mode and return to free access mode:

1. **Remove all users from the "Admin" group (you can leave one user as long as he/she doesn't have a password).**
2. **Restart Wakanda Server.**

Debugging features can be disabled when Wakanda Server is launched through a command line. For more information, please refer to the **Launching Wakanda Server** section.

**Admin Access**

In secured Wakanda solutions, only users belonging to the "Admin" group are allowed to manage Wakanda Server through Wakanda Studio or the Web **Wakanda Server Administration** interface.

When an unlogged user tries to use an admin function for the first time, such as starting or stopping Wakanda Server (e.g., using the **Start** button ), an authentication dialog is displayed:
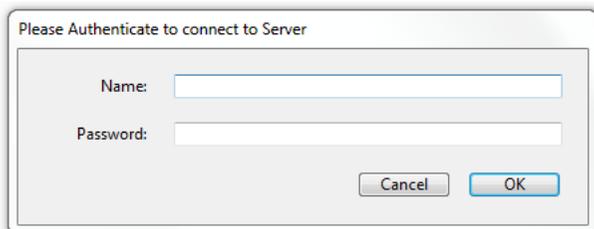


*Note: Admin users are always authenticated using the Basic type regardless of the Solution Settings.*

Depending on the Solution configuration, users are identified either automatically (based on the **directory's** contents), or through a custom **login listener** function. For more information, please refer to the **Authenticating Users** section.

Once user access is granted, the connection between Wakanda Server and Wakanda Studio (or the Server Administration web page) will then be established and an "admin" user session is opened on the server.

If the identification fails (the user does not belong to the Admin group or login information is not correct), the request is rejected and the requested admin function is not executed.

If identification fails, or if you click **Cancel** in the authentication dialog, you can still work on the Solution in Wakanda Studio; however, the connection with Wakanda Server is restricted.

By default, Wakanda solutions do not require authentication. Users can connect to Wakanda Server through Wakanda Studio or the Server Administration using a "default guest" access. This default mode provides free access to all Wakanda administration features. For that reason, we strongly recommend that you activate the Wakanda access control as detailed in the above section once a solution is published over a network or as soon as you want to protect your application.

# Security Best Practices

**Protecting Data from Unwanted Client Access**

Wakanda allows you to control the data for a specific datastore class that can be available client-side. Available data could depend on the logged in user's access rights or your application's business logic. In this context, you want to make sure that only authorized data can be seen client-side even when the user uses the Debugger or sends REST queries.

To meet these needs, Wakanda provides the highest level of security by combining three major features:

- **Scope for classes**: using this feature, you can state that a datastore class will never be available to browsers ("Public on server" scope). Datastore class scope is detailed in the **Datastore Class Properties** paragraph.
- **Extended classes**: this feature allows you to create 'cloned' datastore classes that will be dedicated to client availability. Data is shared with the parent class, but only selected data can be made available in the extended class. Data is selected using a restricting query. Class extensions are detailed in the **Extending a Datastore Class** section.
- **Restricting queries**: a restricting query is very useful with an extended datastore class because it defines which data will be made available in the class. A typical example would be a restricting query based on the logged in user's ID (see **Example of User-Based Shared Classes** below). Restricted queries are defined in the **Datastore Class Properties** paragraph.

For every datastore class that will be accessed client-side and for which you need to restrict access, you should follow these steps:

1. Set the scope of the base class to **Public on Server**.
2. Extend the base class and make this class **Public**.
3. Put a restricting query on the derived extended class.
4. Refer to the derived class in all client-side JavaScript code.
5. Refer to the base class for server-side JavaScript code.

This "best practice" is detailed in the following video:
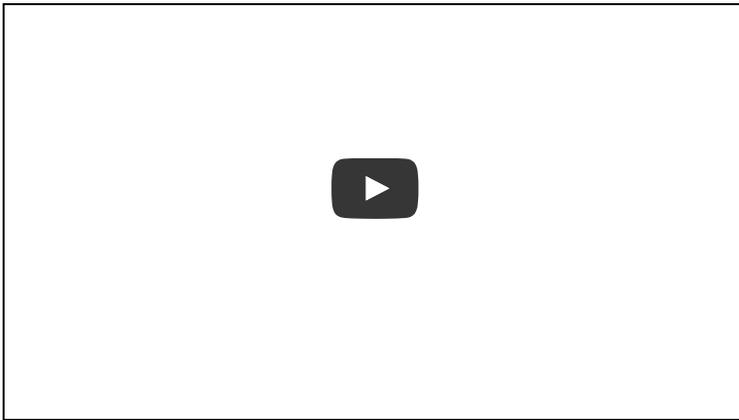


Of course, you do not need to follow these steps for datastore classes that do not need to be protected, or when users may need to load all the data from a base class.

**Example of User-Based Shared Classes**

The following example demonstrates the power and simplicity of user-based data access combined with Datastore class extensions.

We create a "Notes" application shared by several users where each logged in user will be able to write and work on their own notes, but must not see or edit notes from other users.

The datastore class model is defined as shown below:
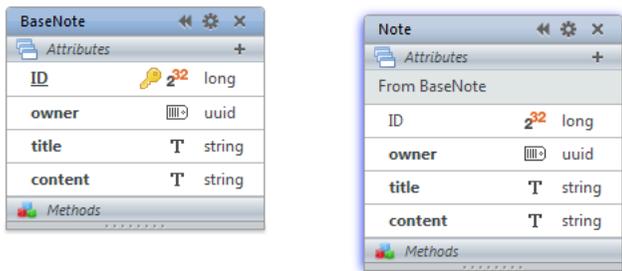


- The "Note" datastore class extends the "BaseNote" datastore class. The scope of "Note" is **Public** and can therefore be accessed from all the web clients. The scope of "BaseNote" is **Public on server**, making it only accessible on the server.

- In the "BaseNote" datastore class's **onInit** event, we write:

```
model.BaseNote.events.onInit = function() // when an entity is created
{
    var user = currentUser(); // we get the user
    if (user != null) // if a user is logged in
        this.owner = user.ID; // we save the user ID in the owner attribute
}
```

- In the "BaseNote" datastore class's **onValidate** event, we write:

```
model.BaseNote.events.onValidate = function() // when an entity is saved
{
    if (this.owner == null) //if the entity does not have a user
    {
        var user = currentUser(); // we save the user ID in the owner attribute
```

```
            if (user != null)
                this.owner = user.ID;
        }
        if (this.owner == null) // error if no user is identified
        {
            err = { error : 1, errorMessage: "The note's owner is empty." };
            return err;
        }
    }
}
```

- The "Note" datastore class is associated with the following restricting query:

```
owner = :$userid
```



Therefore, each time a logged in user accesses the Note class, he/she will only have access to his/her own notes.

- The interface only contains a standard login dialog and widgets that allow the user to browse and enter notes:



Both the Grid and Auto Form widgets are bound to the same datastore class datasource: note.
To allow the user to get his/her own notes, the following code is added in the Login Dialog's **On Login** event:

```
sources.note.all();
```

The above technique is all you need to do to share personal notes between users with a high level of security. Queries asking for all notes or even REST queries will only have access to the logged in user's data in the Note datastore class.