# Roles – manual

| | |
|---|---|
| Plugin | roles (for Elgg 1.8+) |
| Version | 1.0.2 |
| Author | Arck Interactive - András Szepesházi |
| License | GNU General Public License (GPL) version 2 |

## Introduction

The Roles plugin provides more granular control over your Elgg site than Elgg's default user distinction. Elgg basically recognizes two types of users: members (ordinary users) and administrators (users who can do everything on the site). There is a 3rd type of pseudo-user, the "not-logged-in users", who I'll refer to as visitor from now on.

More often than not, you'd need additional user types and associated privileges describing what each "user type" can do to your site. You might want to limit group creation privileges to certain users, or maybe you need special admin users (moderators) who can handle reported content, but can't access other admin features (like installing new plugins).

These are just two of the many use cases I can think of, but this – and much more - is absolutely possible with the Roles plugin. Before starting to use the plugin, please understand that the Roles plugin, in its current state, is rather a framework than a full stand-alone plugin. You'll need to make additional tweaks to make the above-described functionality of special roles work.

## Intended audience

Though future versions of this plugin might target admin users with no php skills, in its current state, this is for Elgg developers to implement user roles in an unobtrusive way. As of this version, there is no user-friendly way to create or configure roles on your Elgg installation with this plugin. Instead of managing roles and dragging role privileges around in a fancy admin interface, you'll need to get your hands dirty and create a php configuration array to fully utilize the possibilities of the plugin.

Having said that, if you're patient enough to read through this document, you'll be able to set up specific roles for your Elgg site - even with very limited programming skills.

Also, I expect this plugin to spring a number of child plugins, defining custom roles. So even if you don't want to absorb the dirty details, this plugin combined with future extensions might just provide you with the granular access you've always dreamed of.

## Roles from a developer's view

Roles are represented as relationships called "has_role" between users and role objects. At any given time, a user has either zero or one role relationship (multiple roles per user is not supported). If a user does not have a role relationship, he/she will automatically be associated with one of the default roles – more on that in the "Default roles" chapter.

In the current (1.0.2) version, role objects are automatically created from the role configuration array. Whenever there is a change to the configuration array, the role objects will be updated accordingly, without the need of running any upgrade script manually. However, the role changes will not take effect until you reload any page on the site as a site administrator after saving the role changes. The new page request will initiate an update to all role configurations, but this will only work if you are logged in as admin.

Removing a role from the configuration array will not remove the corresponding role object. Only permission changes to existing roles will be updated, and new roles will be created automatically.

To completely remove and replace all role objects with a new configuration set, disable, then re-enable the roles plugin. Note that this will also destroy any user <-> role relationship, i.e. all users will lose their present role. Warning: This means that developers need to be especially careful when modifying the roles plugin code in environments with existing roles since Elgg may automatically disable the plugin if it cannot start. If you're unhappy with this behavior, just remove the relevant part from activate.php – it's only a couple of lines of code.

## *The role object*

Role objects are instances of ElggRole class (extending ElggObject class), having the following properties:

- name: A unique string identifier of the given role (staff, super user, etc)
- title: Displayed name of the role
- extends: A flat array containing other role names that this role extends. These will be processed in the array's natural order when the role object is created or updated.
- permissions: A serialized array describing all permission for the given role. More explanation on permissions is in the Configuration chapter.

Please note that role extension does not work in a dynamic, on-the-fly way. Extension is a tool to simplify rule-set definitions: by extending role2 with role1, you can say that "role2 is very similar to role1, except for the following rules: ..." – and you only need to define the rules you need to override from rule1. All role extensions are resolved when creating (or updating) the role object from the configuration array.

There is a useful method attached to each role object called

```
public function getUsers($options)
```

which will return the list of users for the current role object. The array $options accepts the same key => value pairs as the elgg_get_entities* functions.

## Default roles

There are 3 default (built-in) roles that come with the plugin: VISITOR_ROLE, DEFAULT_ROLE, and ADMIN_ROLE. These roles will be automatically associated with users (or non-logged in visitors) who do not possess a specific role.

The default roles were created based on the assumption that for most users, the simple "member" role - that comes with Elgg by default - would be sufficient. Since we only need to store role relationships for users having a non-default role, the number of role-related relationships in the database is significantly reduced. At the same time, we can still define specific rules for these default, built-in rules.

## Getting and setting user roles from code

The intention of this module is that plugin authors should be able to develop most functionality without the explicit need of knowing a user's certain role. Most of role-based functionality – i.e. what the user can see and interact with – can be moved to the configuration array; hence no need for handling role based conditionals in the code.

In rare cases, when you do explicitly need to know what role a user has, you can get it by calling the

```
function roles_get_role($user = null)
```

which will return the corresponding role object for the "$user" parameter. If $user is not passed, it will return the role object for the currently logged-in user. Note that this function is guaranteed to return an ElggRole object – if the user does not have a specific role, one of the default role objects will be returned.

Even more unlikely that you'll have to explicitly set a user's role, but it's still possible with the

```
function roles_set_role($role, $user = null)
```

which expects a role object, and optionally the user whose role needs to be set. If $user is not passed, it will set the role for the currently logged-in user.

For getting all role objects, or a specific one, use the following two functions:

```
function roles_get_all_roles()
function roles_get_role_by_name($role_name)
```

## Setting user roles via user settings page

There is a specific view (roles/settings/account/role) that implements a role selector control. This is essentially a drop-down box with all the defined custom roles in the system, and an additional item called "No specific role".

By default, this view is only added to the user settings page if the current user has the default ADMIN_ROLE. Of course, the beauty of this plugin that you can define other roles, that may have the same privileges for changing user roles.

## Configuration

### *The configuration array*

As of this version, there is no fancy admin interface to set up role permission; this has to be done via an associative configuration array. The array for default roles resides in the mod/roles/lib/config.php file, and has the following structure (example not showing all configuration details):

```
$roles = array(
       VISITOR_ROLE => array(
              'title' => 'roles:role:VISITOR_ROLE',
              'extends' => array(),
              'permissions' => array( ... )
       )
       DEFAULT_ROLE => array(
              'title' => ' roles:role:DEFAULT_ROLE',
              'extends' => array(),
              'permissions' => array( ... )
       ),
       ADMIN_ROLE => array(
              'title' => ' roles:role:ADMIN_ROLE',
              'extends' => array(),
              'permissions' => array( ... )
       ),
);
```

You can override permissions of any of the default roles, or add new roles to the configuration. The recommended way to do this is by creating a lightweight plugin called "roles_*your_role_name*", and register for the ("role:config", "role") plugin hook with a priority greater than 500. Your hook will be triggered by the roles plugin and expect role definitions to be merged into the original role configuration array.

Here is an example of how to implement the "role:config" hook:

```
function myroles_config($hook_name, $entity_type, $return_value, $params) {

       $roles = array(
              'limited_users' => array(
                     'title' => 'yoursitename_roles:limited_users',
                     'permissions' => array(
                            'actions' => array(
                                   'groups/save' => array('rule' => 'deny')
                            ),
                     ),
              )
       );
       if (!is_array($return_value)) {
              return $roles;
       } else {
              return array_merge($return_value, $roles);
       }
}
```

The above code adds a new role to the system identified as "limited_users". Members of this role will not be able to create groups, as defined by the permission section of the configuration array. Permissions will be explained in more detail later.

The above "$roles" associative array could contain any number of new roles and override definitions for the default roles (VISITOR_ROLE, DEFAULT_ROLE, and ADMIN_ROLE). The first level keys (in the above example the single role key "limited_users") should be unique within the Elgg installation – this key is the primary identifier for roles. When another plugin defines the same role with different rule sets, the later definition will override earlier definitions.

Note how the above hook function uses graceful role definition: instead of just returning the new role definition array, it merges with the previously existing array. This is how your role extension should generally behave, letting other plugins define other sets of roles.

The "permissions" section holds individual permission rules that determine what the user can see and interact with on the site. Permissions can contain sections of rules relating to menu items, views, pages, actions, plugin hooks and events. For most of these permission sections, the basic permission rules are as follows:

- `deny`: Deny access to a specific item. In some cases this will generate an error message and result in a redirect (for actions and pages), other cases the given item will simply not be rendered (for views and menus). In case of hooks, the specified hook handler will not be triggered. In case of events, you can unregister an event handler.
- `allow`: Allow access to a specific item. This rule is most useful when extending rules – the default role can deny creating new groups, while an extension of the default role can specifically re-allow group creation.
- `extend`: Add a new item on the fly to the current page – this works for hooks, events, menus and views. I.e. you can add role specific menu items, extend existing views and create new plugin hooks or event listeners, just by using the right configuration values.
- `replace`: Replaces an existing item. Works for views, menus, hooks and events.
- `redirect`: Redirects to another page. Works for pages.

## Menu permissions

Menu items can be dynamically removed from or appended to any Elgg menu. To disable the “Members” main menu item for the default role, the configuration would look like this:

```
DEFAULT_ROLE => array(
	'title' => 'roles:role:DEFAULT_ROLE',
	'extends' => array(),
	'permissions' => array(
		'menus' => array(
			'site::members' => array('rule' => 'deny')
		),
	)
)
```

The “menus” configuration section can contain any number of menu rules, each using the “menu-name::item-name” structure as key, for identifying the menu item to act on. A more complex example shows the menu section from a role configuration that extends the default role:

```
'menus' => array(
	'site::members' => array('rule' => 'allow'),   // 1st menu rule
	'site::books' => array(                        // 2nd menu rule
		'rule' => 'extend',
		'menu_item' => array(
			'name' => 'books',
			'text' => 'Books',
			'href' => 'books/all',
		)
	),
	'site::groups' => array(                       // 3rd menu rule
		'rule' => 'replace',
		'menu_item' => array(
			'name' => 'mygroups',
			'text' => 'My Groups',
			'href' => 'groups/member/{$self_username}',
		)
	),
	'filter::friends' => array(                    // 4th menu rule
		'rule' => 'deny',
		'context' => array('bookmarks', ’files’)
	),
)
```

The first rule will override the default role's ruling on the Members menu item – it will be visible again for those users who have the above role.

The second section demonstrates how to add a new menu item to any given menu dynamically for a role. The rule says to extend the selected menu with a new item called "books", and the new item itself is defined after the rule, under "menu_item". The menu_item associative array expects the same key-value pairs as the ElggMenu::factory() method. To put it short, this rule will add a new menu item called "Books" to the main menu, and will link to the books/all page. Of course, you'll still have to actually implement the content of that page.

The 3rd rule shows how to replace an existing menu item in a menu. For this role's members, the standard "Groups" link in the site menu will be replaced "My Groups", pointing to those groups that the current user is a member of. Note the dynamic variable in the href part: "`{$self_username}`". This reference will be replaced with the currently logged-in user's username.  Read more about dynamic page paths in chapter "Using dynamic paths".

The 4th rule shows how to create context-dependent rules. The rule removes the Friends item from the "All, Mine, Friends" content filtering menu, leaving only "All" and "Mine". However, as there is a context defined for this section, the rule will only kick in when the current context is one of the predefined contexts. In other words, the "Friends" item from the filter menu will only be removed on the Bookmarks and File pages.

## Pages permissions

Accessing pages can be denied or allowed per role; pages can also be silently redirected by the rule set.

```
'pages' => array(
	'groups/add/{$self_guid}' => array(
		'rule' => 'deny',
		'forward' => 'groups/all',
	)
),
```

The above rule will deny access to the "create new group" page, will register an error message to inform the user, and forward him/her to the "Groups" front page. If no forward address is specified, the user will be forwarded to the referrer page. Note that you can use the same dynamic substitutes as described at the Menu permissions section.

The rule "forward" behaves the same way as "deny", except it does not register an error message – it will simply silently forward the user to another page.

The "allow" rule will override a previous role's "deny" or "redirect" rule for a given page path.

## Views permissions

Views can be suppressed, extended or replaced for any roles. Consider the following examples.

```
'views' => array(
       'input/password' => array('rule' => 'deny'),   // 1st view rule

       'forms/account/settings' => array(             // 2nd view rule
              'rule' => 'extend',
              'view_extension' => array(
                     'view' => 'roles/settings/account/role',
                     'priority' => 150
              )
       ),

       'object/blog' => array(                        // 3rd view rule
              'rule' => 'replace',
              'view_replacement' => array(
                     'location' => 'mod/modified_blog/extended/views/',
              )
        ),

)
```

The first rule simply suppresses the “input/password” view. This is a rudimentary way to disable password changing for certain roles. The view will simply not be displayed, without any warnings or error messages.

The second rule shows a more useful example of roles based view management: it extends the “forms/account/settings” view with a new one, “roles/settings/account/role” – which happens to be the role selector box. In other words, this rule will allow members to change their (and other users’) role. The “extend” rule behaves the same way as an elgg_extend_view() call would, but now the view extension is moved from source code to role configuration.

The third rule shows how to replace an existing view for a certain role. The view replacement works the same way as a call to the elgg_set_view_location() function would, the location being relative to Elgg’s installation path. In the example, we replace the blog view with a different one for this role.

## Actions permissions

Actions have pretty simple rule options: allow and deny. To disable access to a specific action, use:

```
'actions' => array(
       'registered/action/name' => array('rule' => 'deny')
)
```

In case of trying to access a non-permitted action, an error message will be registered, and the user will be forwarded to the referring page.

## Hooks permissions

Plugin hook handlers can be registered, unregistered or replaced for any roles.

```
'hooks' => array(
	'usersettings:save::user' => array(
		'rule' => 'extend',
		'hook' => array(
			'handler' => 'roles_user_settings_save',
			'priority' => 500,
		)
	),
),
```

The example above will register a new handler for a role that will be triggered by the "usersettings:save", "user" hook. Obviously, you'll still have to implement the handler function. The above example, as is, is used by the roles plugin for the default ADMIN_ROLE. The handler implementation takes care of saving the user's role, if it has been changed on the user settings page.

Hooks should always be identified either by the hook name, or the hook name and the hook type, separated by a "::" (double colon). If the hook type is not defined, the type "all" will be assumed.

When you "deny" a hook, the roles plugin will basically unregister the given hook handler, or all handlers for the specified hook.

```
'hooks' => array(
	'usersettings:save::user' => array(
		'rule' => 'deny',
		'hook' => array(
			'handler' => 'user_settings_save',
		)
	),
),
```

This will unregister the default Elgg handler for saving user settings, meaning that members of this role will not be able to save any changes to their settings page. Admittedly not a brilliant example, but you should get the general idea. A bit better example would be to replace the default hook for saving user settings, like this:

```
'hooks' => array(
	'usersettings:save::user' => array(
		'rule' => 'replace',
		'hook' => array(
			'old_handler' => 'user_settings_save',
			'new_handler' => 'custom_user_settings_save',
		)
	),
),
```

## Events permissions

Event permissions work much the same way as hooks; event handlers can be registered, unregistered or replaced for any roles. An example for registering an event listener via the configuration array

```
'events' => array(
        join::group' => array(
                'rule' => 'extend',
                'event' => array(
                        'handler' => 'custom_join_group_handler',
                )
        ),
),
```

Whenever a member of the above defined role will join a group, the “custom_join_group_handler” function will be called, where you can do whatever you need to do – for example notify all group members of this fact.

## Using dynamic paths

For action paths and page request URIs, it can be useful to have more advanced path definition possibilities than declaring a static string. You have two options to achieve more complex path matching: using built-in variables and regular expressions.

You can use the following variables in any URL or path-like parameters:

| | |
|---|---|
| • `{$self_username}` | The username of the currently logged-in user |
| • `{$self_rolename}` | The role name of the currently logged-in user |
| • `{$self_guid}` | The GUID of the currently logged-in user |
| • `{$pageowner_username}` | The username of the current page owner |
| • `{$pageowner_rolename}` | The role name of the current page owner |
| • `{$pageowner_guid}` | The GUID of the current page owner |

The above variables will be replaced when resolving page, menu and action paths. You can see an example of this at the “Menu permissions” chapter. You can also use regular expressions for your path definitions, as seen below:

```
'actions' => array(
        'regexp(/^admin\/((?!user\/ban|user\/unban).)*$/)' => array('rule' =>
'deny')
)
```

The above action rule will match all “admin/*” actions, except for “admin/user/ban” and “admin/user/unban”, and deny access to all matching actions. In other words, we just created an admin user with very limited capabilities, the user will be able to ban and uban other users, but won’t be able to perform any other administrative actions.

When using regular expressions, you’ll have to wrap your pattern string in a “regexp( )” pseudo-call. This will tell Roles’ rule parser to interpret the expression inside as a regular expression.

When combining the above two techniques, i.e. using both built-in variables and regular expressions, please remember that first the variables will be resolved, and the resulting, substituted pattern will be passed to the regular expression matcher.

## Guidelines for role implementations – using real world examples

As you can see from all the configuration options, you have a pretty flexible set of tools to implement different roles for your Elgg users. For most purposes, you'll have to use a combination of rules to achieve your goals in a secure way.

### *Implementing the Group Administrator role*

Consider the following, commonly requested role: only privileged members should be able to create and edit groups. To implement this, you'd need to take the following steps.

- Override the default role.
- In your default role definition, deny access to the "Create a new group" menu item on group pages.
- Also deny access to the "Groups I own" menu item in the sidebar (so you won't confuse your ordinary members with a dead link).
- Deny access to the "groups/add/{$self_guid}" page.
- For security reasons, deny access to the groups/edit action that is called both when creating a new group or editing an existing one.

With the above rules, you practically shut down every access to group creation for default members. Now all you have to do is to define another role for group administrators (who can create and edit groups). The rule set of this role can be totally empty – as this is a new role, the above-defined restrictions do not apply to members of this role. They will simply be able to access all those pages, actions and menus just as any member in a vanilla Elgg installation.

To make someone a group admin, simply visit (as an admin user) the target user's settings page, and set his/her role to the newly created group administrator role.

Additionally, if you want to make this feature really convenient, you can add new menu items to the hovering context menu that can be opened in the corner of member icons. This is slightly more complicated, here is how you can do it.

- Override the default admin role.
- Add a new handler to the hook section of the configuration array by extending the ("register", "menu:userhover") hook.
- Implement the handler function.
- Inside your handler, check for the current role of the user (user is passed as parameter from the hook trigger).
- If the current role is group administrator, add a new menu item to the hover menu called "Revoke group admin privileges". If the current role is the default member role, add a new menu item to the hover menu called "Grant group admin privileges". Link these menu items to two separate actions.
- In your action implementation, set the user's role explicitly to group administrator or default member, as necessary.

There is a full working example of the above role configuration at the Elgg Community. Look for the plugin “roles_group_admins”.

## *Implementing the Moderator role*

Another very common request I saw on the community is the ability to elevate certain members to Moderators. Moderators can edit content created by other users, and ban/unban them. However, they aren’t able to perform any other administrative actions, like installing plugins or changing site settings.

The natural approach from Roles point of view is to extend the default ADMIN_ROLE, and gradually revoke most of the administrative rights via “deny” rules.

- Create a new role called “moderator”, overriding the default admin role.
- In this new role, deny access to all admin pages, except the reported content pages. This can be achieved by using regular expression for the page matching pattern.
- Remove the generic “Administration” link for the top bar by denying access to the “administration” menu item in the “toolbar” menu.
- Add a new menu item to either the topbar menu or the site menu that links directly to the reported content page.
- Deny access to all administrative actions except for banning and unbanning users. This, again, can be done by using regular expressions for the action matching.
- Finally suppress the whole view displaying the administrative sidebar. This is to remove all administrative shortcuts when showing the reported content page.

And that is all. You can now make a Moderator from any user by first elevating the user to admin, then setting his/her role to Moderator. There is a full working example of the above role configuration at the Elgg Community. Look for the plugin “roles_moderators”.

## *Conventions for custom role developers*

If you’re implementing a specific role and you think that it might be useful for others as well, by all means please release it on the community site. Here are a couple of tips on creating a role extension plugin:

- Start your plugin’s name with “roles_” – an example is “roles_group_admins” for the previous example.
- Use a pattern like “My specific role for Roles” for your plugin title, so role-related searches will find your awesome extension
- In your manifest file, use the “<requires>” tag to indicate that your plugin is dependent on the roles plugin.
- In your role configuration hook, use graceful role definition: instead of just returning the new role definition array, merge it with the previously existing array.
- Keep security in mind. It’s usually not enough just to remove an item from a given menu – you’ll want to deny access to corresponding pages and actions as well.

## Guidelines for roles-aware plugin implementations

If you're a plugin developer, it might make sense for a lot of projects to make your plugin roles-aware. Site owners might want to restrict part of the functionality your plugin offers to certain set of users – roles provides an easy way to deal with that wish.

If you decide that you want to leave the door open for roles configuration on the features of your plugin, here is an approach for that. It requires a bit of extra work, but it's not too excessive.

- In your init function, check if the roles plugin is available and enabled in the current environment.
- If the roles plugin could not be found, execute your usual init steps – register menu items, extend view, register actions, etc. as necessary.
- If the role plugin is enabled, leave view and menu registration out of the init function, as this can be done via role configuration. Register all actions as public – you'll be able to restrict those actions via configuration. Avoid using elgg_is_logged_in() and elgg_is_admin_logged_in() calls in your code, as you can have finer grains of control, again via the configuration.
- Register for the "roles:config", "role" hook to provide your own set of rules, with a priority greater than 500. This is to make sure that default roles will already be initialized by the time your hook handler is triggered.
- Merge into the existing roles the rule sets for menus, actions, etc as your plugin needs. All you need to handle are the default roles (VISITOR_ROLE, DEFAULT_ROLE, ADMIN_ROLE)
- Register your menu items and view extensions for the appropriate default roles.
- Restrict access to actions and pages by adding "deny" rules for roles you'd like to exclude from certain activities. To give you an example: you can register an action that should only be available for administrators as public, but afterwards add "deny" rules for that action for the VISITOR_ROLE and DEFAULT_ROLE.

Why would you put in the extra effort? Because you leave open the possibility for others to override some of your rules. So, content that is available to all members from your plugin in one installation could be restricted to users of a certain role in another installation.

## The future of Roles plugin

This functionality clearly belongs to the core, so the implementation, as a plugin, is a bit hackish by nature – it's hard to implement a generic roles framework if the concept is not part of the core.

As of writing this document, Elgg core developers do not seem very keen on bringing roles into the core. I can very much understand this, as it would introduce heavy complexity to Elgg. I've checked all open Elgg tickets, and none of them for the foreseeable roadmap mentions anything about role implementation. So however needs a finer grain of control, for now is stuck with this implementation (or any other fork this implementation might inspire).

The next big step for this plugin would be to move – at least part of – the configuration from php arrays to a user-friendly admin interface. Creating and editing roles would be very straightforward; handling role extensions would also not present any issues. On the other hand, managing the privileges for roles is when the almost unmanageable complexity comes in.

The administrator of roles would probably like to see privileges like “can create groups”, “can write internal message to any members”, “can write internal message to friends only”, “can edit other users’ blogs but can’t access the general admin area”, etc. This all can be implemented for core functionality, but once plugins come into the picture, it is impossible to guess what rules could be applied to them.

Anyways, for now, this is what it is: a framework for implementing custom roles that requires a bit of bare-hand configuration for specific needs. We’ll try to create and gradually extend a user-friendly interface for managing roles – stay tuned for updates.

Enjoy!