PostOpus
Webdevelopment

Yii - Access Control Lists
in a nutshell

dispy
thanks to Qiang Xue and the folks in #yii

August 10, 2012

Contents

1 Features

- Multiple groups for each object (both aco and aro)
- Unlimited depth in hierarchy
- Non-recursive permission lookup
- Convenient, integrated interface
- Transactions and exceptions upon critical errors
- (optional) Implicit conversion between aro and aco
- (optional) AJAX Permission Management
- (optional) Advanced group/query filtering capabilities
- (optional) Business Rules for both acos and aros
- (optional) General Permissions
- (optional) Support for access control filters with backward compatible syntax
- (optional) Fine grained control on the grant permission
- (optional) Fine grained control on the relation changes (join/leave)

2 Limitations

- Due to parametrized static method calls this extension is limited to PHP versions ¿= 5.3.0.
- this extension supports solely positive rights-management =¿ the access check only checks if you are explicitely allowed to do something, it doesn't checkif you are explicitely denied access
- A fellow told me that INNER JOINs have a different syntax (regarding " and so on) on different DBMS. The extension was developed with MySQL and was never tested with competitors such as Oracle or PGSQL but it should be an easy task to adjust the things if I knew what to change.
- The system currently requires numerical primary keys which are all named "id" for both your ACOs and AROs. I keep my eyes peeled but nothing has shown up yet to remedy that in an easy way.

3 The Concept of Access Control Lists

3.1 Objects

3.1.1 ACL-Objects

The following will probably seem very abstract and at some point overkill at the first glance. However, if we put up with it, we'll be bestowed with more powerful behavior later on without any additional effort.

ACL-Objects are in general all the objects somehow involved with access control.
These objects have a unique identifier, which you can use to retrieve them, alter them and save them back to the database. You can assign them a kind of label in order to facilitate their retrieval: this label is called *Alias*. They might just dangle in the database waiting for you to perform something with them, but they might as well be associated with a real-world entity in your database. If that is the case, you can also use this entity to retrieve the object (and vice versa). [1].

[1]This is what the ACL-extension does internally: Whenever you use a real-world entity, it is resolved to it's internal acl-object-representation. In most cases that is sufficient, however you can always access the acl-objects directly.

Methods of Identification:

1. **The Object-ID / the object itself**
   Every ACL-Object is in fact implemented as a regular CActiveRecord you can retrieve, alter, save and delete just as you can with any other using its unique ID.

```php
<?php
        $myObject = CGroup::model()->findByPk($id);

       //use the object directly
       whateverAclMethod($myObject);
?>
```

2. **The Alias**
   You can (but you don't have to) assign an *Alias* to every acl-object. This facilitates you to access this object later on, because you can use the alias instead of the ID. [2]

```php
<?php
        //Either use the alias directly
       whateverAclMethod('myAlias');

       //Or use the array-syntax
       whateverAclMethod(array('alias' => 'myAlias'));
?>
```

3. **The Real-Word-Entity**
   Almost always when an acl-object is expected as a parameter you can use a real-world entity instead (simply the CActiveRecord-instance).

```php
<?php
       //use the object directly
       $myObject = WhateverModel::model()->find(...);
       whateverAclMethod($myObject);
       //Or use the array-syntax
       whateverAclMethod(array('model' => 'myModel', 'foreign_key' => $myId));
?>
```

ACL-Objects can be located in *Hierarchies*.
Hierarchies of acl-objects provide a certain level of *Inheritance*: Each acl-object will inherit the position of its parent and thus act as if it was his parent. Acl-objects can have an arbitrary number of parents, the depth of the hierarchy is not limited. Whereas the lookup-operations perform linearly (and thus don't slow down with an increasing hierarchy-depth), the insertion of acl-objects can have a poor performance.
[3] Both of the following objects are AclObjects and therefore can use its features. The main methods you can perform on an acl-object:

```php
<?php
abstract class AclObject{
       /**
      * Joins the given object (now called: group)
```

[2] It is also necessary to set an alias if you want to use Business Rules with this record. More on this in the chapter on Business Rules.
[3] Background: Each aco-object copies all of its children in a recursive manner if it joins a new group. This shouldn't be a problem if you don't have crazy setups.

```
     * @param mixed $obj
     * @return boolean
     */
    abstract public function join($obj);

    /**
     * Leaves the given group
     * @param mixed $obj
     * @return boolean
     */
    abstract public function leave($obj);

    /**
     * Checks whether this object is somehow a child of the given object
     * @param mixed $obj
     * @return boolean
     */
    abstract public function is($obj);

    /**
    * Fetches all parent-objects and returns them in an array
    * @return array[AclObject] the parent-objects
    */
   public function getParentObjects();

   /**
     * Loads the associated object, if possible, and returns it
     * @return mixed   NULL or a child of CActiveRecord
     */
    public function getAssociatedObject()

    }
```

You can specify parents in

```
<?php

static Model::$autoJoinGroups
?>
```

to be automatically joined by an object upon its creation.
Example:

```
<?php

class MyModel extends CActiveRecord{
        public static $autoJoinGroups = array('All', 'anotherAlias',
        array('model' => 'anyModel', 'foreign_key' => 123456));
}
?>
```

3.1.2 Access Control Objects

Access Control Objects (referred to as: Acos) are basically all objects and/or entities which are under access control. That means, the access on them is restricted - not everyone can perform everything on every object, but he needs to have the *Permission* to perform an *Action* on a specific *Access Control Object*.

You cannot perform any action on an arbitrary object: e.g. you can burn your cigarette but you couldn't burn your sausage. Well, perhaps you could; but it doesn't make much sense. Therefore, if you use a real-world-entity as an aco, you can limit the actions one can perform on it using :

```
 <?php
/**
     * This contains all possible actions for the objects of this class.
     * All other actions will never be granted and always denied upon inquiry.
     * Example: array('update', 'read') << all other action are not available
     * @var array[string]
     */
    public static $possibleActions = NULL;
 \?>
```

Because every aco-object is also an acl-object, if someone has specific access to one or more of an aco objects parents, he also has access on this object.
The bare acl-equivalent of acos is the CActiveRecord-class *CGroup* (ControlGroup).

The operations on Aco-Objects:

```
<?php
class RestrictedActiveRecordBehavior{
        /**
     * generates a name unique to this Behavior-instance
     * @param   string  the tableName
     * @return  string  the unique tableName
     */
    public function getUniqueName($table);

    /**
     * Gets the Aros who are directly (no inheritance!) permitted to perform
     * one of the specified actions on this object
     * @param mixed £actions the actions to be considered
     * @return array All of the objects which have one of the permissions
     */
    public function getDirectlyPermitted($actions = '*');

    /**
     * Checks whether the current ARO has the given permission on this object
     * @param string £permission
     */
    public function grants($permission);
}
?>
```

The *RestrictedActiveRecordBehavior::getUniqueName* method may seem quite obscure. Whenever you query an object under access control from the database, a deluge of other information is retrieved a) in advance and b)simultaneously (= in the same query).
As you can query several models in a row (using relations, with()), the requested information must use separate names to distinguish the different results. To do this, the whatever access information is queried, an ID unique to the query will be appended so that all subqueries of a query are distinct. In fact, the extension provides a wrapper for those intricacies, so you do not have to bother with them. See on page **??**.

3.1.3 Actions

Whenever sometihng is *done* to an *Aco*, an *action* is performed.
Actions are also regular *CActiveRecord*-objects. Unlike the other acl-objects, you can edit them directly in the database, as well as add new ones and delete others. The main and only property of an Action is its *name*.
There are some shipped default actions:

- "create" - used if acos are created
- "read" - used if acos are requested
- "update" - used whenever an aco is saved and it is not a new record to the database
- "delete" - used upon deletion of aco-objects
- Configurable, optional actions
  - "grant" - used if another action is granted to an Aro
  - "grant_*" - used to grant an aro to grant a specific action (read twice ;))
  - "deny" - used if an action is denied to an Aro
  - "deny_*" - used to grant an aro to deny a specific action
  - "join" - used if an acl-objects joins another one
  - "leave" - used if an acl-object leaves another one
  - "is" - virtual action used for Business-Rules

(all Actions after "delete" are optional and must be activated using *config.php*)
You should neither delete those actions nor should you use them manually. You can simply add your own actions in the database - they will be available immediately as of your insertion.

Actions can be determined using different ways:

```
<?php
        //The standard method
        $single    = 'create';

        //a set of actions
        $actionSet = array('create', 'read');

        //The wildcard method
        //Equals the one above if CRUD are the possibleActions of the model
        //The + operator is also supported though there is no known use yet ;D
        $actionSet2 = '* - update, delete';

?>
```

Note that most (not all) methods accept all of the three variants.

3.1.4 Access Request Objects

Access Request Objects (referred to as: Aros) are all the objects requesting access on an aco.

Whatever Aros do, they do it in terms of *Actions on Acos*. This doesn't have to happen explicitely, but in most cases it works in the background: the standard CRUD actions are performed behind the curtains. If you do not explicitely specify an aro to perform an action, the extension tries to find its own following this scheme:

1. **Yii::app()->user**
   If this property is not empty, the extension tries to load the aro using the id from this CWebUser instance, the assumed model is "User". [4]

2. **Guest** If the first step failed, because the user is either not logged in or couldn't be retrieved from the database, it is assumed that the user is a guest and his permissions equal the ones of the guest-group (default: "Guest"). If this step fails, the algorithm will fail altogether. If you don't have an appropriate acl-object, simply create one such as "None". You cannot perform anything without a valid aro-object. [5]

Aros are most likely to be users, but keep in mind that you can use any arbitrary object by overwriting RestrictedActiveRecord::$inAttendance.

Let's take a look at what Aros can do:

```php
<?php

        /**
     * Looks up if the user is granted a specific action to the given object
     * @param   string|array    $obj    The object to be checked
     * @param   string          $action the action to be performed
     * @return bool true if access is granted, false otherwise
     */
    public function may($obj, $action);

    /**
     * Grants the object denoted by the $obj-identifier the given actions
     * @param type $obj the object identifier
     * @param array $actions        the actions to grant
     * @param bool  $byPassCheck    Whether to bypass the additional grant-check
     * @return bool
     */
    public function grant($obj, $actions, $byPassCheck = false);

    /**
     * Denies the object denoted by the $obj-identifier the given actions
     * @param type $obj the object identifier
     * @param array $actions the actions to deny
     * @return bool
     */
    public function deny($obj, $actions);

?>
```

Examples:

```php
<?php

        //Note that you need the CActiveRecord, not the CWebUser!
        $user = User::model()->findByPk(Yii::app()->user->id);
        $anotherUser = User::model()->find(...);

        //With the default settings, the User has all permissions on this picture
        //To change that, edit 'autoPermissions' in config.php
```

[4] You can change the default model in RestrictedActiveRecord:: $model
[5] You can change the default guest Group in config.php "guestGroup"

```php
        $picture         = new Picture();
        $picture->save();

        //As of now, the other user can see that picture
        $anotherUser->grant($picture, 'view');

        $anotherUser->may($picture, 'view'); // => true
        $anotherUser->may($picture, 'delete'); // => false

        //Simulate an attempt
        RestrictedActiveRecord::$inAttendance = $anotherUser;
        $picture->delete();        // => RuntimeException

        //Get back (the session will be used if it's NULL)
        RestrictedActiveRecord::$inAttendance = NULL;

?>
```

Or another one, leveraging the fact that both the acos and the aros are all acl-objects and thus can use inheritance:

```php
<?php

        $group = new RGroup(); //RGroup => RequestGroup
        $group->alias = 'myGroup';

        //this works due to internal assurance
        //Equals $anotherUser->join($group);
        $anotherUser->join('myGroup');

        $group->grant($picture, 'edit');

        $group->may($picture, 'edit');                  // => true
        $anotherUser->may($picture, 'edit'); //=> true

?>
```

Or the other way round:

```php
<?php

        $group = new CGroup(); //ControlGroup
        $group->alias = 'picGroup';

        $picture->join('picGroup');
        $picture->is('picGroup'); // => true

        $anotherUser->grant('picGroup', 'delete');
        $anotherUser->may($picture, 'delete'); // => true

        $picture->leave('picGroup');
        $anotherUser->may($picture, 'delete'); // => false

?>
```

3.2 Permission Management

3.2.1 Regular Permissions

Permissions are CActiveRecord-Objects representing the *link* between Aros, Acos and Actions.
When an action is granted, a permission is created, if an action is denied, the appropriate permission is removed from the database. If rows are selected from the database, a premission granting access for each individual row is sought (don't be aghast - it's done in one query).
Permissions are internally used and should never be created, modified or deleted directly in the database. The only interesting thing you can use is the timestamp: Each permission stores the timetamp when it was created in "created", so you can use that whenever you select acos from the database (you can find an example on page **??**).

There are many funny things you can do with permissions. Just a smattering:

- **Very general permissions**
  With this type you can permit everybody to perform specific actions. By default, this is only the create action (see config.php):

```
<?php
        /**
     * Permissions which are allowed for all the users on all objects
     * default: only create
     */
            'generalPermissions' => array('create'),
?>
```

  All aros will have the permission to perform the actions you list there on any indiscriminate aco. So be careful what you write there.

- **Auto assigned permissions**
  If a user (or in general terms: aro) created an object, you may want to grant him some special permissions nobody else has in the first place. In real-world applications it is very likely that an owner is given certain privileges - for example the admins and he himself can delete his own objects.

```
<?php

        /**
     * Defines which permissions are automatically assigned to the creator of                an o
     * default: all
     * You can overwrite this using the autoPermissions-value of each object
     */
            'autoPermissions' => '*',
?>
```

  Note that this method will override any restrictions you have imposed on grant/deny operations.

- **General Permissions**
  Sometimes, you're weary with assigning permissions on each new aco to an aro. You do that over and over again, ad nauseam. You want a general rule, something like: "that User has permission X on each object of type Y" (Real- world example: "Every Admin (possibly a group with several users) can publish and delete comments"). Then, general permission are with you.
  [6] General Permissions don't refer to regular, single acos as objects but to special acos. Those acos do not

[6] You might, at first, think about access to pending comments. Well, there's a better method because pending is only a subset of all comments. What about simply adding the "Pending"-group to the $autoJoinGroups property of the model and granting the admins access on this group?

represent single entities but "types" of entities - most likely models. whenever you grant some aro a general permission, he is permitted to perform this permission on each object of that type. With general permissions, it's as easy as that:

```php
<?php
        //Admin record
        $admin = User::model()->find();

        //Assuming that "Pending" is an aco-group
        //£admin->grant('Model', 'action');

        $admin->grant('Picture', 'delete');
?>
```

From now on, this admin (a user or an entire group) will have delete-access on all acos of type 'Picture'. Please note that those special acos can be treated just as regular acos - you can use inheritance in particular.

- **Restriction of Permission-Changes**
  You can control which aros can grant/deny on which acos. To enable this control, set

```php
<?php
        'enablePermissionChangeRestriction' => true;
?>
```

in config.php. From now on, nobody can grant or deny anything on objects unless you have permitted him to do so before.

```php
<?php
        $me = User::model()->findByPk(Yii::app()->user->id);
        //Assuming none of both users has such permissions yet
        $user1 = User::model()->find(...);
        $user2 = User::model()->find(...);
        $picture = Picture::model()->find(..);

        //Simulate being £user1
        RestrictedActiveRecord::$inAttendance = $user1;

        //Try
        $user2->may($picture, 'delete');           // => false
        $user2->grant($picture, 'delete'); // => RuntimeException

        //Give him the permission
        RestrictedActiveRecord::$inAttendance = $me;
        $user1->grant($user2, 'grant');

        //Simulate again
        RestrictedActiveRecord::$inAttendance = $user1;
        $user2->grant($picture, 'delete'); // => true

?>
```

Note that you can combine, of course, combine generalPermissions with this restriction.

- **Specific PermissionChange Restriction**

  Sometimes it's not sufficient to restrict grant-permissions in general, but you have to restrict *what* an aro can grant another aro on an aco.

Enable this feature this way:

```php
<?php
    /**
     * Enables you to restrict WHAT an Aro can grant/deny, whether it can grant something at all
     */
            'enableSpecificPermissionChangeRestriction' => true,
?>
```

Actions are mapped to permissions like this: 'myAction' =>'grant_myAction' / 'deny_myAction'. Note that Actions are **not automatically created even if *strictMode* is disabled**. Hence you must create them in advance in your database. In these premises it's really a good idea to limit the actions on a model using it's $possibleActions property, otherwise you'll probably run into problems if you use the wildcard '*' and haven't definede all appropriate grant/deny-actions for the regular actions before.
The action to grant an aro to grant the permission to grant actionX to another aro is consequently 'grant_actionXgrant_actionX' (Got it? ;D)

Note that this detailed grant-mangement does not come to bear if you grant someone the general "grant" -permission.
Quiz: What is the specific action to grant an aro to grant "*" to another aro? [7]

```php
<?php
        var $me;
        //Assuming the user has no rights with respect to the above
        //They've been assigned somewhere above
        var $user1;

        var $att = &RestrictedActiveRecord::$inAttendance;

        //Simulate
        $att = $user1;

        //Try
        $user2->grant($picture, 'delete'); // => RuntimeException

        //Grant
        $att = $me;
        $user1->grant('User', 'grant_delete');

        //Retry
        $att = $user1;
        $user2->grant($picture, 'delete'); // => true


?>
```

3.2.2 Relation Change Restriction

Your control is not confined to grant/deny-operations but you can also limit who can change the relations between acl objects (joining and leaving) - and in which way. This is called *Relation Change Restriction*. First of all you must enable it, like any other special feature:

[7] It is "'grant_grantgrant_grant"'. Frankly, it's a bit ugly. ;)
If you got that right off the cuff, you are quite quick on the draw - congratulations.

```php
<?php
/**
     * Enables the check of join/deny operations based on permissions
     * the actions are "join" and "leave"
     */
    'enableRelationChangeRestriction' => true,
?>
```

From now on, you have to grant somebody the permission to join a specific group before he can do that:

```php
<?php

        /**
         * The £me-object must have the grant_join permission
         */
        var $me, $user, $group;

        //Change Aro
        $place = &RestrictedActiveRecord::$inAttendance;
        $place = $user;

        $user->may($group, 'join'); // => false
        $user->join($group); //=> RuntimeException

        //Change back
        //Assume you have the permission to grant this
        $place = $me;
        $user->grant($group, 'join');

        //Return
        $place = $user;
        $user->join($group); // => true
?>
```

[8]

If you want to allow somebody to join/leave all groups, you can grant him those permissions on 'All' (assuming that you have used the autoJoinGroups feature).
If you want that he can only join a given type of groups, you can employ the generalPermissions.

3.3 Business Rules

This extension *does* support the so-called *Business Rules*. Business-Rules are, in brief, PHP methods determining if a given acl-object is a child of another acl-object (for illustration: if it is a member of a specific group). Yii uses PHP-expressions to determine that - *really* a bad habit. Business-Rules are methods of the class *BusinessRules* in the /strategies subfolder of the form "isGroup" where "Group" is the alias of the group the given acl-object is checked for being a child of. [9]

Please note that Business-Rules are applied for *all* operations just as all the other features are, except for the search/find-operations. I.e. a $aro- >may() *does* apply the business rules as well as $aro->is() does. But whenever you do a find()-operation on a restricted model, they are *not* applied

[8] Astute readers will have noticed that I used $group, which is obviously an aro, as an aco and the script didn't complain. well. This is quite a weird feature: the acl extension automatically converts between both types if necessary. That means you can use any aro as an aco and vice versa. Please note that you need to add the behaviors to your models to use those new operations on a model if you use a model. However, the functionality doesn't depend on it.

[9] You can easily overwrite the default naming convention by using the $ruleMap property of the class. The key is the regular rule name and the value the name you want it to be.

by default. Nonetheless you can apply them yourself, using just the same cumbersome method you'd have to use with RBAC anyway:

```
<?php

        $results = myRestrictedModel()model()->findAll(..);
        foreach($results as $key=>$obj){
                if(!$obj->grants('read'))
                        unset($results[$key]);
        }
?>
```

Please note that this *is as slow as you expect it to be*. For n queries, it will send in turn n * a queries, where a depends in your configuration and is in any case greater or equal to 3.
[10]

You can activate Business Rules as easy as follows:

```
<?php
        /**
     * Enables the business rules for all actions (automatical lookup) _except_ the
     */
    'enableBusinessRules' => true,
?>
```

If you want to use Busines Rules, you have to do two things:

1. Get an acl object which has an alias, or set it if it doen't already have one.
2. Create a business rule in the class returning true/false

Let's take a look at the interface of business rules:

```
<?php

        /**
     * prototype:
     * public static function isRule(arr('child' => .., 'father' => ..),
     *                                arr('child' => .., 'father' => ..),
     *                                string)
     * public static function isRule(&aro, &aco, &action)
     */
?>
```

The child objects are always the objects you have passed to the performing method. E.g. if you do $aro->may($aco, 'action');, $aro is the aro child and aco is the aco child. The father is always the group to determine whether the given objects (each child) are children of. You can take into account the $action, which is always given as a string.

There is a special $action "is", whose sole purpose is to determine whether the given child is a children of the given father. In this case, the parameter of the other, not requested, type is NULL.

Note that the passed objects are always the *highest* objects, that means *if* there is an associated real world entity for an object, *it is* passed instead of the bare acl object.

Let's give an example: We assume that there is an aro group called (this expression will in the ensuing text refer to "aliased") "Author". This aro group has all permissions on an aco group called "Picture" (this is a general permission, of course. It will be created automatically). Now, we have the check:

[10] Note that this is not a shortcoming of this extension but it's rather simply not possible to alter already fetched resultsets with Yii builtin features.

```php
<?php
        //Assuming $picture is a random CActiveRecord of type "Picture"
        //Assuming you have activated the general Permissions
        $picture->save();
?>
```

Well, there are two actions that you might need to make this run without an exception being thrown:

1. "update": If the object already exists
2. "save" : If the object is to be created

Let's take a look at the Business Rule: Suppose we want to grant all privileges if the attribute *author_id* equals *id* of the user object.

```php
<?php
class BusinessRules{
        /**
         * $aro['father'] will be the acl object "Author"
         * $aco['father'] will be teh acl object "Picture"
         */
        public function isAuthor($aro, $aco, $action){
                return $aco['child']->author_id == $aro['child']->id;
        }

};

?>
```

I think that's quite easy to get. And I just slipped by a feature that can be hard to lash on, if you name it explicitely: Business Rules on both sides *simultaneously*. What does that mean?
Business Rules, in general, check whether a given object is a children of a given group. Now, as both aros and acos are objects, we can easily do this on both aros and acos. And we can do it simultaneously: In the case above, neither is the user a member of "Author", nor is the picture explicitely a member of "Picture".
If you check for a permission, the regular permission check is performed in the first place. If that doesn't succeed and you have enabled Business Rules, an extended permission check is conducted: Permissions which become valid if the given aro or the given aco (or both of them) miraculously becomes member of a group just in time. This isn't *that* hard because this can only happen if the alias of that group corresponds to one of the defined Business Rules. For details about how those checks are done, please resort to the "Permission Checking Algorithm".

Special configuration setting

```php
<?php
        /**
     * Sets the direction in which business-rules are applied. Default is to check
     * both sides
     * possible values: "all", "both", "aro" and "aco"
     *
     * The difference between "all" and "both" is that all evaluates _all_ rules
     * whereas both only accepts rules where only one (of aro and aco) are
     * determined using Business-Rules
     */
    'lookupBusinessRules' => 'all',
?>
```

This special configuration option allows you to specify, whether the check algorithm should:

- "aco": allow Business Rules for aco groups

- "aro": allow Business Rules for aro groups
- "both": allow Business Rules for aros and acos, but not simultaneously. [11]
- "all": allow Business Rules in all its power

This setting does influence the performance, however it doesn't decrease it that much. It should mainly be changed if you are going to use Business Rules only on one side anyway but there in excess.

4 Special Features

4.1 Permission Query Modification

Permission Query Modification

Normally, the permission system works somewhere in the background for database queries, if you don't want to make it's acquaintance, you are not going to make it. However, in certain situations you might want to *modify* the behavior somehow. This can be done using the *Permission Query Modification* - you can set this for each query individually (actually, you have to ;-)). Note that the usage of this criteria is dedicated for permission-related modifications. For any other (regular) filtering, you can use the regular facilities of Yii.
To make modifications, you can modify the criteria in *RestrictedActiveRecordBehavior::$criteria['relation']*. It's the simply array-form of the criteria you also use for your scopes in models - it is merged with the regular CDbCriteria. With the relation key you are also capable of changing the behavior for some extraneous related models of your mainly selected models (the relation is simply the one you defined in your relations definition in your model, if you want to refer to the model itself, simply use 't').
The additional tables are:

- **"map"**: This refers to the permission-table itself. While you could access all of its properties, the most important one is the "created" attribute, storing the timestamp of its creation.
- **"acoC"**: This refers to the collection-table for the given acl object
- **"aco"**: This refers to the node table for the given acl object

Don't get in a frenzy if you look at the queries and can't find your condition in a snap. In fact, it's depicted in here easier than it actually is.

Here is an example:

```
<?php
        /**
         * This will order the results of whatever selection you perform later on
         * by the time you were granted to see them. This can be useful if you
         * develop another facebook and want to order the items by the date they
         * have been shared to a user (they may be uploaded a long time ago before
         * they are actually shared)
         */
        RestrictedActiveRecordBehavior::$criteria['t']['order'] = 'map.created DESC';
?>
```

4.2 Group Conditions

Group Conditions

Group Conditions let you filter real world entities when you retrieve them from the database. Sometimes, you may want to show only a subset of records having an attribute in common. For example you may want to show a

[11] That means: If a permission requires that both the involved aro and aco group grant the involved aco and aro inheritance just in time using Business Rules, the permission will not be accepted. Only one relation can be established just in time.

user all images being related to "woman" but not the ones who are definitely "explicit content".

Let us take a look at the definition:

```
<?php
class RestrictedActiveRecordBehavior{
        /**
      * This method will add a criteria to the next query, which limits the resultset
      * to a subset which fulfills the given group-conditions.
      * The relation is the name of the relation you use in your relations definition
      * in the model, if it's the accessed model itself, the relation is 't'.
      *
      * Structure of the Restriction:
      * restriction := array('operator', [recursiveRestriction [,recursiveRestriction...]]);
      * recursiveRestriction := aclIdentifier
      *    | array('operator', [recursiveRestriction [,recursiveRestriction ...]])
      * aclIdentifier    = 'alias'
      *    | array('alias' => 'myAlias')
      *    | array('model' => 'myModel', 'foreign_key' => 'foreignKey')
      *    | any Instance of CActiveRecord
      * operator := 'and'
      *    |  'not'
      *    |  'or
      *
      * This is quite a theoretical definition. Let's get down to the code:
      * array('and', 'Group1', array('not', 'Group2'));
      *  => This will yield all objects being in group1 but not group 2
      *
      * @param string $relation  Name of the invoked relation
      * @param array  $restr     the group-restrictions
      */
     public static function addGroupRestriction($relation, $restr);
};
?>
```

Because it's quite cumbersome to write the fully qualified name, there's a shortcut: *Util::addGroupRestr*. As aforementioned, the nesting can be done infinitely, though you should keep in mind that each condition decreases the performance slightly (a subquery is necessary for each condition). This shouldn't be an issue for small-scale applications. Please note that the filtering is only applied once, to the imminent query - not to any ensuing ones.

4.3 Access Control Filter

Access Control Filter

You can extend the existing Access Control Filter to allow only users of certain groups, and in this case acl groups, the access. The syntax doesn't change, it's very convenient.

4.3.1 Installation

(It's not possible to introduce a better procedure, as the core development team has rejected to make things protected.)

First, you have to update the framework itself. Go to *framework/web/auth/CAccessControlFilter.php* and change line 67 to:

```
<?php

        protected $_rules=array();
?>
```

In order to use the new Filter, overwrite the method *"filterAccessControl"* in your controllers (preferrable: a base controller):

```
<?php

public function filterAccessControl($filterChain) {
            $filter = new ExtAccessControlFilter;
            $filter->setRules($this->accessRules());
            $filter->filter($filterChain);
        }
?>
```

You are done. There are no changes in syntax or behavior to keep in mind. Roles correspond to User-Groups (and thus Aro-Objects). Everything works just as it did before.

4.4 Ajax Permission Management (unstable)

(marked as unstable)
The extension ships with its own ajax permission management dialog. This dialog requires you to use *JQuery* and *jQuery UI Dialog* as optionally *Bootstrap* (but only for one button) (all of which you have to include yourself). The permission manager allows users to add, edit and delete permissions on a set of given acos. the permission manager is entirely generic, so you can use him with any aros and acos you like, with any permissions you can think of.

4.4.1 Features

- Supports individual actions
- Supports an arbitrary large set of objects, which can have different types
- Provides ajax autocompletion for the objects
- Lets the user change the permissions of several aros in a row (bulk-operations)
- As it resorts to the regular acl implementation, it also relies on its full security

4.4.2 Limitations

The manager does not take Business Rules into account. That means even if someone *has* a certain permission, this person will *not* show up in the permission dialog.

4.4.3 Configuration

You should configure the actions in the first place. The actions can be defined in *components/management/ActionMeta.php*. You can set the actions also in your individual models to specialize them - "Delete an object" will serve in a general manner, but "Remove this image" simply sounds more accurate. The structure to use is always the same as in the main file:
(Please note that the extension does *not* ship with icons for the default actions. Please set the path according to your individual project.)

```
<?php
        public static function getActionMeta(){
        $am = Yii::app()->getAssetManager();
        $path = $am->publish('images/crystal_project/16x16/actions');

       return
          array(
```

```
            'create' => array(
                'name' => Yii::t('acl', 'Create Object'),
                'enabled_img' => $path.'/save_all.png',
                'disabled_img' => $path.'/save_all_disabled.png',
                /*
                'short_des' => Yii::t('acl', 'Creates an object of type X'),
                'long_desc' => Yii::t('acl', 'Creates an object of type X'
                .' If you do this, x, y and z will happen and you have b choices')
                */
             ));
    }
?>
```

Also take a look at *ActionMeta::listActions* and *ActionMeta::allowedActions*: the former represents the actions the user is going to see while the latter are the ones you can access using ajax (you probably want to use the interface yourself - independent of the manager himself).
Note that while the action definitions themselves are merged between models and the general definiton (where model takes precedence in case that two definitions overlap), the listActions and allowedActions from the model will entirely overwrite the general settings, if set. To use the permission manager for a specific model, you have to activate that firstly for the specific model:

```
<?php
class myModel{

        public static $enableAjaxPermissionManagement = true;
}
?>
```

There are two additional configuration options you may want to change:

```
<?php

class ManagementController{
        /**
     * Defines the model name used for bare acl collections
     * (that are virtual entities which do only exist in the acl system)
     * @var string
     */
    public $virtualModel = 'Collection';

    /*
     * Defines which models should show up always in the dialog
     * If the dialog is called on some acos, each type of aro gets a new tab.
     * However, if no aro of a certain type has got any permission on all
     * objects in the set, the type will not show up.
     * The types you specify here will always show up - independent from the
     * existence of any aros
     * @var array
     */
    public $aroForceList = array('User');

};
?>
```

4.4.4 Usage

Whenever you want to use the Permission Manager, just perform a javascript call like this:

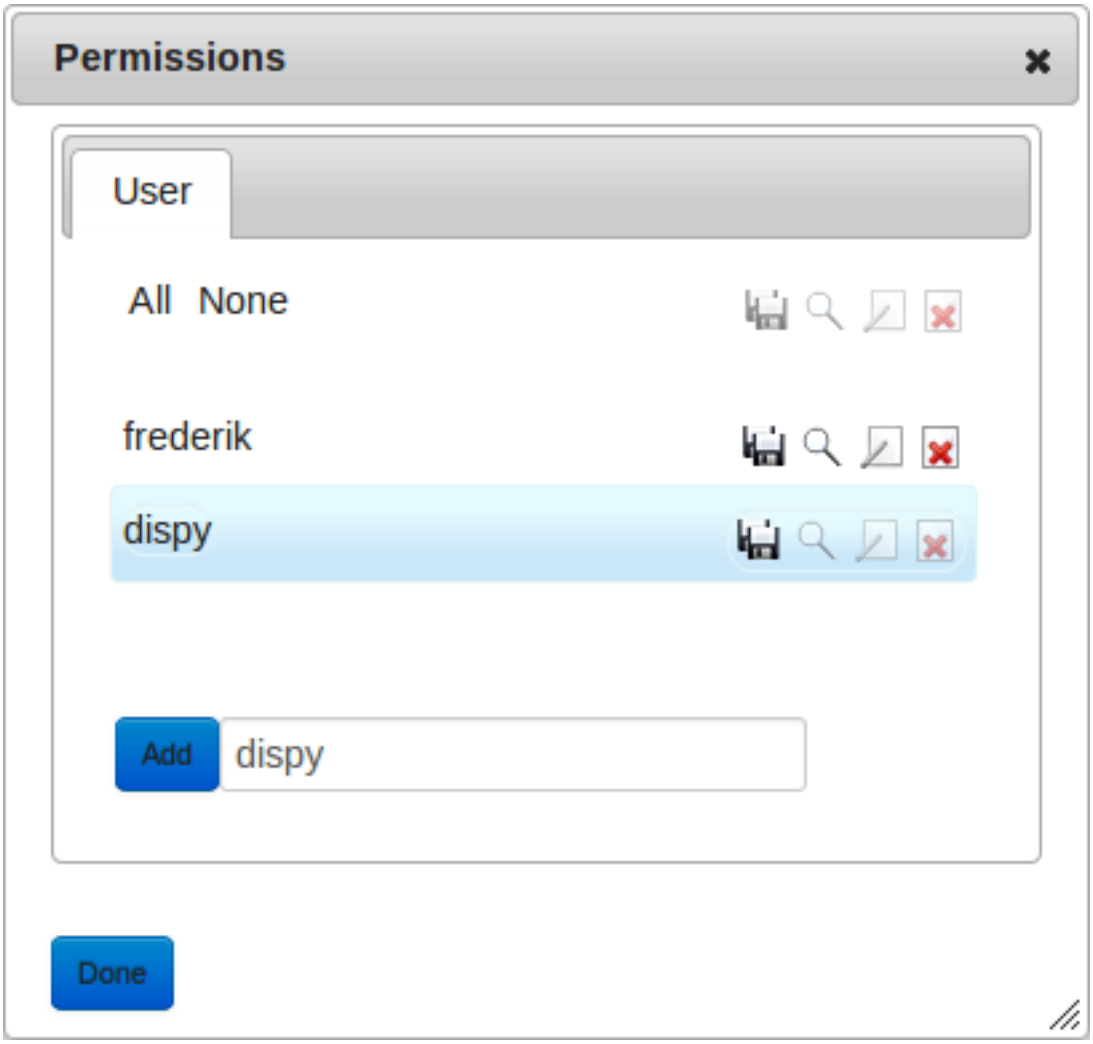
```
var objects = new Array();
objects.push({
        'model' : 'MyModel',
        'id'                : myId
});
showPermissionDialog(objects);
```

Note that you can different objects of *different models* in the same set. The actions available then are the intersection set of the actions possible in the distinct models.

The attributes used for the autoCompletion are these ones, tried in the order they are shown. The first match will be used as the descriptive value:

```
<?php
        array('title', 'name', 'shortcut', 'alias', 'id')
?>
```

Please note that the Permission Manager is currently not in a stable version and should not yet be used in a production environment. However, feel free to test and report issues. The manager itself looks like this:



4.5 Caching

The extension has builtin support for query based caching. You can specify in detail what should be cached and how long.

Note that it's safe to set high values for the lifetime of cached values as the cache is flushed whenever write access is performed. Therefore it's usually a good idea to use a dedicated cache medium for acl it will flush much more often than regular caches will do.

```
<?php
        'caching' => array(
      /**
       *  If enabled, the given number defines the number of seconds after which
       * the cached values are invalidated. A value of 0 indicates that caching is disabled.
       */

      /**
       * The acl objects themselves are cached
       */
      'collection' => 0,

      /**
        * The actions are cached
        */
      'action'     => 0,

       /**
        * The permissions are cached
        * Note that this does only apply for all checks where may() is involved. -
        * This is the case whenever you call it, or for the general permissions
        * Note: the regular find() method is _not_ affected unless you use
        * general permissions
        */
      'permission' => 0,

      /**
       * This caches the objects associated with the acl objects (above: the collections)
       * This cache affects the REAL objects whenever this extension accesses them
       */
      'aclObject' => 0,

      /**
       * Caches _only_ the objects used as the _current_ aro - not aros in general
       * if you do not enable this, the aro will be requested every single time from your database
       * whenever you do something with acl
       */
      'aroObject' => 0,

      /**
       * caches the structures of objects, e.g.:
       * relations (parent, child) and paths
       */
      'structureCache' => 0,

       /**
        * Allows you to use another cache component than the default one, solely for acl
        */
      'cacheComponent' => 'cache'
    ),
?>
```

5 Installation

1. Copy the files of the module in your module folder
2. Include the search paths into your configuration:

```
<?php
        'import'=>array(
        'application.modules.acl.components.*',
        'application.modules.acl.models.*'
)
?>
```

3. Rename "config.default.php" to "config.php" in "components/strategies/nestedSet/pathMaterialization"
4. Create the tables using the install/install.sql file
5. Add the behaviors (RestrictedActiveRecordBehavior/RequestingActiveRecordBehavior) to your models
6. Add this code to your models having *RestrictedActiveRecord*:

```
<?php
        public function defaultScope(){
            $criteria=  $this->access->generateAccessCheck();
           return $criteria->toArray();
        }
?>
```

Note that it's no problem to modify the criteria as you wish - you can simply merge it.
Also note that you *must* provide a valid aro. The easiest way is to log in as a user (your user object will be used implicitely) or to fill *RestrictedActiveRecord::$inAttendance* with a CActiveRecord or an acl object of your choice.

6 Caveats

Things you should never do...

You should never create an acl object, operate on it and associate a real-world entity to it later on.

As mentioned earlier in this book, the extension is capable of treating every aro as an aco and vice versa. If you do that, the objects have to be transformed. The first question that loomes up is that: What identifier shall I use to denote the new object? Easy answer: the one given to you! That's a fallacy. You can use the entity and the object interchangeable on all methods.
If the received object is a bare acl object without an associated entity (what we assumed in the beginning), the identifier created has the model "CGroup"/"RGroup" (depending on which object type you used) and the ID the one of the acl object.
If you now happen to add an entity to the bare acl object you used to create this new object, and you again use it to transform itself into the other type, there are two objects in the database essentially referring to the same original object (because real world entities are always transformed using their real world model and id).
You *could* use the lowest level's object (the acl object) for the transformation and you wouldn't have this hassle. On the other hand, you always get the *highest* possible object back from the interfaces, because in most cases that's what you want to deal with.
So the decision is quite straightforward in this case; convenience for the most cases takes precedence over this rare scenario.

You should never use an alias which equals a model name if you enabled generalPermissions.

The General Permissions use the model name as the alias. And they are adamant in what they do :)

7 Internals

7.0.1 Permission Check Algorithm

The following will give you a sketch of how the permission check works and in which order the different layers are processed.

1. The aco object is loaded from the database
2. The action is loaded from the database
   =>a *RuntimeException* is thrown if it doesn't exist
3. *if enabled*
   If present, the action is checked against the possibleActions of the model
   =>a *AccessViolation* is thrown if it's not possible to perform that action on that model
4. It is checked whether the *very* generalPermissions grant the actions
5. *if enabled*
   This check (up from the top) is done with the type specific generalPermissions. If that fails, the check continues.
6. The regular permission check is performed. If the check fails and the Business Rules are disabled, false is returned
7. *if enabled*
   The Business Rules are enabled. If any of the fetched permissions is accepted by the Business Rules, true is returned, otherwise false.

8 Changelog

0.5.1

- added
  - database migrations
  - caching
- improved
  - performance

0.5:

- added
  - created-column for Permissions
  - RestrictedTreeActiveRecoreBehavior
  - support for join/leave restrictions
  - autoJoins for created objects (both aro and aco)
  - automatic Conversion between aros and acos
- fixed
  - assureSafety() issue
  - RestrictedActiveRecordBehavior
  - General Permissions: the model-independent General Permissions are not affected by the "enableGeneralPermission" flag the config anymore
  - issue with Business Rules lookup: now you can use Business Rules simultaneously on both directions. Plus, it's faster :-¿

  - Business Rules: is() did not take them into account
  - if the permission was given as a string, RestrictedActiveRecord::mayGenerally returned directly with any result if generalPermission was activated, not checking any further

- improved
  - Changed parameters of Business Rules
  - for granting/denying something on an aco, the object being affected is no longer considered the involved aro, but the object returned by RestrictedActiveRecord::getUser();
  - support for General Permissions also for read-operations (e.g. calls to find())
  - placeholders for criteria (aco, acoC, map) for 'condition', 'order', 'join' and 'select'
  - ExtAccessControlFilter now restors to RestrictedActiveRecord::getUser() instead of the session
- deprecated
  - the use of RestrictedActiveRecord, RestrictedTreeActiveRecord and RequestingActiveRecord as base classes is no longer supported

0.42:

- added extended AutoSave Facility
- improved usage of CActiveRecord in RestrictedActiveRecord::$inAttendance
- fixed autoPermissions infinite loop
- fixed missing methods in the behavior adapter for the interfaces

0.41:

- improved bugfix for General Permissions

0.4:

- added Business Rules
- added General Permissions
- added fine grained Permission Restriction

0.3:

- improved fully recoded
- improved offers behaviors now
- added switchable strategies (shipped: Path Materialization)
- added abstracted actions
- added limitation of actions and new syntax for action-definitions
- added several other convenient interfaces

0.2:

- fixed if strict mode was disabled, automatic creation of aro collections did not work properly in all cases
- improved some minor fixes in RestrictedActiveRecord
- added access control filter

0.1:

- fixed proper removal of associated acl objects upon deletion of the ActiveRecord
- fixed errors in conjunction with DataProviders