

---

## acl9

build unknown

Acl9 is a role-based authorization system that provides a concise DSL for securing your Rails application.

Access control is pointless if you're not sure you've done it right. The fundamental goal of acl9 is to ensure that your rules are easy to understand and easy to test - in other words acl9 makes it easy to ensure you've got your permissions correct.

### Installation

Acl9 is Semantically Versioned, so just add this to your [Gemfile](#) (note that you need 3.2 for Rails 6+ support):

```
1 gem 'acl9', '~> 3.2'
```

You will need Ruby > 2.0

### Rails 4 - stick with 2.x

```
1 gem 'acl9', '~> 2.0'
```

### Rails < 4 - upgrade Rails!

We dropped support for Rails < 4 in the 1.x releases, so if you're still using Rails 2.x or 3.x then you'll want this:

```
1 gem 'acl9', '~> 0.12'
```

### Getting Started

The simplest way to demonstrate this is with some examples.

### Access Control

You declare the access control directly in your controller, so it's visible and obvious for any developer looking at the controller:

---

```
1 class Admin::SchoolsController < ApplicationController
2   access_control do
3     allow :support, :of => School
4     allow :admins, :managers, :teachers, :of => :school
5     deny :teachers, :only => :destroy
6
7     action :index do
8       allow anonymous, logged_in
9     end
10
11     allow logged_in, :only => :show
12     deny :students
13   end
14
15   def index
16     # ...
17   end
18
19   # ...
20 end
```

You can see more about all this stuff in the wiki under Access Control Subsystem

## Roles

The other side of acl9 is where you give and remove roles to and from a user. As you're looking through these examples refer back to the Access Control example and you should be able to see which access control rule each role corresponds to.

Let's say we want to create an admin of a given school, not a global admin, just the admin for a particular school:

```
1 user.has_role! :admin, school
2 user.has_role! :admin, of: school
```

Then let's say we have some support people in our organization who are dedicated to supporting all the schools. We could do two things, either we could come up with a new role name like `:school_support` or we can use the fact that we can assign roles to any object, including a class, and do this:

```
1 user.has_role! :support, School
2 user.has_role! :support, for: School
```

You can see the `allow` line in our `access_control` block that this corresponds with. If we had used `:school_support` instead then that line would have to be: `allow :school_support`

---

Now, when a support person leaves that team, we need to remove that role:

```
1 user.has_no_role! :support, School
2 user.has_no_role! :support, at: School
```

You can see more about all this stuff in the wiki under Role Subsystem

## Database Setup

As mentioned in Role Subsystem you don't have to use these, if your role system is very simple all you need is a `has_role?` method in your subject model that returns a boolean and the Access Control part of Acl9 will work from that.

However, most commonly, the roles and role assignments are stored in two new tables that you create specifically for Acl9. There's a rails generator for creating the migrations, role model and updating the subject model and optionally any number of object models.

You can view the USAGE for this generator by running the following in your app directory:

```
1 bin/rails g acl9:setup -h
```

## Configuration

There are five configurable settings. These all have sensible defaults which can be easily overridden in `config/initializers/acl9.rb`

You can also override each of the `:default_*` settings (dropping the "default\_" prefix) in your models/controllers - see below for more detail:

### **:default\_role\_class\_name**

Set to `'Role'` and can be overridden in your "user" model, see the wiki for more.

### **:default\_association\_name**

Set to `:role_objects` and can be overridden in your "user" model, see the wiki for more. We chose a name for this association that was unlikely to conflict with existing models but a lot of people override this to be just `:roles`

---

### **:default\_subject\_class\_name**

Set to `'User'` and can be overridden in your “role” model, see the wiki for more.

### **:default\_subject\_method**

Set to `:current_user` and can be overridden in your controllers, see the wiki for more.

### **:default\_join\_table\_name**

This is set to `nil` by default, which will mean it will use the Rails method of calculating the join table name for a `has_and_belongs_to_many` (eg. `users_roles`). Remember that if you override this value, either do it before you run `rails g acl9:setup` or be sure to update your migration or database.

### **:normalize\_role\_names**

Set to `true` (see “Upgrade Notes” below if you’re upgrading) and can only be changed by setting it in `Acl9.config`. When true this causes Acl9 to normalize your role names, normalization is `.to_s.underscore.singularize`. This is done on both the setter and getter.

### **:protect\_global\_roles**

Set to `true` (see “Upgrade Notes” below if you’re upgrading) and can only be changed by merging into `Acl9.config`. This setting changes how global roles (ie. roles with no object) are treated.

Say we set a role like so:

```
1 user.has_role! :admin, school
```

When `:protect_global_roles` is `true` (as is the default) then `user.has_role? :admin` is `false`. Ie. changing the role on a specific instance doesn’t impact the global role (hence the name).

When `:protect_global_roles` is `false` then `user.has_role? :admin` is `true`. Ie. setting a role on a specific instance makes that person a global one of those roles.

Basically these are just two different ways of working with roles, if you’re protecting your global roles then you can use them as sort of a superuser version of a given role. So you can have an admin of a school **and** a global admin with different privileges.

---

If you don't protect your global roles then you can use them as a catch-all for any specific roles, so then the admins of schools, classrooms and students can all be granted a privilege by allowing the global `:admin` role.

### Example

```
1 # config/initializers/acl9.rb
2 Acl9.config.default_association_name = :roles
3
4 # or...
5 Acl9.configure do |c|
6   c.default_association_name = :roles
7 end
```

### Reset Defaults

On the off chance that you ever need to reset the config back to its default you can use:

```
1 Acl9.config.reset!
```

### Upgrade Notes

#### Acl9 now protects global roles by default

Please, PLEASE, **PLEASE** note. If you're upgrading from the `0.x` series of `acl9` then there's an important change in one of the defaults for `1.x`. We flipped the default value of `:protect_global_roles` from **false** to **true**.

Say you had a role on an object:

```
1 user.has_role! :manager, department
```

We all know that this means:

```
1 user.has_role? :manager, department # => true
2 user.has_role? :manager, in: department # => true
```

With `:protect_global_roles` set to **false**, as it was in `0.x` then the above role would mean that the global `:manager` role would also be **true**.

ie. this is how `0.x` behaved:

```
1 user.has_role? :manager # => true
```

---

Now in 1.x we default `:protect_global_roles` to **true** which means that the global `:manager` role is protected, ie:

```
1 user.has_role? :manager      # => false
```

In words, in 1.x just because you're the `:manager` of a `department` that doesn't make you a global `:manager` (anymore).

### **Acl9 now normalizes role names by default**

So basically we downcase, underscore, and singularize your role names, so:

```
1 user.has_role! 'FooBars'
2
3 user.has_role? 'FooBars'      # => true
4 user.has_role? :foo_bar      # => true
5
6 user.has_role! :foo_bar      # => nil, because it was already set above
```

If you're upgrading then you will want to do something like this:

```
1 Role.all.each do |role|
2   role.update! name: role.name.underscore.singularize
3 end
```

**Then check for any duplicates** and resolve those manually.

### **Acl9 now raises ArgumentError on bad args to allow/deny**

In 2.x and above we now try to help the developer by raising `ArgumentError` if they mess up with the options they pass to `allow/deny`, this prevents people doing things that they think are going to work but actually aren't like:

```
1 allow all, actions: [ :index, :show ]      # <---- BROKEN!!
```

## **Community**

**Gitter:** Join the gitter chat here

**docs:** Rdocs are available here.

**StackOverflow:** Go ask (or answer) a question on StackOverflow

**Mailing list:** We have an old skule mailing list as well `acl9-discuss` group

---

**Contributing:** Last but not least, check out the Contributing Guide if you want to get even more involved

## **Acknowledgements**

All these people are awesome! as are all the people who have raised or investigated issues.