The ActiveRBAC Manual

Manuel Holtgrewe

April 21, 2006

For ActiveRBAC 0.3.1

Please note that this manual is work in progress. We try to keep it up to date but we cannot guarantee on correctness in all cases.

Contents

1 Foreword

Ruby on Rails (RoR) has whirled some things up in the web development community: It strikes a balance between easy but unstructured programming as many people do with PHP and structured but hard programming as Java is considered to be. It propagated the Convention over Configuration (CoC) along with its parent Don't Repeat Yourself (DRY) paradigm. And it shipped with the (I would sometimes call it excentric) opinion of its creator David Heinemeier Hansson or DHH as he is called by the community.

While all people using RoR came to love the DRY and CoC, some disagree with some of David's opinions. I include me in this. For example, according to David it is easier to create components anew from scratch than to configure existing components to fit your requirements. His idea of components includes things like ActiveRBAC.

This effectively means that the creator of Ruby on Rails is against a component (or engine) like ActiveRBAC and will never use it. However, there are still those people who consider software reuse A Good Thing. I include me here, too. There *are dangers*, of course. The more high level your components become, the harder it is to intergrate them into your software: Flexibility, amount of problems solved (power), ease of usage - pick two. Configuration easily becomes hard to unmaintainable.

However, I still believe that a common authentication layer is A Good Thing:

- Access Control is a problem that has been solved in many good ways. Each of these ways is well tested and widely used. Reconsidering solved problems is only feasible in education and research. Rails is all about pragmatism. Why should you waste time thinking about new way to solve a problem as authentication?

- Face it: We don't write perfect code (this lies in the nature of humans). If we roll our own authentication code, it should have the quality of Knuth's TEX(he sends you a cheque if you find an error and has not send one for many years). If you used a common authentication layer that is well tested then the time you invest in checking that it is really so is added to the time others looked at it. Thus more time in checking that it works well has been invested into it than you could with your own time.

- If you use a copy and paste approach to using software, applying patches becomes hard. If you add a feature (or squash a bug) to the code used in your latest application that you want your other applications to have

(or not to have in the case of a bug) then you have to apply the patch manually to all other applications. A central, seperated component is upgraded much more easily.

Additionally, with the advent of Rails Engine, the Rails community now has a good foundation for building reuseable components. And with the Ruby programming language, not everything must be configureable: You can overwrite everything and everything. Why not change the behaviour of a component's class? On updates, you might still have to adjust your overrides to internal changes but these are in a different location and you can upgrade the component's code seperated from it.

Okay, so much for arguing that components make sense in some cases. I promise the rest of the manual is all about understanding ActiveRBAC and getting things done with it.

Manuel Holtgrewe

2 Tutorial

Attention, Windows users. In this guide, we use Unix syntax for commands executed at the command line (or shell as Unix people say). You only have to do the following things to run this tutorial as a Windows user: First, replace every occurance of a slash (/) into a backslash (\). Second, replace every call to a script in the *script* directory that looks like `./script/some_name` by `ruby script\some_name`. That's it.

In this tutorial, we walk through writing a small demo application and securing it using ActiveRBAC using *roles*. We try to motivate and explain each step and will also explain some things that are more related to Ruby On Rails than to ActiveRBAC. We all know how frustrating learning new things through failure can be (arguably it is a good way, though) and thus we will try to take all the stones out of our way.

We will assume that you already have Ruby On Rails 1.1 installed. Any later or previous version *might* work, but do not blame us if they do not. This means, specificially, that *EdgeRails/Trunk Rails might not work*.

This tutorial will also keep theoretical explanations to a minimum. If you want to have detailed information about how the RBAC model is implemented in ActiveRBAC then read chapter 3 and the API reference.

2.1 Creating Our Sample Application

We will first create the small application we want to protect later. We call it *MiniCMS*. The creation of the applicatioin is not very interesting since we will only use scaffolding for a very simple `Article` class. You might want to skip this section and go to section 2.2 instead.

In our application, we will have only the model class `Article` and the `ArticlesController`. `Article` objects represent articles - who would have thought so? Articles have an id, time stamps for when they were created and last updated, a title and a body. The SQL schema in figure 2.1 sums all this up. The schema is in PostgreSQL dialect but you should be able to translate it into any database vendor dialect.

```
CREATE TABLE articles (
  id BIGSERIAL NOT NULL,
  created_at TIMESTAMP NOT NULL,
  updated_at TIMESTAMP NOT NULL,

  title CHARACTER VARYING(200) NOT NULL,
  body TEXT NOT NULL,

  PRIMARY KEY (id),
  UNIQUE (title)
);
```

Figure 2.1: SQL schema for our Article model

2.1.1 Creating the Rails project

So, let us create the Rails minicms project by `rails minicms`. As we would do in every Rails project, the first thing is to remove the file *public/index.html* so we can actually let our controllers display something: `rm public/index.html`.

Then, we have to configure our database in *config/database.yml*. Documentation about how to configure this file can be found in the comments inside the file. After this configuration, create at least the development database in your database server so we import our schema from figure 2.1. You can find a file with this content and a translation into the MySQL dialect in the finished *minicms* demo which is located the downloadable tarball archive of ActiveRBAC.

2.1.2 Creating the Controllers

Now that we have set up the database with our schema we can go ahead and perform scaffolding for our Article model class. Scaffolding will create a controller for administrating articles. Execute the following on your command line: `./script/generate scaffold Article`.

We could now go ahead and make the page pretty but that is out of the scope of this tutorial. We will merely add a link list to the layout template in *app/views/layouts/articles.rhtml*. The lines we added are shown in figure 2.2.

If we point our browser at `http://127.0.0.1:3000/articles`, we should get something like shown in figure 2.3.

Now we have a very basic application to manage articles. We will use it to demonstrate how we can protect applications with ActiveRBAC. Though it is a toy application, you should easily be able to transfer the methods below to your certainly bigger and more complex application.

```
<ul>
  <li><%= link_to 'New Article', :action => 'new' %></li>
  <li><%= link_to 'List Articles', :action => 'new' %></li>
  <li><%= link_to 'View Article #1', :action => 'show', :id => 1 %></li>
</ul>
```

Figure 2.2: Our minimal navigation

Figure 2.3: The article list from our minimal CMS.

2.2 ActiveRBAC enters the scene

2.2.1 Installing Engines and ActiveRBAC

ActiveRBAC is a *Rails Engine*. This means it is a rails plugin that has a very similar structure as normal projects that use Ruby On Rails. We will, however, need to install the `engines` plugin to be able to do this. As of Rails 1.0 we can use Rail's built in plugin manager script.

First, we have to get a list of all plugins available through the plugin manager. Enter `./script/plugin discover` at your shell in your Rails project's directory. Then, you can install Rails Engines by typing `./script/plugin install engines`[1].

After you have installed the `engines` plugin, you can go ahead and install ActiveRBAC. The easiest way is - again - to use Rails's plugin manager. Type `./script/plugin install active_rbac` at your shell and ActiveRBAC will be installed in *vendor/plugins*[2].

Then, you have to make your Rails application aware of the fact that you have installed the ActiveRBAC engine. You have to add the line `Engines.start :active_rbac` to your *config/environment.rb*.

ActiveRBAC needs some database tables which can be created with the migrations inside the ActiveRBAC engine [3]. You can execute the engine's migration and let it create the necessary tables by executing `rake db:migrate:engines` at the shell in your project's directory.

Note that we currently only have schema definition files for MySQL, PostgreSQL and MS SQL Server. If your database server system is not supported then you can create a schema file for your database server with the following steps:

- Choose a one of the files *create.*.sql vendor/plugins/active_rbac/db* you want to base your schema on.
- Create a schema with the same tables, fields and corresponding field types.
- Save it as *create.your_database_name.sql* in *vendor/plugins/active_rbac/db* where *your_database_name* is the name of your database as you would enter it in *config/database.yml*.

[1] This is the simplest way to install the plugin. You could use subversion to check out the latest `engines` version from the SVN repository: `svn co http://svn.rails-engines.org/plugins/engines`. You can learn more at http://rails-engines.org/download

[2] Again, you could check out the most current ActiveRBAC from our subversion repository. The URL for this is https://activerbac.turingstudio.com/source/active_rbac/trunk/active_rbac/

[3] See http://wiki.rubyonrails.org/rails/pages/UsingMigrations and http://wiki.rubyonrails.org/rails/pages/UnderstandingMigrations for more information about migrations

- Send the migration file to our mailing list at mailto:rbac-dev@lists.cloudcore.com.

The next thing to do is to do some configuration. First, we have to adapt our *config/routes.rb*. We add the lines from figure 2.4 to our *config/routes.rb* right at the beginning right after the `do |map|`.

```
# ActiveRbac's RegistrationController confirmation action needs
a special route
map.connect '/active_rbac/registration/confirm/:user/:token',
            :controller => 'active_rbac/registration',
            :action => 'confirm'
```

Figure 2.4: We need a special route for the RegistrationController

Now, we have to configure ActiveRBAC a bit and tell it which layout to use. The layout file *app/views/layouts/articles.rhtml* was created when performing scaffolding for the `Article` model. We set the layout to use to this file as shown in figure 6.2.2. In a real application you might want to have a `admin` layout with a special admin menu. Note that changes to *environment.rb* only take effect if you restart your server.

Add the lines from 2.5 to your *config/environment.rb* (you added the line `Engines.start :active_rbac` already above). You can find a complete list of configuration options in section 6.2.2.

```
module ActiveRbacConfig
  # controller and layout configuration
  config :controller_layout, "articles"
end

Engines.start :active_rbac
```

Figure 2.5: Configure ActiveRBAC and start it.

After this, we have ActiveRBAC up and running. We have yet to protect something but if you point your browser at http://127.0.0.1:3000/active_rbac/user then you should see an administration screen as in figure 2.6.

2.2.2 Creating initial Role and User

We are going to protect our `ArticlesController` and the controllers of ActiveRBAC in the next steps so it is a good idea not to lock ourselves out of the application. We have to create a role that is allowed to access the

Figure 2.6: An empty ActiveRBAC user list.

`ArticlesController` as well as the ActiveRBAC controllers and a user that this role is assigned to.

There are multiple ways to do this:

- We could write some SQL code to create the user and role directly in the database. Obviously this is error prone and not elegant.
- We could use the ActiveRBAC web interface as long as it is unprotected.
- We could create a migration that creates the user and role programmatically.
- We could fire up a console with `./script/console` and create the user and role there programmatically.

We will use the last option but we hould keep in mind that there are many ways to edit users and roles (though using plain SQL should be our last resort). Migrations are a very nice thing in real world applications.

Start a console with `./script/console` at the shell in your project's directory and execute the commands shown in figure 2.7.

This will create a user with the login *Admin* and the password *password*. The user has a role with the title *Admin*.

2.3 Protecting with Roles

With the steps above, we have set up a minimal CMS application and configured it to use Rails Engines ActiveRBAC. We can now protect this CMS in different ways. The following sections demonstrate two possible ways.

Create a backup copy of this minimal project so we can explore multiple different ways of protecting it with ActiveRBAC.

```
role = Role.new
role.title = 'Admin'
role.save

user = User.new
user.login = 'Admin'
user.update_password 'password'
user.email = 'root@localhost'
# the next line is essential
user.state = User.states['confirmed']
user.save

user.roles << role
user.save
```

Figure 2.7: Create the initial user and role.

2.3.1 Protecting the ArticlesController with Roles

The first way to protect your controllers with ActiveRBAC is checking that the user has the necessary *role* he needs to access a controller and/or action. We call this *protection by roles*. *Protection with roles* can be extended by *protection with permissions* which is explained in section 2.3.4.

We can find the currently logged in user's data in `session[:rbac_user]` after the user has logged in. We can then use the `User` object's `has_role` method to check whether he has the necessary role. `has_role` expects one or more strings and returns `true` if the user has a role whose title equals one of the strings given to the method in the call as parameters. We could call this method like `session[:rbac_user].has_role("Admin")`, for example.

The very first thing that comes into our mind is to protect our `ArticlesController`. We want to make sure that no user that does not have the *Admin* role can access any but the `show` and `list` actions.

We could now go ahead and simply add checks for the user having the Admin role in all actions we want to protect but that would add redundancy to our code and make it harder to maintain. It is much cleaner to add our access control code into a `before_filter`.

`before_filter`s are methods of controller objects that are called before any action is invoked when the user tries to access the action via a URL. If a `before_filter` returns `false` then the action method is not executed and if it returns `true` then it is. It is pretty straightforward to add our access control code here. We add the lines from figure 2.8 after the last method definition in our *articles_controller.rb*.

We put the before filter into the `protected` context of the class so it cannot

```
protected

  before_filter :protect_controller, :except => [ :list, :index,
:show ]

  def protect_controller
    if !session[:rbac_user].nil?  and session[:rbac_user].has_role("Admin"
      return true
    else
      redirect_to "/articles/list"
      flash[:notice] = "You are not allowed to access this
page"
      return false
    end
  end
```

Figure 2.8: Protecting `ArticlesController` using a `before_filter`.

be invoked as an action by requesting a URI from our application. In line 3, we register our method with `before_filter` as a before filter method. By using the `:except` option of `before_filter`[4] we can skip our authorization filter.

Our before filter works the following way: When we are satisfied with the user's role, i.e. we actually have a registered user and the user has the *Admin* role then we return true in the before filter and let the action execute. Otherwise, we set a notification about the missing permissions into the flash and redirect to the article list.

If we wanted to be fancy and redirect the user to the login screen and back there to the current location if he has logged in successfully then we could also do this easily with ActiveRBAC. See section 7.2.2 for detailed instructions.

You can see the result output from a request to our protected `ArticlesController` in figure 2.9. If you want to log in, you have to point your browser at `http://127.0.0.1:3000/active_rbac/login/login` and you can do so there with the login created above[5].

That is all you have to do to protect your `ArticlesController`. You can learn more about good ways to protect your controllers in chapter 4. However, there is one thing we have yet to do: Protect ActiveRBAC itself.

[4] As documented in `http://api.rubyonrails.org/classes/ActionController/Filters/ClassMethods.html`, *Filter chain skipping*.

[5] Yes, this URL is ugly but we could change by customizing *routes.rb* it if we wanted to. See section 7.2 for more information

Figure 2.9: Thou shall not pass - our `ArticlesController` is protected now.

2.3.2 Protecting ActiveRBAC itself with Roles

When it is installed, ActiveRBAC is not protected in any way. Let us repeat that *ActiveRBAC does not protect itself and we have to do it manually*. The developers of ActiveRBAC decided that this is better than to force an access controll schema on us.

Our first impulse is just to edit every controller in *vendor/plugins/active_rbac* and add a before filter as we learned above in section 2.3.1. Though this would work, we would have to do this every time we upgrade our ActiveRBAC engine. So, on second thought this does not seem to be the best way.

However, the Rails Engines plugins comes to help us out. Since ActiveRBAC is an Engine, we can use all the goodness the Engines plugin offers us. In this case, we can add or override methods in the ActiveRBAC plugin by writing code *inside our own application*. We just have to make sure we create the files at the same path with the same name as we would place it in the ActiveRBAC plugin.[6].

We will add a before filter to the `ActiveRBAC::ComponentController` which is the parent class of all of ActiveRBAC's controllers. You can see the necessary code in figure 2.10. Place it into *app/controllers/active_rbac/component_controller.rb*. Since we only want to protect the administration controllers and not `RegistrationController` or `LoginController` we have an additional line of code compared to the code in figure 2.8 but otherwise the two code pieces are pretty much the same. Also note that the `:except` option of `before_filter` can only exclude actions from being filtered and not specific controllers.

[6] If you want to learn more about Rails Engines and what it can do for you, read the documentation at http://api.rails-engines.org/engines/files/vendor/plugins/engines/README.html.

```
# Mix in the a before_filter into ActiveRbac::ComponentController
to secure
# all ActiveRbac controllers.
class ActiveRbac::ComponentController
  before_filter :protect_with_active_rbac

  protected

    def protect_with_active_rbac
      # only protect certain controllers
      return true if [ActiveRbac::LoginController,
          ActiveRbac::RegistrationController].include?(self.class)

      # protect!
      return true if not session[:rbac_user].nil?  and
          session[:rbac_user].has_role 'Admin'

      flash[:notice] = 'You have insufficient permissions!'
      redirect_to '/articles/list'
      return false
    end

    def protect_me
    end
end
```

Figure 2.10: Protecting `ActiveRBAC::ComponentController`.

After adding this file we have also protected the administration controllers of ActiveRBAC. That was not that hard, was it? Now we have a simple but well protected application and that is all you have to know to start using ActiveRBAC.

2.3.3 Improving the templates

Now we have authentication built into our application controllers, it might be nice to make the views to be aware of this, too. We should hide all links to the actions that could be used to edit the database's contents and display a link for the user to log in or log out, depending on whether he is authenticated.

However, this is not hard at all and we can see an example of how to do this in figure 2.11. The code is from the *articles.rhtml* layout file we already mentioned above.

```
<ul>
  <% if !session[:rbac_user].nil?  and
       session[:rbac_user].has_role("Admin") %>
  <li><%= link_to 'Log Out', :controller =>'/active_rbac/login',
            :action => 'logout' %></li>
  <li><%= link_to 'New Article', :controller =>'/articles',
            :action => 'new' %></li>
  <% else %>
  <li><%= link_to 'Log In', :controller =>'/active_rbac/login',
            :action => 'login' %></li>
  <% end %>
  <li><%= link_to 'List Articles', :controller =>'/articles',
            :action => 'new' %></li>
  <li><%= link_to 'View Article #1', :controller
=>'/articles',
            :action => 'show', :id => 1 %></li>
</ul>
```

Figure 2.11: An authentication aware navigation.

2.3.4 Protecting the ArticlesController with Permission

TODO

3 The Concepts Behind RBAC

ActiveRBAC is a RBAC (Role Based Access Control) implementation. Thus, it makes sense to understand what RBAC is and how ActiveRBAC implements it.

3.1 Users, Groups, Permissions

RBAC is a way of organizing permissions in a computer system. Instead of assigning permissions to a resource based on ownership (i.e. assigning permissions to users), permissions are granted to roles. Roles are then assigned to users.

For example, we could have the three users Alice, Bob and Ceasar on our file server. We have two roles: Administrators and Users. The User role is assigned to Alice and Bob and Ceasar has the Administrator role. We would now grant permissions to read files to the User role and permissions to read, write and delete files to the Administrator role. Thus, Alice and Bob can read files and Ceasar can also write and delete those files. This use case is show in figure 3.1.

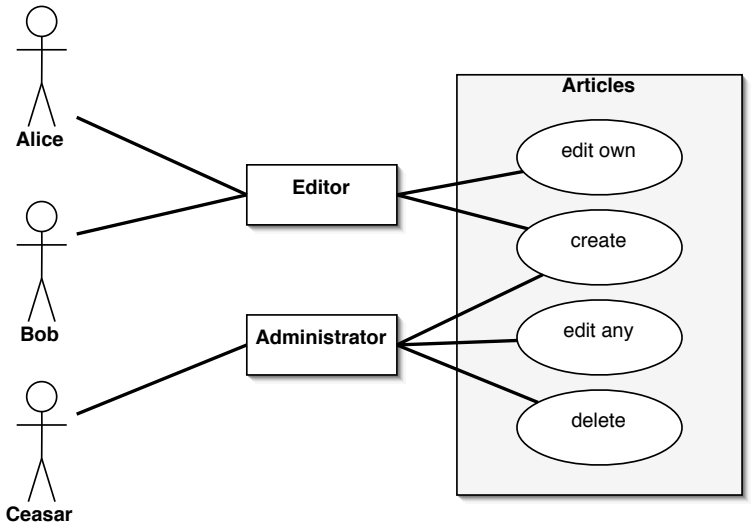

Figure 3.1: A simple RBAC use case.

This is conceptually the same as organizing users into groups and only as-

signing permissions to these groups and not directly to users. However, there is a bit more to roles than to - say Unix - groups as we will see below.

Note, that permissions could be more complex like may edit the files if he is the creator of the file or may edit the file if he is in the contributor list for this file. This way, you can create hybrid systems that also include ACL (Access Control Lists) or any other thing you that fits your requirements.

Additionally, you can choose freely how you store the permissions. Let us consider how you could protect articles ob jects in your database.

- Store all permissions in the database. You could have a table storing the ID of your article, the action that can be performed on it (edit, delete, rename) and a role id. This way, all permissions are stored in the database. While this is very flexible, it is also very complex.
- Maybe your permission schema for the articles is very simple: Users with the Administrator role can edit all articles and users with the Editor role can only edit the articles they have created. You can then hardcode this information in your code and check for the condition like to be true: The user has the Administrator role or is the creator of this article.

The second point is important, so let us stress it once again: Sometimes it is better to use only a part of the ActiveRBAC library because we can simplify our access control schema. This reduces complexity in our application, in the user interface and possible security flaws because of too great complexity or wrong usage on the user's side.

We can formalize the things we said about users, roles and permissions a bit: A RBAC system contains users, roles, permissions and objects. Ob jects provide actions, permissions to execute an action on a given ob ject can be granted to roles. Roles can then be assigned to users. Users are granted permissions to perform actions on ob jects through the roles assigned to them.

3.2 Role Inheritance

The description of RBAC up to this point describes something often called Core RBAC, which means the most simple RBAC set that is interesting. In bigger access control setups, we might require some more power.

Let us consider a more complex permission setup: We extend our CMS (where the articles were managed with) a bit and sell it to a company who want to manage their web site through it (yes, yet another CMS - the world has just waited for that). Our simple access control setup does not work for them since they want to audit all articles by some users who have the Supervisor role but they do not want to give their supervisors the full administration role. Supervisors should be able to do exactly the same as editors but they can also mark an article as published.

We could now go simply ahead and copy the editor role to a new supervisor role and assign the additional permission to publish an article to it. But what happens if we add images to our CMS and editors should be able to upload images? We would have to perform the changes to the editor role's permissions to the supervisor's, too. As we realize, this is not perfect since it complicates things greatly.

Another way would be to create a Supervisor role and assign this role to supervisor users besides their editor role. This is not the best way to do it, too, since we would need a lot of roles that are granted one or two permissions for users that can do almost the same thing as those with an existing role but one or two things more. Maybe we could also try to convince our customer that they want a simple piece of software since it is so much easier to use and it is less likely to cause problems.

However, as Einstein said, *things should be as simple as possible but not simpler* and for some big software pro jects, you might eventually just need a complex permission schema that raises problems as described above. But don't despair, role inheritance is there to help you.

You can specify a role to extend another role and thus to inherit their parent role's permissions. This works just like one class extending another one in most ob ject oriented languages like Ruby or Java. One advantage is that we circumvent to introduce redundancy by the copy-and-extend approach described above. The second advantage is that since all users who are assigned the supervisor role are automatically also assigned the editor role through inheritance: Since supervisor extends editor, all supervisors are editors.

So, if we have the same setup above with Alice, Bob and Ceasar and introduce a new supervisor user Daedalus, our permission setup would look like in figure 3.2.

The concepts described up to this point are called Hierarchial RBAC. There are further extensions to Hierarchial RBAC that are less interesting in web development and thus not implemented in ActiveRBAC.

3.3 Groups

However, ActiveRBAC extends Hierarchial RBAC by a concept parallel to roles: Groups. Groups can be assigned to users but roles can also be assigned to groups. This way, users can be granted permissions through roles assigned to groups which are again assigned to them. Groups can also be arranged in a tree structure. Roles assigned to a group are inherited by the group's children. When a group is assigned to a user, the user automatically gets all permissions of the group's parents, too.

It is pretty complex to describe this structure and in fact it makes many permission schemas more complex. However, if your overall access control requirements are complex the power added by this complexity might even reduce

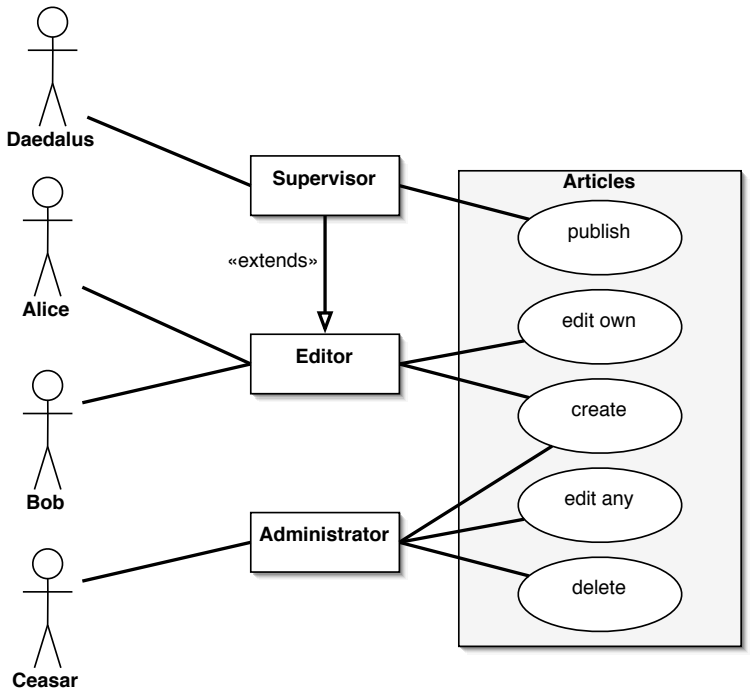

Figure 3.2: A use case with role inheritance.

the number of roles and role assignments. You can think about groups as role macros.

Additionally, groups let you manage your users without necessarily assigning permis- sions to them: Groups can have no roles assigned to them.

In many cases, you will not need groups in your permission setup and roles will be sufficient. If your pro ject, however, reaches the critical complexity, ActiveRBAC's groups are there, waiting for you.

4 Protection Patterns

This chapter describes common patterns to protect your controllers. If you want to redirect your users after logging in then look at section 7.2.2 for more information.

4.1 Using before_filter

The ActionController::Base class allows for declaring `before_filters`. These filters can be methods that are executed before the action method is called. If these filters return true then the action method is rendered correctly. If they return false then the action method is not invoked and a blank page is rendered.

Most of the time we want to tell the user that he does not have the necessary permissions to access the requested controller action and thus redirect him to the front page and set the `flash`'s `:error` entry to notice the user.

We can accomplish this by the code in figure 4.1.

```
class MyController < ApplicationController
  before_filter :protect_controller

  protected
    def protect_controller
      if [USER IS ALLOWED? CONDITION]
        return true
      else
        redirect_to "/"
          flash[:error] = "You are not allowed to access this
page"
        return false
      end
    end
end
```

Figure 4.1: Using before_filter to protect controllers.

4.2 Check that the user is logged in

The simplest thing that could possibly work is checking whether the user is logged in or not. We can accomplish this by looking in the session hash.

The current user is stored in the session hash as `session[:rbac_user]`. You can access the session hash in all your controllers. If the entry with the key :rbac_user is nil then the user is logged in.

We can perform this check by modifying the code in figure 4.1 as show in figure 4.2.

```
def protect_controller
  if !session[:rbac_user].nil?
    return true
  else
    redirect_to "/"
    flash[:error] = "You are not allowed to access this page"
    return false
  end
end
```

Figure 4.2: Using before_filter to protect controllers.

4.3 Checking for specific roles

Sometimes it is simply enough to check that the currently logged in user (if any) has a given role. To do this, the User class provides the `has_role` method. You pass this methods one or more strings and the method will return true if the user ob ject has one of the given roles.

Our protection could would look as in figure 4.3.

4.4 Protecting only selected actions

What should you do if you want to use a before_filter to protect only some of your actions in a controller? Well, the most elegant way is to the either the `:only` or `:except` parameter of the `before_filter` macro of `ActionController::Base`[1].

You can see an example of that in figure 4.4.

[1] As documented in http://api.rubyonrails.org/classes/ActionController/Filters/ClassMethods.html, *Filter chain skipping*.

```
def protect_controller
  if !session[:rbac_user].nil?  and
      session[:rbac_user].has_role("Admin", "Editor")
    return true
  else
    redirect_to '/'
    flash[:error] = "You are not allowed to access this page"
    return false
  end
end
```

Figure 4.3: Checking for roles

```
before_filter :protect_controller, :only => :delete

def protect_controller
  if !session[:rbac_user].nil?  and
      session[:rbac_user].has_role("Admin", "Editor")
    return true
  else
    redirect_to '/'
    flash[:error] = "You are not allowed to access this page"
    return false
  end
end
```

Figure 4.4: Protecting only selected actions.

4.5 Using a common authentication failure handler

It might be convenient to write a common authentication failure handler method in your ApplicationController. If you want to change what will happen on authentication failure (redirect to a specific failure URL or to the log in page, for example), you can do this in one common place. You can see this pattern applied in figure 4.5.

You must not forget, however, to return false in your filter. The auth_failed method will only change the state of the controller. redirect_to does not immetiately redirect the user but only tells the controller that it should redirect to the speficic URL after performing the action selected in the URL.

```
class ApplicationController

  protected
    def auth_failed(message)
      # do extra logging
      # maybe send email to admin
      flash[:error] = message
      redirect_to '/'
    end
end

class MyController < ApplicationController
  before_filter :protect_controller

  protected

    def protect_controller
      return true if params[:action].to_s != "delete"

      if !session[:rbac_user].nil?  and
        session[:rbac_user].has_role("Admin", "Editor")
        return true
      else
        auth_failed("You are not allowed to access this page")
        return false # make sure you still return false here!
      end
    end
end
```

Figure 4.5: Using a common auth failure handler.

4.6 Protection inside the methods

Sometimes you might want really fine grained control about what is allowed inside one of your controller's methods (or in rails speak: actions). For example, in a CMS the update action of the `Admin::ArticleController` might be accessible by Users with the Administrator and Editor role. However, you only want to allow changes to the state property of the article by Administrators.

Again, this is not difficult. Figure 4.6 shows how this works.

```
class MyController < ApplicationController
  def update
    @article = Article.find(params[:id])

    # delete the submitted article state
    if session[:rbac_user].nil?  or !session[:rbac_user].has_role("Admin")
      params[:article].delete(:state)
    end

    if @article.update_attributes(params[:article])
      flash[:notice] = 'Article was successfully updated.'
      redirect_to :action => 'show', :id => @article
    else
      render :action => 'edit'
    end
  end
end
```

Figure 4.6: RBAC protection from within methods.

5 Permissions Schema Patterns

This chapter contains common permission schema patterns with examples. A permission schema is the way you organize your users, roles, groups and permissions.

5.1 Logged in Users only

The simplest thing that could possibly work is checking whether the current user has logged in. You do not need any roles to do this.

You can check whether the user has logged in using `session[:rbac_user].nil?` both in your templates and in your controller code.

5.2 Administrator, Editor

A simple content management system could require *editors* who are allowed to create articles and edit their own articles and *administrators* who are allowed to create, edit and delete all articles. The schema is visualized in figure 5.1.

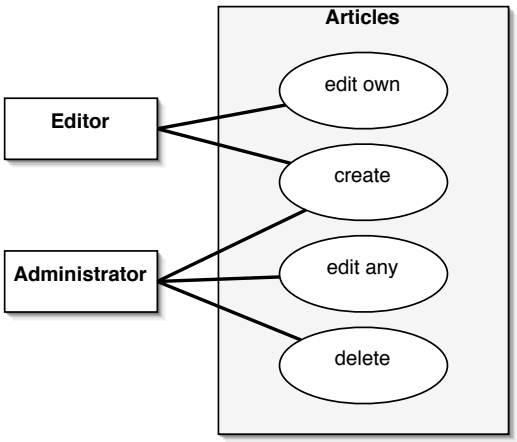

Figure 5.1: A RBAC schema for a simple CMS.

5.3 Administrator, Editor, Supervisor

A slightly more complex content management syste could require a *Supervisor* role besides administrators and editors. The supervisors are allowed to publish or unpublish an article. This could be done by setting a property of the `Article` object, for example `state`.

The supervisor role *extends* the editor role so all supervisors are also editors. This works like class inheritance in Ruby and Java. This way we reduce redundancy in the assignment of roles to users and permissions to roles as described in section 3.2.

The schema is visualized in figure 5.2.

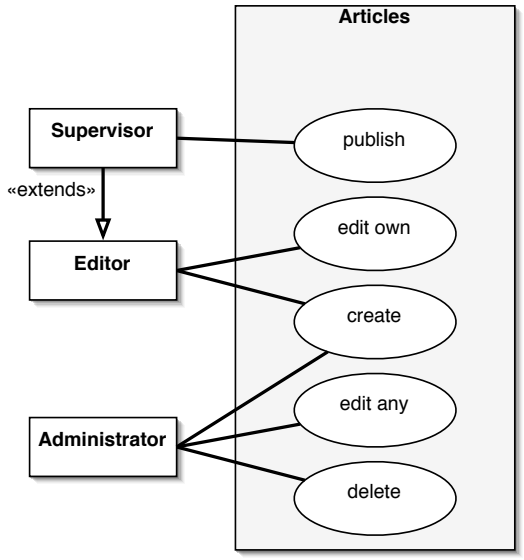

Figure 5.2: A RBAC schema for a CMS with supervisors.

5.4 Administrator, Editor, Subscriber

A variation of the simple CMS schema is to add a *Subscriber* role. This role is assigned only to users who subscribed to your web page and pay a monthly rate for some articles. For example, many newspapers and magazine publish some articles for free on their web site but for the more interesting (not to say: better) articles are only available to users who subscribed to their site.

So we introduce a subscriber role and add a `premium_content` property to our `Article` class that stores whether this content is only available to subscribers in a boolean value. We can then check if an article is visible to the current user by the protection pattern described in section 4.6.

See figure 5.3 for a diagram of this schema.

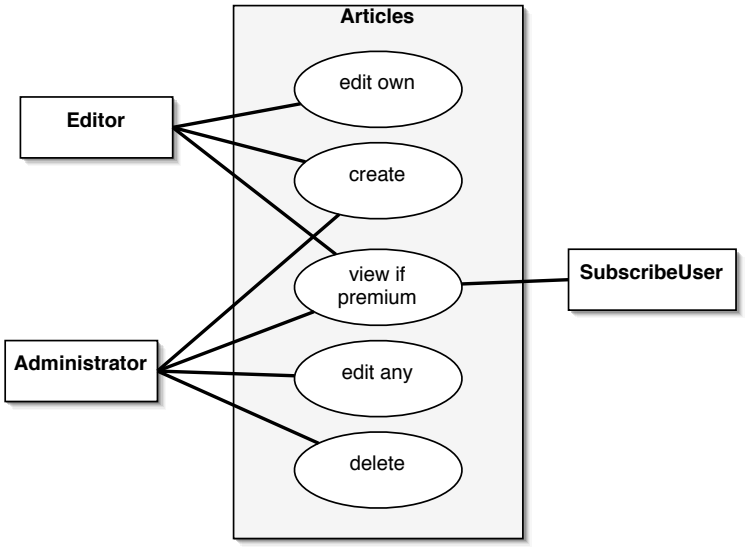

Figure 5.3: A RBAC schema for a CMS with subscribers

6 Reference

6.0.1 How ActiveRBAC uses the flash

ActiveRBAC will use Rail's flash to tell the user things things like *You have been logged in successfully.* or *This user could not be found.* It will either use the `:notice` or the `:error` entry of the flash.

Your layout should display these two flash entries as you think appropriate (i.e. display the notices in a green and the errors in a red box). *If you do not show the flash's content to your users, they might miss important information.*

6.1 The ActiveRBAC Model Classes

6.2 Configuration

6.2.1 How to set configuration settings.

You can change ActiveRBAC's configuration by placing a `ActiveRbacConfig` module definition in your *environment.rb* and performing `config` calls in it. You can see an example in figure 6.1.

```
module ActiveRbacConfig
  # controller and layout configuration
  config :controller_layout, "articles"
end

Engines.start :active_rbac
```

Figure 6.1: Configure ActiveRBAC and start it.

6.2.2 Configuration Settings

You can find an explanation of all configuration settings in table 6.1. If you ever want to have have a look at where they are defined then *active_rbac/lib/active_rbac_config.rb* is what you are searching for.

setting name	explanation	default value
`:mailer_from`	The *Sender:* field of emails sent by ActiveRBAC	ActiveRbac <activerbac@localhost>
`:mailer_subject_confirm_registration`	The subject used for registration confirmation emails.	'Please confirm your registration'
`:mailer_subject_lost_password`	The subject used for password retrieval emails.	ActiveRbac 'Your new password'
`:mailer_headers`	A hash with additional headers	`Hash.new`
`:controller_layout`	The layout to use for ActiveRBAC controllers	'application'
`:controller_registration_signup_fields`	Additional signup fields	`Array.new`
`:model_default_hash_type`	The default hash type to use for passwords	'md5'

Table 6.1: ActiveRBAC's configuration settings

7 Howtos

7.1 How to create an initial RBAC schema elegantly

When deploying your application, you might want to set up the same initial user and role set on every installation. Rail's migrations[1] come in handy for recording such database evolution.

You can create a migration by executing `./script/generate migration InitialRbacSchema` at the shell in your project's directory. This will create a new, empty migration in *db/migrate*. You can create your initial users, roles and permissions programmatically. Figure 7.1 shows an example of such a migration.

7.2 How to make ActiveRBAC's URLs prettier

By default, ActiveRBAC's controllers will be accessible as */active_rbac/controller_name*. Of course, this is absolutely not acceptable for an application to be deployed. We have to change these URLs but Rails makes it very easy for us.

Figure 7.2 gives an example of prettified routes:

- The controllers `ActiveRbac::UserController`, `ActiveRbac::RoleController`, `ActiveRbac::GroupController`, `ActiveRbac::StaticPermissionController` are mapped so they are available under */admin/arbac/controller_name*.
- The `ActiveRbac::LoginController`'s actions `login` and `logout` are mapped directly to the root URLs */login* and */logout*
- The `ActiveRbac::RegistrationController` is mapped to */path*.
- All URLs below */active_rbac* are mapped to a `ErrorController` which would display an error page

The nice thing of Rail's routes feature is that when you redirect with the code `redirect_to :controller => '/active_rbac/login', :action => 'login'` then the user will be redirected to */login* now that we have set up `ActiveRbac::LoginController`'s routes with as in figure 7.2.

[1] See the documentation at http://api.rubyonrails.org/classes/ActiveRecord/Migration.html

```
class InitRbacSchema < ActiveRecord::Migration
  def self.up
    role = Role.new
    role.title = 'Admin'
    role.save

    user = User.new
    user.login = 'Admin'
    user.update_password 'password'
    user.email = 'root@localhost'
    # the following line is essential!
    user.state = User.states['confirmed']
    user.save

    user.roles << role
    user.save
  end

  def self.down
    User.find_by_login('Admin').destroy
    Role.find_by_title('Admin').destroy
  end
end
```

Figure 7.1: A migration to create an initial role and use.

```
# map the admin stuff into '/admin/'
map.connect '/admin/arbac/group/:action/:id',
    :controller => 'active_rbac/group'
map.connect '/admin/arbac/role/:action/:id',
    :controller => 'active_rbac/role'
map.connect '/admin/arbac/static_permission/:action/:id',
    :controller => 'active_rbac/static_permission'
map.connect '/admin/arbac/user/:action/:id',
    :controller => 'active_rbac/user'

# map the login and registration controller somewhere prettier
map.connect '/login', :controller => 'active_rbac/login',
    :action => 'login'
map.connect '/logout', :controller => 'active_rbac/login',
    :action => 'logout'
map.connect '/register/confirm/:user/:token',
    :controller => 'active_rbac/registration',
    :action => 'confirm'
map.connect '/register/:action/:id',
    :controller => 'active_rbac/registration'

# hide '/active_rbac/*'
map.connect '/active_rbac/*foo',
    :controller => 'error'
```

Figure 7.2: Prettifying ActiveRBAC's routes.

7.2.1 How to set the controller layout for each controller

By default, the ActiveRBAC controllers use the layout template as set in configuration value `:controller_layout` when rendered (see section 6.2.1 and 6.2.2 for more information).

However, in some cases we want to change this so the administration classes (`ActiveRbac::UserController` and so on) have a different layout than the LoginControllers. We can do this by creating a class file in your *app/controllers* directory. Its name and path must be the same as the file defining this controller in the ActiveRBAC engine.

For example, to override things in *ActiveRbac::UserController*, we create a file defining this class having the name and path *app/controllers/active_rbac/user_controller.rb*. In this class definition, we call the class method `layout` with the name of the layout we want this controller to use - just as we would if we would define this controller yourself. See figure 7.3 for an example.

```
class ActiveRbac::UserController < ActiveRbac::ComponentController
  layout 'admin'
end
```

Figure 7.3: Configure ActiveRBAC and start it.

7.2.2 How to redirect to a URL after login

When we redirect the user to the login page as described in chapter 4 and the user logs in then he will see the Congratulations, you have logged in successfully page. After this, he has to find his way to the page he wanted to see originally again.

This is not optimal. It would be much better to redirect the user back to the page he wanted to see the first place. `LoginController`'s `login` can do this for us if we tell it where it should redirect to. It will also set the `:notice` entry of `flash` that login has been successful before redirecting.

How can we tell `LoginController` that it should redirect somewhere? There are two options:

- We can pass `LoginController` the URL to redirect to in the `return_to` parameter.
- We can set the `:return_to` session value to specify the location to redirect to.

If we set the `:return_to` parameter to the target URL or path when redirecting to the login with the `redirect_to` method then this URL/path will be

passed to the `LoginController` as a query parameter. This means that *?return_to=TARGET_URL* will be appended to the URL redirected to. Thus we do not have to add an additional route for this.

If we use the `:return_to` session field then we can either pass in a URL or path to redirect to or we can set this field to a Hash with a controller and action name (and parameters) as we would when using `return_to`.

Figure 7.4 shows some examples of lines that you could write into your protection before filters.

```
# How to LoginController to redirect after logging in.

# redirect to server root
redirect :controller => '/active_rbac/login', :action =>
'login',
  :return_to => '/'
# redirect to absolute URL
redirect :controller => '/active_rbac/login', :action =>
'login',
  :return_to => 'http://www.example.com/foo/bar'

# one of these lines could be in our code before redirecting

# redirect to current controller, action and id
session[:return_to] = { :controller => params[:controller],
:action => params[:action],
  :id => params[:id] }
# another way to redirect to server root
session[:return_to] = '/'
```

Figure 7.4: Telling `LoginController` to redirect after login.

7.2.3 How To Change ActiveRBAC's Views/Templates

It's easy to change the views provided by ActiveRBAC: Simply locate the file you want to change in ActiveRBAC's engine directory. Let's assume you want to provide an updated version of *vendor/plugins/active_rbac/app/views/active_rbac/user/show.rhtml*.

Create a new file *in your app directory* with the name and path *app/views/active_rbac/user/show.rhtml*. Now place the code of your view in this file. The next time when you access the `show` action of you `UserController` will use your customized *show.rhtml*.

You can do this with any other template file[2].

7.2.4 How To Change ActiveRBAC's Controllers

Sometimes you will want to change the behaviour of ActiveRBAC's controllers beyond the scope of ActiveRBAC's configuration. This is easy since ActiveRBAC is a Rails Engine[3].

First, locate the controller and action you want to change in ActiveRBAC's engine directory. Let's assume you want to chang the `show` action of the `UserController`. This controller is defined in the file *user_controller.rb* in *vendor/plugins/active_rbac/app/controllers/active_rbac*.

Now, create a new file called *user_controller.rb* in the directory *app/controllers/active_rbac* (relative from your Rails Project's root). In this file, you define the `ActiveRbac::User` class and its `show` action as you can see in figure 7.5.

```
class ActiveRbac::UserController < ActiveRbac::ComponentController
  def show
    # we can place our own implementation here, now
    render_text 'We are to lazy to implement "show".'
  end
end
```

Figure 7.5: Overriding the action `show` in `UserController`

All additions and overwrites done in this file will be mixed into the base `ActiveRbac::UserController` provided by the Engine. This way, you can completely change and tweak any aspect of ActiveRBAC's controllers without too much configuration.

7.2.5 How To Change ActiveRBAC's Models

Modifying models defined in Engines is a bit less straightforward - but only a bit [4]. The problem is that Rails' custom include mechanism does not expect a file to be there twice where the second is to override some part of what the first file defines.

Thus, you have to reimplement the a whole model file if you want to change an spect of it. But don't be afraid: This can be done very elegantly.

[2] You can find out more about tweaking Engines at http://api.rails-engines.org/engines/files/vendor/plugins/engines/README.html.

[3] You can find out more about tweaking Engines at http://api.rails-engines.org/engines/files/vendor/plugins/engines/README.html again.

[4] See http://api.rails-engines.org/engines/files/vendor/plugins/engines/README.html for more information.

If you look at the model files in ActiveRBAC's engine directory then you will notice that they are very short and only define the model classes and include a module ending on `Mixin`. These mixin modules provide the real functionality. By including a mixin modules, the including classes become the fully fledged `User`, `Role` etc. classes.

So we have to reimplement the model in our application if we want to change an aspect of a model? Why don't we do this by importing the mixin module. This only takes a line and after this line we can put anything we want.

Defining an already existing method will override the old one and `validates_*`, `has_one` etc. macros will be executed and work as they would if you had copied and pasted all the lines from engine's directories (the mixin modules' lines).

So let's go ahead and extend the `User` class a bit. Look at figure **??** and see how easy this is.

```
class User < ActiveRecord::Base
  include ActiveRbacMixins::UserMixin

  has_one :accounting_data
  has_one :contact_information

  def welcome_message
    "Welcome #{self.login}"
  end

  def self.foo
    "BAR!"
  end
end
```

Figure 7.6: An example of tweaking the `User` model class