
AccessGranted



AccessGranted is a multi-role and whitelist based authorization gem for Rails. And it's lightweight (~300 lines of code)!

Installation

Add the gem to your gemfile:

```
1 gem 'access-granted', '~> 1.3'
```

Run the bundle command to install it. Then run the generator:

```
1 rails generate access_granted:policy
```

Add the `policies` (and `roles` if you're using it to split up your roles into files) directories to your autoload paths in `application.rb`:

```
1 config.autoload_paths += %W(#{config.root}/app/policies #{config.root}/
  app/roles)
```

Supported Ruby versions

Because it has **zero** runtime dependencies it is guaranteed to work on all supported MRI Ruby versions, see CI to check the up to date list. It might and probably is working on Rubinius and JRuby but we are no longer testing against those.

Summary

AccessGranted is meant as a replacement for CanCan to solve major problems:

1. Performance

On average AccessGranted is **20 times faster** in resolving identical permissions and takes less memory. See benchmarks.

2. Roles

Adds support for roles, so no more **ifs** and **elses** in your Policy file. This makes it extremely easy to maintain and read the code.

3. Whitelists

This means that you define what the user can do, which results in clean, readable policies regardless of application complexity. You don't have to worry about juggling `can`s and `cannot`s in a very convoluted way!

Note: `cannot` is still available, but has a very specific use. See Usage below.

4. Framework agnostic

Permissions can work on basically any object and `AccessGranted` is framework-agnostic, but it has Rails support out of the box. :) It does not depend on any libraries, pure and clean Ruby code. Guaranteed to always work, even when software around changes.

Usage

Roles are defined using blocks (or by passing custom classes to keep things tidy).

Order of the roles is VERY important, because they are being traversed in top-to-bottom order. At the top you must have an admin or some other important role giving the user top permissions, and as you go down you define less-privileged roles.

I recommend starting your adventure by reading the wiki page on how to start with Access Granted, where I demonstrate its abilities on a real life example.

Defining an access policy

Let's start with a complete example of what can be achieved:

```
1 # app/policies/access_policy.rb
2
3 class AccessPolicy
4   include AccessGranted::Policy
5
6   def configure
7     # The most important admin role, gets checked first
8     role :admin, { is_admin: true } do
9       can :manage, Post
10      can :manage, Comment
11    end
12
13    # Less privileged moderator role
14    role :moderator, proc { |u| u.moderator? } do
15      can [:update, :destroy], Post
16      can :update, User
17    end
18  end
19 end
```

```
18
19   # The basic role. Applies to every user.
20   role :member do
21     can :create, Post
22
23     can [:update, :destroy], Post do |post, user|
24       post.author == user && post.comments.empty?
25     end
26   end
27 end
28 end
```

Defining roles Each `role` method accepts the name of the role you're creating and an optional matcher. Matchers are used to check if the user belongs to that role and if the permissions inside should be executed against it.

The simplest role can be defined as follows:

```
1 role :member do
2   can :read, Post
3   can :create, Post
4 end
```

This role will allow everyone (since we didn't supply a matcher) to read and create posts.

But now we want to let admins delete those posts. In this case we can create a new role above the `:member` to add more permissions for the admin:

```
1 role :admin, { is_admin: true } do
2   can :destroy, Post
3 end
4
5 role :member do
6   can :read, Post
7   can :create, Post
8 end
```

The `{ is_admin: true }` hash is compared with the user's attributes to see if the role should be applied to it. So, if the user has an attribute `is_admin` set to `true`, then the role will be applied to it.

Note: you can use more keys in the hash to check many attributes at once.

Hash conditions Hashes can be used as matchers to check if an action is permitted. For example, we may allow users to only see published posts, like this:

```
1 role :member do
2   can :read, Post, { published: true }
3 end
```

Block conditions Sometimes you may need to dynamically check for ownership or other conditions, this can be done using a block condition in `can` method, like so:

```
1 role :member do
2   can :update, Post do |post, user|
3     post.author_id == user.id
4   end
5 end
```

When the given block evaluates to `true`, then `user` is allowed to update the post.

Roles in order of importance Additionally, we can allow admins to update **all** posts despite them not being authors like so:

```
1 role :admin, { is_admin: true } do
2   can :update, Post
3 end
4
5 role :member do
6   can :update, Post do |post, user|
7     post.author_id == user.id
8   end
9 end
```

As stated before: **:admin role takes precedence over :member** role, so when AccessGranted sees that admin can update all posts, it stops looking at the less important roles.

That way you can keep a tidy and readable policy file which is basically human readable.

Usage with Rails

AccessGranted comes with a set of helpers available in Ruby on Rails apps:

Authorizing controller actions

```
1 class PostsController
2   def show
3     @post = Post.find(params[:id])
4     authorize! :read, @post
5   end
6 end
```

```
7   def create
8     authorize! :create, Post
9     # (...)
10  end
11 end
```

`authorize!` throws an exception when `current_user` doesn't have a given permission. You can rescue from it using `rescue_from`:

```
1 class ApplicationController < ActionController::Base
2   rescue_from "AccessGranted::AccessDenied" do |exception|
3     redirect_to root_path, alert: "You don't have permission to access
      this page."
4   end
5 end
```

You can also extract the action and subject which raised the error, if you want to handle authorization errors differently for some cases:

```
1   rescue_from "AccessGranted::AccessDenied" do |exception|
2     status = case exception.action
3       when :read # invocation like `authorize! :read, @something`
4         403
5       else
6         404
7     end
8
9     body = case exception.subject
10      when Post # invocation like `authorize! @some_action, Post`
11        "failed to access a post"
12      else
13        "failed to access something else"
14    end
15  end
```

You can also have a custom exception message while authorizing a request. This message will be associated with the exception object thrown.

```
1 class PostsController
2   def show
3     @post = Post.find(params[:id])
4     authorize! :read, @post, 'You do not have access to this post'
5     render json: { post: @post }
6   rescue AccessGranted::AccessDenied => e
7     render json: { error: e.message }, status: :forbidden
8   end
9 end
```

Checking permissions in controllers To check if the user has a permission to perform an action, use the `can?` and `cannot?` methods.

Example:

```
1 class UsersController
2   def update
3     # (...)
4
5     # only admins can elevate users to moderator status
6
7     if can? :make_moderator, @user
8       @user.moderator = params[:user][:moderator]
9     end
10
11    # (...)
12  end
13 end
```

Checking permissions in views Usually you don't want to show "Create" buttons for people who can't create something. You can hide any part of the page from users without permissions like this:

```
1 # app/views/categories/index.html.erb
2
3 <% if can? :create, Category %>
4   <%= link_to "Create new category", new_category_path %>
5 <% end %>
```

Customizing policy By default, AccessGranted adds this method to your controllers:

```
1 def current_policy
2   @current_policy ||= ::AccessPolicy.new(current_user)
3 end
```

If you have a different policy class or if your user is not stored in the `current_user` variable, then you can override it in any controller and modify the logic as you please.

You can even have different policies for different controllers!

Usage with pure Ruby

Initialize the Policy class:

```
1 policy = AccessPolicy.new(current_user)
```

Check the ability to do something:

with `can?`:

```
1 policy.can?(:create, Post) ==> true
2 policy.can?(:update, @post) ==> false
```

or with `cannot?`:

```
1 policy.cannot?(:create, Post) ==> false
2 policy.cannot?(:update, @post) ==> true
```

Common examples

Extracting roles to separate files

Let's say your app is getting bigger and more complex. This means your policy file is also getting longer.

Below you can see an extracted `:member` role:

```
1 class AccessPolicy
2   include AccessGranted::Policy
3
4   def configure
5     role :administrator, is_admin: true do
6       can :manage, User
7     end
8
9     role :member, MemberRole, -> { |user| !u.guest? }
10  end
11 end
```

And roles should look like this:

```
1 # app/roles/member_role.rb
2
3 class MemberRole < AccessGranted::Role
4   def configure
5     can :create, Post
6     can :destroy, Post do |post, user|
7       post.author == user
8     end
9   end
10 end
```

Compatibility with CanCan

This gem has been created as a replacement for CanCan and therefore it requires minimum work to switch.

Main differences

1. AccessGranted does not extend ActiveRecord in any way, so it does not have the `accessible_by ?` method which could be used for querying objects available to current user. This was very complex and only worked with permissions defined using hash conditions, so I decided to not implement this functionality as it was mostly ignored by CanCan users.
2. Both `can?/cannot?` and `authorize!` methods work in Rails controllers and views, just like in CanCan. The only change you have to make is to replace all `can? :manage, Class` with the exact action to check against. `can :manage` is still available for **defining** permissions and serves as a shortcut for defining `:create, :read, :update, :destroy` all in one line.
3. Syntax for defining permissions in the AccessPolicy file (Ability in CanCan) is exactly the same, with roles added on top. See Usage above.

Contributing

1. Fork it
2. Create your feature branch (`git checkout -b my-new-feature`)
3. Commit your changes (`git commit -am 'Add some feature'`)
4. Push to the branch (`git push origin my-new-feature`)
5. Create new pull request