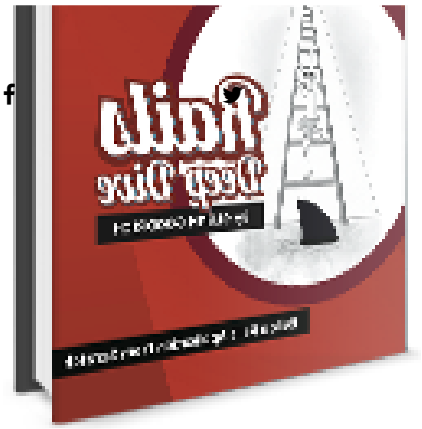
CanCanCan: The Rails Authorization Dance

Ilya Bodrov-Krukowski(http://www.sitepoint.com/author/ibodrov/) May 25, 2015 +4



Recently, I have written an overview of some popular authentication solutions for Rails (http://www.sitepoint.com/series/authentication-in-rails/). However, in many cases, having authentication by itself is not enough – you probably need an authorization mechanism to define access rules for various users. Is there an existing solution, preferably one that isn't very complex, but is still flexible?

Meet CanCanCan (https://github.com/CanCanCommunity/cancancan), a flexible authorization solution for Rails. This project started as CanCan (https://github.com/ryanb/cancan) authored by Ryan Bates, the creator of RailsCasts (http://railscasts.com). However, a couple of years ago this project became inactive, so members of the community decided to create CanCanCan, a continuation of the initial solution.

In this article, I'll integrate CanCanCan into a simple demo project, defining access rules, looking at possible options, and discussing how CanCanCan can reduce code duplication. After reading this post, you will have a strong understanding of CanCanCan's basic features and be ready to utilize it in real projects.

The source code can be found on GitHub (https://github.com/bodrovis/Sitepoint-source/tree/master/Authorization_with_CanCanCan).

A working demo is available at sitepoint-cancan.herokuapp.com (https://sitepoint-cancan.herokuapp.com).

Preparing Playground

Planning and Laying the Foundation

To start hacking on CanCanCan we have to prepare a playground for our experiments first. I am going to call my app iCan because I can (hee!):

```
$ rails new iCan -T
```

I am going to stick with Rails 4.1 but CanCanCan is compatible with Rails 3, as well.

The demo application will present users with a list of projects, both ongoing and finished. Users with different roles will have different level of access:

Guests won't have any access to the projects. They will only see the main page of the site.
Users will be able to see only the ongoing projects. They won't be able to modify or delete anything.
Moderators will have access to all projects with the ability to edit the ongoing ones.
Admins will have full access.

Our task will be to introduce those roles and define proper access rules for them.

I prefer to start with Bootstrap to style the app:

Gemfile

```
[...]
gem 'bootstrap-sass'
[...]
```

Run

```
$ bundle install
```

Set up the root route:

config/routes.rb

```
[...]
root to: 'pages#index'
[...]
```

Create a controller:

pages_controller.rb

```
class PagesController < ApplicationController
  def index
  end
end
```

and the view

views/pages/index.html.erb

```
<div class="page-header"><h1>Welcome!</h1></div>
```

Modify the layout to take advantage of Bootstrap's styles:

views/layouts/application.html.erb

```
[...]
<nav class="navbar navbar-inverse">
  <div class="container">
    <div class="navbar-header">
      <%= link_to 'iCan', root_path, class: 'navbar-brand' %>
    </div>
    <div id="navbar">
      <ul class="nav navbar-nav">
      </ul>
    </div>
  </div>
</nav>

<div class="container">
  <% flash.each do |key, value| %>
    <div class="alert alert-<%= key %>">
      <%= value %>
    </div>
  <% end %>
  <%= yield %>
</div>
[...]
```

Fake Authentication

Okay, so we've already briefly discussed roles to be added and their access levels, but first we need to introduce some kind of authentication. CanCanCan does not really care what authentication system you use. It only requires that a `current_user` method returning user's record or `nil` is present.

Recently, I've written a series of articles about authentication in Rails (http://www.sitepoint.com/series/authentication-in-rails/), so feel free to choose one of the solutions described there. For this demo, however, I will not use a real authentication to simplify things and focus on authorization only. What I will do, instead, is introduce a basic `User` class with a bunch of simple methods:

models/user.rb

```
class User
  ROLES = {0 => :guest, 1 => :user, 2 => :moderator, 3 => :admin}


  attr_reader :role

  def initialize(role_id = 0)
    @role = ROLES.has_key?(role_id.to_i) ? ROLES[role_id.to_i] : ROLES[0]
  end

  def role?(role_name)
    role == role_name
  end
end
```

Basically, there is a dictionary with all available roles. Upon initializing, assign the user one of the roles based on the provided ID. `role?` is just a conventional method that we'll use later.

Now let's define controller's action to set the role:

sessions_controller.rb

```
class SessionsController < ApplicationController
  def update
    id = params[:id].to_i
    session[:id] = User::ROLES.has_key?(id) ? id : 0
    flash[:success] = "Your new role #{User::ROLES[id]} was set!"
    redirect_to root_path
  end
end
```

Set up the route:

config/routes.rb

```
[...]
resources :sessions, only: [:update]
[...]
```

Add the links to choose the role:

views/layouts/application.html.erb

```
<nav class="navbar navbar-inverse">
  <div class="container">
    <div class="navbar-header">
      <%= link_to 'iCan', root_path, class: 'navbar-brand' %>
    </div>
    <div id="navbar">
      <ul class="nav navbar-nav">
      </ul>

      <ul class="nav navbar-nav pull-right">
        <li class="dropdown">
          <a class="dropdown-toggle" aria-expanded="false" role="button" data-toggle="dropdown" href="#"
            Role
            <span class="caret"></span>
          </a>
          <ul class="dropdown-menu" role="menu">
            <% User::ROLES.each do |k, v| %>
              <li>
                <%= link_to session_path(k), method: :patch do %>
                  <%= v %>
                  <% if v == current_user.role %>
                    <small class="text-muted">(current)</small>
                  <% end %>
                <% end %>
              </li>
            <% end %>
          </ul>
        </li>
      </ul>
    </div>
  </div>
</nav>
```

I am relying on the Bootstrap's dropdown widget here, so include it:

application.js

```
[...]
//= require bootstrap/dropdown
[...]
```

Also, if you are using Turbolinks, include `jquery-turbolinks` to bring default jQuery events back:

Gemfile

```
[...]
gem 'jquery-turbolinks'
[...]
```

application.js

```
[...]
//= require jquery.turbolinks
[...]
```

Lastly, introduce the `current_user` method:

application_controller.rb

```
[...]
private

def current_user
  User.new(session[:id])
end

helper_method :current_user
[...]
```

Great! Boot up the server and check that roles are being switched correctly.

Adding Projects

The last thing to do is to add the `Project` model and the corresponding controller. Each project will only have a title for now:

```
$ rails g model Project title:string
$ rake db:migrate
```

Controller:

projects_controller.rb

```
class ProjectsController < ApplicationController
  def index
    @projects = Project.all
  end
end
```

The routes:

config/routes.rb

```
[...]
resources :projects
[...]
```

And the view:

views/projects/index.html.erb

```
<div class="page-header"><h1>Projects</h1></div>

<% @projects.each do |project| %>
  <div class="well well-sm">
    <h2><%= project.title %></h2>
  </div>
<% end %>
```

Let's also utilize *seeds.rb* to add some demo records into the database:

db/seeds.rb

```
20.times {|i| Project.create!({title: "Project #{i + 1}"}) }
```

Run

```
$ rake db:seed
```

to populate your `projects` table.

Now the playground is ready and we can turn on the music and dance the CanCanCan.

Integrating CanCanCan and Defining Abilities

Drop CanCanCan into your Gemfile

Gemfile

```
[...]
gem 'cancancan', '~> 1.10'
[...]
```

and run

```
$ bundle install
```

Now we have to generate the *ability.rb* file that is going to host all our access rules:

```
$ rails g cancan:ability
```

Open up this file:

models/ability.rb

```
class Ability
  include CanCan::Ability

  def initialize(user)
  end
end
```

All your access rules (are belong to us….sorry) should be placed into the `initialize` method. There are some commented out examples to help you get started.

The `user` argument contains your `current_user`. Under the hoods `Ability` is being instantiated in the following way:

```
def current_ability
  @current_ability ||= Ability.new(current_user)
end
```

If, for example, you don't want to name this method `current_user` or you want to use another name for the `Ability` class, you can simply override the `current_ability` method in the `ApplicationController`.

Another option to renaming `current_user` is to introduce an alias method:

```
alias_method :current_user, :my_own_current_user
```

This way `current_ability` does not have to be redefined. Read more here (https://github.com/CanCanCommunity/cancancan/wiki/Changing-Defaults).

In our case `current_user` always returns a User object. In a real authentication scenario it will probably return `nil` if a user is not logged in. Therefore, it is a great idea to add a so called "nil guard":

models/ability.rb

```
[...]
def initialize(user)
  user ||= User.new
end
[...]
```

Now, introduce the first access rule saying that an admin has full access everywhere:

models/ability.rb

```
[...]
def initialize(user)
  user ||= User.new

  if user.role?(:admin)
    can :manage, :all
  end
end
[...]
```

`can` is the method to define abilities. `:manage` means “perform any action” and `:all` means basically “on everything”.

Checking Abilities

Let’s display a link on the main page leading to the list of projects and check if the user has the proper access:

views/pages/index.html.erb

```
<div class="page-header"><h1>Welcome!</h1></div>

<% if can? :index, Project %>
  <%= link_to 'Projects', projects_path, class: 'btn btn-lg btn-primary' %>
<% end %>
```

So `can?` is the method to check if the current user has the permission to perform an action. `:index` is the actual action’s name and `Project` is the class to on which to perform the action. You can also provide an object instead of a class (we will see an example on this later).

There is also the `cannot?` method that, as you’ve probably guessed, performs the opposite check of `can?`. Read more here (https://github.com/CanCanCommunity/cancancan/wiki/Checking-Abilities).

Unfortunately, nothing prevents the user from accessing the projects page directly (like “http://localhost:3000/projects”). Therefore, we have to enforce an authorization check inside the controller, as well. This is easy:

projects_controller.rb

```
[...]
def index
  @projects = Project.all
  authorize! :index, @project
end
[...]
```

Go ahead and try to access this page directly as a non-admin. The app will now raise an error, but that’s not very user-friendly. We have another problem to solve: How to rescue from an “access denied” error?

Rescuing from “Access Denied” Error

Rails provides us with a nice `rescue_from` method that we can call from the `ApplicationController`:

application_controller.rb

```
[...]
rescue_from CanCan::AccessDenied do |exception|
  flash[:warning] = exception.message
  redirect_to root_path
end
[...]
```

This way if `CanCan::AccessDenied` is raised in any of the child controllers, the error will be handled properly. Apart from `message`, an exception also responds to `action` (like `:index`) and `subject` (`Project`) methods.

You can manually raise an “Access Denied” error and provide your own message, action, and subject:

```
raise CanCan::AccessDenied.new("You are not authorized to perform this action!", :custom_action, Project
```

Give it a try! Read more about exception handling here (https://github.com/CanCanCommunity/cancancan/wiki/Exception-Handling).

Adding More Abilities

Let’s add a couple of other controller actions to create a new project and define who can do that:

projects_controller.rb

```
[...]
def new
  @project = Project.new
  authorize! :new, @project
end

def create
  @project = Project.new(project_params)
  if @project.save
    flash[:success] = 'Project was saved!'
    redirect_to root_path
  else
    render 'new'
  end
  authorize! :create, @project
end

private

def project_params
  params.require(:project).permit(:title)
end
[...]
```

The views:

views/projects/new.html.erb

```
<div class="page-header"><h1>New Project</h1></div>

<%= render 'form' %>
```

views/projects/_form.html.erb

```
<%= form_for @project do |f| %>
  <div class="form-group">
    <%= f.label :title %>
    <%= f.text_field :title, class: 'form-control' %>
  </div>

  <%= f.submit 'Post', class: 'btn btn-primary' %>
<% end %>
```

Add a new link to the top menu:

```
[...]
<ul class="nav navbar-nav">
  <% if can? :create, Project %>
    <li><%= link_to 'Add Project', new_project_path %></li>
  <% end %>
</ul>
[...]
```

As you can see, I am using `:create` instead of `:new` – if the user can create the record, they can access the “new record” page, as well.

Now add a couple more abilities:

models/ability.rb

```
def initialize(user)
  user ||= User.new

  if user.role?(:admin)
    can :manage, :all
  elsif user.role?(:moderator)
    can :create, Project
    can :read, Project
  elsif user.role?(:user)
    can :read, Project
  end
end
```

Wait, what does this `:read` action mean? What about `:index`? It appears, that CanCanCan introduces some action aliases by default:

```
alias_action :index, :show, :to => :read
alias_action :new, :to => :create
alias_action :edit, :to => :update
```

`:read` incorporates both `:index` and `:show`, whereas `:create` means being able to call `:new`, as well. That's really handy and you can easily define your own aliases using the same principle:

```
alias_action :update, :destroy, :to => :modify
```

Read more here (https://github.com/CanCanCommunity/cancancan/wiki/Action-Aliases).

Lastly let's deal with the `edit`, `update`, `destroy` actions:

projects_controller.rb

```
[...]
def edit
  @project = Project.find(params[:id])
  authorize! :edit, @project
end

def update
  @project = Project.find(params[:id])
  if @project.update_attributes(project_params)
    flash[:success] = 'Project was updated!'
    redirect_to root_path
  else
    render 'edit'
  end
  authorize! :update, @project
end

def destroy
  @project = Project.find(params[:id])
  if @project.destroy
    flash[:success] = 'Project was destroyed!'
  else
    flash[:warning] = 'Cannot destroy this project...'
  end
  redirect_to root_path
  authorize! :destroy, @project
end
[...]
```

The view:

edit.html.erb

```
<div class="page-header"><h1>Edit Project</h1></div>

<%= render 'form' %>
```

Now, add two buttons to edit and destroy a project:

views/projects/index.html.erb

```
[...]
<% @projects.each do |project| %>
  <div class="well well-sm">
    <h2><%= project.title %></h2>
    <% if can? :update, project %>
      <%= link_to 'Edit', edit_project_path(project), class: 'btn btn-info' %>
    <% end %>

    <% if can? :destroy, project %>
      <%= link_to 'Delete', project_path(project), class: 'btn btn-danger', method: :delete, data: {conf
    <% end %>
  </div>
<% end %>
```

I am passing the `project` object instead of a `Project` class – this way I can introduce more precise access rules. For example, I can add a rule saying the user can only edit a project that was added less than 2 hours ago.

On to the abilities:

models/ability.rb

```
if user.role?(:admin)
  can :manage, :all
elsif user.role?(:moderator)
  can [:create, :read, :update], Project
elsif user.role?(:user)
  can :read, Project
end
```

Notice that the `can` method accepts an array of actions as the first argument. Actually, the second argument can also be an array:

```
can [:create, :read, :update], [Project, Task]
```

We can rewrite this line

```
can [:create, :read, :update], Project
```

in another way by excluding some permissions:

```
can :manage, Project
cannot :destroy, Project
```

This means that a user can do everything with the projects, but cannot destroy them. Please notice that the order of lines **is important** here. If you place `cannot` before `can`, user will be able to perform **any action** on projects. You can read more about precedence here (https://github.com/CanCanCommunity/cancancan/wiki/Ability-Precedence).

Dealing with Code Duplication

Don't you think that calling `authorize!` in every controller's action is quite tedious? Moreover, what if you forget to include it in some method. CanCanCan handles this as well! Using `load_and_authorize_resource` helps you remove code duplication:

projects_controller.rb

```
class ProjectsController < ApplicationController
  load_and_authorize_resource
  [...]
end
```

Actually this method is composed of two method: `load_resource` and `authorize_resource`, each one being pretty self-explanatory. You can call them separately, if you like:

```
load_resource
authorize_resource
```

`load_resource` is going to load the record and `authorize_resource` is going to check if the user is authorized to perform an action (that equals to the current action's name) on that record. But how is the resource loaded for different actions?

> For `index`, the resource (assigned to the instance variable with a name in plural form) will be set to `Model.accessible_by(current_ability)`. `accessible_by` is a cool method that loads only the records that the current user can actually access (in our case, basic users can't view finished projects – we will introduce this scenario shortly).
> For `show`, `edit`, `update`, and `destroy`, the resource will simply be loaded using the `find` method: `Model.find(params[:id])`
> For `new` and `create`, the resource will be initialized with `new` method.
> For custom (non-CRUD) actions, the resource will be loaded using `find`, but this behavior can be modified.

So, our controller can be simplified like this:

projects_controller.rb

```
class ProjectsController < ApplicationController
  load_and_authorize_resource
  [...]

  def update
    if @project.update_attributes(project_params)
      flash[:success] = 'Project was updated!'
      redirect_to root_path
    else
      render 'edit'
    end
  end

  def create
    if @project.save
      flash[:success] = 'Project was saved!'
      redirect_to root_path
    else
      render 'new'
    end
  end

  def destroy
    if @project.destroy
      flash[:success] = 'Project was destroyed!'
    else
      flash[:warning] = 'Cannot destroy this project...'
    end
    redirect_to root_path
  end
end
```

What has changed?

I removed the `authorize!` method calls because `authorize_resource` does this job for us.
Second, I've removed the `index`, `new`, and `edit` actions completely, because they are handled by `load_resource`.
Third, I've removed the `@project = Project.find(params[:id])` line from the `update` and `destroy` actions as well as the `@project = Project.new(project_params)` from `create`, because once again `load_resource` takes care of this for us.

Yeah, I know what you are thinking. What about strong parameters? What about sorting and pagination? What if I want to skip loading and authorizing the resource for some actions? What if I need a custom finder? Those are great questions, let's discuss them one by one.

Strong params. When initializing a resource, CanCanCan checks if the controller responds to the following methods:

`create_params` or `update_params`. CanCanCan is going to use one of these methods to sanitize the input depending on the current action. This is cool, because you can define different sanitization rules for create and update.
If there is no `create_params` or `update_params` method defined, CanCanCan will search for `_params` method – this is what we are using in our demo (`project_params`).
Lastly, CanCanCan will search for a method with a static name `resource_params`.
You can also provide your own sanitizer method's name to override this default behavior:
`load_and_authorize_resource param_method: :my_sanitizer`.
If CanCanCan was not able to find any of these methods in the controller and a custom sanitizer is not set, it will initialize the resource as normal.

Override resource loading. For example, I want projects on the index page to be sorted by creation date, descending. You can do this easily:

projects_controller.rb

```
[...]
before_action :load_projects, only: :index
load_and_authorize_resource

[...]

private

def load_projects
  @projects = Project.accessible_by(current_ability).order('created_at DESC')
end

[...]
```

The idea is that `load_resource` will load a resource into the instance variable only if it hasn't been set yet. As long as I've added the `before_action` to set `@projects`, `load_resource` will not do anything for the `index` action. It is important to place `before_action` **before** `load_and_authorize_resource`.

Note that inside the `load_projects` I am using `accessible_by` to load only the records that the current user has rights to access.

Skipping loading and authorizing the resource. If for some reason you want to skip those actions, just write:

```
load_and_authorize_resource only: :index
# or
load_and_authorize_resource except: :index
```

Custom finders. As we've seen, `load_resource` uses the `find` method to load the resource. This is easy to change by providing `find_by` option:

```
load_resource find_by: :title
authorize_resource
```

CanCanCan is really flexible and you can easily override its default behavior! Full info can be found here (https://github.com/CanCanCommunity/cancancan/wiki/Authorizing-controller-actions).

Adding Conditions to Abilities

Now, let's say every project has the `ongoing` boolean attribute. Users should only be able to access only ongoing projects, whereas moderators can view all projects, but update only ongoing ones.

First of all, add a new migration:

```
$ rails g migration add_ongoing_to_projects ongoing:boolean
```

Modify the migration file:

xxx_add_ongoing_to_projects.rb

```
[...]
def change
  add_column :projects, :ongoing, :boolean, default: true
  add_index :projects, :ongoing
end
[...]
```

Apply the migration:

```
$ rake db:migrate
```

Don't forget to update the form:

views/projects/_form.html.erb

```
<%= form_for @project do |f| %>
  <div class="form-group">
    <%= f.label :title %>
    <%= f.text_field :title, class: 'form-control' %>
  </div>

  <div class="form-group">
    <%= f.label :ongoing %>
    <%= f.check_box :ongoing %>
  </div>

  <%= f.submit 'Post', class: 'btn btn-primary' %>
<% end %>
```

And modify the list of permitted attributes:

projects_controller.rb

```
[...]
def project_params
  params.require(:project).permit(:title, :ongoing)
end
[...]
```

Now the abilities:

models/ability.rb

```
[...]
if user.role?(:admin)
  can :manage, :all
elsif user.role?(:moderator)
  can :create, Project
  can :update, Project do |project|
    project.ongoing?
  end
  can :read, Project
elsif user.role?(:user)
  can :read, Project, ongoing: true
end
[...]
```

`can` accepts either a block or a hash of conditions to be more specific when defining rules. For the hash, it's important to only use table columns because those conditions will be used with the `accessible_by` method. Read more here (https://github.com/CanCanCommunity/cancancan/wiki/defining-abilities).

Now, switch to the `user` role and open the projects page. In the console, you should see output similar to this:

```
SELECT "projects".* FROM "projects" WHERE "projects"."ongoing" = 't' ORDER BY created_at DESC
```

This means that `accessible_by` is working correctly – it automatically loads only the resources that the user can access using the condition provided in *ability.rb*. Really cool.

Enforcing Authorization

If you want to check that authorization takes place in every controller, you can add `check_authorization` to the `ApplicationController`:

application_controller.rb

```
class ApplicationController < ActionController::Base
  check_authorization
  [...]
end
```

If authorization is not being performed in one of the actions, the `CanCan::AuthorizationNotPerformed` error will be raised. Still, we'll want to skip this check for some controllers, as any user should be able to access the main page and switch between roles. This is easy:

pages_controller.rb

```
class PagesController < ApplicationController
  skip_authorization_check
end
```

sessions_controller.rb

```
class SessionsController < ApplicationController
  skip_authorization_check
end
```

`skip_authorization_check` also accepts the `only` and `except` options. Moreover, `check_authorization` accepts `if` and `unless` options to define conditions when checking authorization should or should not take place, for example:

```
check_authorization if: :admin_subdomain?

private

def admin_subdomain?
  request.subdomain == "admin"
end
```

Read more here (https://github.com/CanCanCommunity/cancancan/wiki/Ensure-Authorization).

Conclusion

In this article we've discussed CanCanCan, a great authorization solution for Rails. I hope that now you are feeling confident to use it in your projects and implement more complex scenarios. I really encourage you to browse CanCanCan's wiki (https://github.com/CanCanCommunity/cancancan/wiki), as it has really useful examples.

Thank you for reading! As always, any reader is authorized to send his feedback on this article :). See you!

Tags:

Was this helpful?

Ilya Bodrov-Krukowski (http://www.sitepoint.com/author/ibodrov/)

(https://twitter.com/bodrovis) (https://plus.google.com/103641984440210150447) (http://ru.linkedin.com/pub/ilya-bodrov/93/8a6/603) (https://github.com/bodrovis)

Ilya Bodrov is a senior engineer working at Campaigner LLC, teaching assistant at Learnable and lecturer at Russian State Technological University (Internet Technology department). His primary programming languages are Ruby (with Rails) and JavaScript (AngularJS). He enjoys coding, teaching people and learning new things. Ilya also has some Cisco and Microsoft certificates and was working as a tutor in an educational center for a couple of years. In his free time he writes posts for his website (http://radiant-wind.com), participates in OpenSource projects, goes in for sports and plays music.

Join the discussion…

Steve Crozier · 5 months ago
"Meet CanCanCan, a flexible authentication solution for Rails." I think you mean, "...a flexible authorization solution...." It's a bummer that those two words sound so much alike.
1 · Reply · Share ›

Ilya Bodrov ↗ Steve Crozier · 5 months ago
Omg, I apologize. That is a typo and of course it should be authorization. Will fix that asap.
Reply · Share ›

James Hibbard SitePoint Staff ↗ Ilya Bodrov · 5 months ago
And fixed. Thanks for pointing that out.
Reply · Share ›

KvApril · 5 months ago
Good job thanks....
Reply · Share ›

Ilya Bodrov ↗ KvApril · 5 months ago
Glad you've liked it :)
Reply · Share ›

f.i. · 5 months ago
did you moved away from discourse? :)

Great Article! Thanks a lot for putting this together!

I am wondering if you could compare CanCanCan to pundit as my understanding was/is they do target the same thing.
Reply · Share ›

Ilya Bodrov ↗ f.i. · 5 months ago
Sitepoint implemented a new commenting system some time ago :)

Thank you! Yeah, I thought about pundit, probably I will add it to my pipeline!
Reply · Share ›

Sumit Bisht · 5 months ago
Hi Ilya,
thanks for sharing, was looking for a replacement for cancan and this is it!
Reply · Share ›

Ilya Bodrov ↗ Sumit Bisht · 5 months ago
Thank you!
Reply · Share ›

Habib · 5 months ago
"CanCanCan does not really care what authentication system you use."
"Meet CanCanCan, a flexible authentication solution for Rails"
Reply · Share ›

Ilya Bodrov ↗ Habib · 5 months ago
Yeah, I answered above. It's great that readers are so vigilant!
1 · Reply · Share ›

Next Article

The Pathway for New Railists → (http://www.sitepoint.com/the-pathway-for-new-railists/?utm_source=sitepoint&utm_medium=nextpost&utm_term=ruby)