Pundit provides a set of helpers which guide you in leveraging regular Ruby classes and object oriented design patterns to build a simple, robust and scaleable authorization system.

Links:

* [API documentation](http://www.rubydoc.info/gems/pundit)
* [Source Code](https://github.com/elabs/pundit)
* [Contributing](https://github.com/elabs/pundit/blob/master/CONTRIBUTING.md)
* [Code of Conduct](https://github.com/elabs/pundit/blob/master/CODE_OF_CONDUCT.md)

Sponsored by:

**Installation**

gem "pundit"

Include Pundit in your application controller:

class ApplicationController < ActionController::Base

include Pundit

protect\_from\_forgery

end

Optionally, you can run the generator, which will set up an application policy with some useful defaults for you:

rails g pundit:install

After generating your application policy, restart the Rails server so that Rails can pick up any classes in the new app/policies/ directory.

**Policies**

Pundit is focused around the notion of policy classes. We suggest that you put these classes in app/policies. This is a simple example that allows updating a post if the user is an admin, or if the post is unpublished:

class PostPolicy

attr\_reader :user, :post

def initialize(user, post)

@user = user

@post = post

end

def update?

user.admin? or not post.published?

end

end

As you can see, this is just a plain Ruby class. Pundit makes the following assumptions about this class:

* The class has the same name as some kind of model class, only suffixed with the word "Policy".
* The first argument is a user. In your controller, Pundit will call the current\_user method to retrieve what to send into this argument
* The second argument is some kind of model object, whose authorization you want to check. This does not need to be an ActiveRecord or even an ActiveModel object, it can be anything really.
* The class implements some kind of query method, in this case update?. Usually, this will map to the name of a particular controller action.

That's it really.

Usually you'll want to inherit from the application policy created by the generator, or set up your own base class to inherit from:

class PostPolicy < ApplicationPolicy

def update?

user.admin? or not record.published?

end

end

In the generated ApplicationPolicy, the model object is called record.

Supposing that you have an instance of class Post, Pundit now lets you do this in your controller:

def update

@post = Post.find(params[:id])

authorize @post

if @post.update(post\_params)

redirect\_to @post

else

render :edit

end

end

The authorize method automatically infers that Post will have a matching PostPolicy class, and instantiates this class, handing in the current user and the given record. It then infers from the action name, that it should call update? on this instance of the policy. In this case, you can imagine that authorize would have done something like this:

unless PostPolicy.new(current\_user, @post).update?

raise Pundit::NotAuthorizedError, "not allowed to update? this #{@post.inspect}"

end

You can pass a second argument to authorize if the name of the permission you want to check doesn't match the action name. For example:

def publish

@post = Post.find(params[:id])

authorize @post, :update?

@post.publish!

redirect\_to @post

end

If you don't have an instance for the first argument to authorize, then you can pass the class. For example:

Policy:

class PostPolicy < ApplicationPolicy

def admin\_list?

user.admin?

end

end

Controller:

def admin\_list

authorize Post # we don't have a particular post to authorize

# Rest of controller action

end

You can easily get a hold of an instance of the policy through the policy method in both the view and controller. This is especially useful for conditionally showing links or buttons in the view:

<% if policy(@post).update? %>

<%= link\_to "Edit post", edit\_post\_path(@post) %>

<% end %>

**Headless policies**

Given there is a policy without a corresponding model / ruby class, you can retrieve it by passing a symbol.

# app/policies/dashboard\_policy.rb

class DashboardPolicy < Struct.new(:user, :dashboard)

# ...

end

# In controllers

authorize :dashboard, :show?

# In views

<% if policy(:dashboard).show? %>

<%= link\_to 'Dashboard', dashboard\_path %>

<% end %>

**Scopes**

Often, you will want to have some kind of view listing records which a particular user has access to. When using Pundit, you are expected to define a class called a policy scope. It can look something like this:

class PostPolicy < ApplicationPolicy

class Scope

attr\_reader :user, :scope

def initialize(user, scope)

@user = user

@scope = scope

end

def resolve

if user.admin?

scope.all

else

scope.where(published: true)

end

end

end

def update?

user.admin? or not post.published?

end

end

Pundit makes the following assumptions about this class:

* The class has the name Scope and is nested under the policy class.
* The first argument is a user. In your controller, Pundit will call the current\_user method to retrieve what to send into this argument.
* The second argument is a scope of some kind on which to perform some kind of query. It will usually be an ActiveRecord class or a ActiveRecord::Relation, but it could be something else entirely.
* Instances of this class respond to the method resolve, which should return some kind of result which can be iterated over. For ActiveRecord classes, this would usually be an ActiveRecord::Relation.

You'll probably want to inherit from the application policy scope generated by the generator, or create your own base class to inherit from:

class PostPolicy < ApplicationPolicy

class Scope < Scope

def resolve

if user.admin?

scope.all

else

scope.where(published: true)

end

end

end

def update?

user.admin? or not post.published?

end

end

You can now use this class from your controller via the policy\_scope method:

def index

@posts = policy\_scope(Post)

end

Just as with your policy, this will automatically infer that you want to use the PostPolicy::Scope class, it will instantiate this class and call resolve on the instance. In this case it is a shortcut for doing:

def index

@posts = PostPolicy::Scope.new(current\_user, Post).resolve

end

You can, and are encouraged to, use this method in views:

<% policy\_scope(@user.posts).each do |post| %>

<p><%= link\_to post.title, post\_path(post) %></p>

<% end %>

**Ensuring policies and scopes are used**

When you are developing an application with Pundit it can be easy to forget to authorize some action. People are forgetful after all. Since Pundit encourages you to add the authorize call manually to each controller action, it's really easy to miss one.

Thankfully, Pundit has a handy feature which reminds you in case you forget. Pundit tracks whether you have called authorize anywhere in your controller action. Pundit also adds a method to your controllers called verify\_authorized. This method will raise an exception if authorize has not yet been called. You should run this method in an after\_action hook to ensure that you haven't forgotten to authorize the action. For example:

class ApplicationController < ActionController::Base

include Pundit

after\_action :verify\_authorized

end

Likewise, Pundit also adds verify\_policy\_scoped to your controller. This will raise an exception similar to verify\_authorized. However, it tracks if policy\_scope is used instead of authorize. This is mostly useful for controller actions like index which find collections with a scope and don't authorize individual instances.

class ApplicationController < ActionController::Base

include Pundit

after\_action :verify\_authorized, except: :index

after\_action :verify\_policy\_scoped, only: :index

end

**This verification mechanism only exists to aid you while developing your application, so you don't forget to call authorize. It is not some kind of failsafe mechanism or authorization mechanism. You should be able to remove these filters without affecting how your app works in any way.**

Some people have found this feature confusing, while many others find it extremely helpful. If you fall into the category of people who find it confusing then you do not need to use it. Pundit will work just fine without using verify\_authorized and verify\_policy\_scoped.

**Conditional verification**

If you're using verify\_authorized in your controllers but need to conditionally bypass verification, you can use skip\_authorization. For bypassing verify\_policy\_scoped, use skip\_policy\_scope. These are useful in circumstances where you don't want to disable verification for the entire action, but have some cases where you intend to not authorize.

def show

record = Record.find\_by(attribute: "value")

if record.present?

authorize record

else

skip\_authorization

end

end

**Manually specifying policy classes**

Sometimes you might want to explicitly declare which policy to use for a given class, instead of letting Pundit infer it. This can be done like so:

class Post

def self.policy\_class

PostablePolicy

end

end

**Just plain old Ruby**

As you can see, Pundit doesn't do anything you couldn't have easily done yourself. It's a very small library, it just provides a few neat helpers. Together these give you the power of building a well structured, fully working authorization system without using any special DSLs or funky syntax or anything.

Remember that all of the policy and scope classes are just plain Ruby classes, which means you can use the same mechanisms you always use to DRY things up. Encapsulate a set of permissions into a module and include them in multiple policies. Use alias\_method to make some permissions behave the same as others. Inherit from a base set of permissions. Use metaprogramming if you really have to.

**Generator**

Use the supplied generator to generate policies:

rails g pundit:policy post

**Closed systems**

In many applications, only logged in users are really able to do anything. If you're building such a system, it can be kind of cumbersome to check that the user in a policy isn't nil for every single permission.

We suggest that you define a filter that redirects unauthenticated users to the login page. As a secondary defence, if you've defined an ApplicationPolicy, it might be a good idea to raise an exception if somehow an unauthenticated user got through. This way you can fail more gracefully.

class ApplicationPolicy

def initialize(user, record)

raise Pundit::NotAuthorizedError, "must be logged in" unless user

@user = user

@record = record

end

end

**Rescuing a denied Authorization in Rails**

Pundit raises a Pundit::NotAuthorizedError you can [rescue\_from](http://guides.rubyonrails.org/action_controller_overview.html#rescue-from) in your ApplicationController. You can customize the user\_not\_authorized method in every controller.

class ApplicationController < ActionController::Base

protect\_from\_forgery

include Pundit

rescue\_from Pundit::NotAuthorizedError, with: :user\_not\_authorized

private

def user\_not\_authorized

flash[:alert] = "You are not authorized to perform this action."

redirect\_to(request.referrer || root\_path)

end

end

Alternatively, you can globally handle Pundit::NotAuthorizedError's by having rails handle them as a 403 error and serving a 403 error page. Add the following to application.rb:

config.action\_dispatch.rescue\_responses["Pundit::NotAuthorizedError"] = :forbidden

**Creating custom error messages**

NotAuthorizedErrors provide information on what query (e.g. :create?), what record (e.g. an instance of Post), and what policy (e.g. an instance of PostPolicy) caused the error to be raised.

One way to use these query, record, and policy properties is to connect them with I18n to generate error messages. Here's how you might go about doing that.

class ApplicationController < ActionController::Base

rescue\_from Pundit::NotAuthorizedError, with: :user\_not\_authorized

private

def user\_not\_authorized(exception)

policy\_name = exception.policy.class.to\_s.underscore

flash[:error] = t "#{policy\_name}.#{exception.query}", scope: "pundit", default: :default

redirect\_to(request.referrer || root\_path)

end

end

en:

pundit:

default: 'You cannot perform this action.'

post\_policy:

update?: 'You cannot edit this post!'

create?: 'You cannot create posts!'

Of course, this is just an example. Pundit is agnostic as to how you implement your error messaging.

**Manually retrieving policies and scopes**

Sometimes you want to retrieve a policy for a record outside the controller or view. For example when you delegate permissions from one policy to another.

You can easily retrieve policies and scopes like this:

Pundit.policy!(user, post)

Pundit.policy(user, post)

Pundit.policy\_scope!(user, Post)

Pundit.policy\_scope(user, Post)

The bang methods will raise an exception if the policy does not exist, whereas those without the bang will return nil.

**Customize Pundit user**

In some cases your controller might not have access to current\_user, or your current\_user is not the method that should be invoked by Pundit. Simply define a method in your controller called pundit\_user.

def pundit\_user

User.find\_by\_other\_means

end

**Additional context**

Pundit strongly encourages you to model your application in such a way that the only context you need for authorization is a user object and a domain model that you want to check authorization for. If you find yourself needing more context than that, consider whether you are authorizing the right domain model, maybe another domain model (or a wrapper around multiple domain models) can provide the context you need.

Pundit does not allow you to pass additional arguments to policies for precisely this reason.

However, in very rare cases, you might need to authorize based on more context than just the currently authenticated user. Suppose for example that authorization is dependent on IP address in addition to the authenticated user. In that case, one option is to create a special class which wraps up both user and IP and passes it to the policy.

class UserContext

attr\_reader :user, :ip

def initialize(user, ip)

@user = user

@ip = ip

end

end

class ApplicationController

include Pundit

def pundit\_user

UserContext.new(current\_user, request.ip)

end

end

**Strong parameters**

In Rails 4 (or Rails 3.2 with the [strong\_parameters](https://github.com/rails/strong_parameters) gem), mass-assignment protection is handled in the controller. With Pundit you can control which attributes a user has access to update via your policies. You can set up a permitted\_attributesmethod in your policy like this:

# app/policies/post\_policy.rb

class PostPolicy < ApplicationPolicy

def permitted\_attributes

if user.admin? || user.owner\_of?(post)

[:title, :body, :tag\_list]

else

[:tag\_list]

end

end

end

You can now retrieve these attributes from the policy:

# app/controllers/posts\_controller.rb

class PostsController < ApplicationController

def update

@post = Post.find(params[:id])

if @post.update\_attributes(post\_params)

redirect\_to @post

else

render :edit

end

end

private

def post\_params

params.require(:post).permit(policy(@post).permitted\_attributes)

end

end

However, this is a bit cumbersome, so Pundit provides a convenient helper method:

# app/controllers/posts\_controller.rb

class PostsController < ApplicationController

def update

@post = Post.find(params[:id])

if @post.update\_attributes(permitted\_attributes(@post))

redirect\_to @post

else

render :edit

end

end

end

If you want to permit different attributes based on the current action, you can define a permitted\_attributes\_for\_#{action}method on your policy:

# app/policies/post\_policy.rb

class PostPolicy < ApplicationPolicy

def permitted\_attributes\_for\_create

[:title, :body]

end

def permitted\_attributes\_for\_edit

[:body]

end

end

If you have defined an action-specific method on your policy for the current action, the permitted\_attributes helper will call it instead of calling permitted\_attributes on your controller.

**RSpec**

**Policy Specs**

Pundit includes a mini-DSL for writing expressive tests for your policies in RSpec. Require pundit/rspec in your spec\_helper.rb:

require "pundit/rspec"

Then put your policy specs in spec/policies, and make them look somewhat like this:

describe PostPolicy do

subject { described\_class }

permissions :update?, :edit? do

it "denies access if post is published" do

expect(subject).not\_to permit(User.new(admin: false), Post.new(published: true))

end

it "grants access if post is published and user is an admin" do

expect(subject).to permit(User.new(admin: true), Post.new(published: true))

end

it "grants access if post is unpublished" do

expect(subject).to permit(User.new(admin: false), Post.new(published: false))

end

end

end

An alternative approach to Pundit policy specs is scoping them to a user context as outlined in this [excellent post](http://thunderboltlabs.com/blog/2013/03/27/testing-pundit-policies-with-rspec/) and implemented in the third party [pundit-matchers](https://github.com/chrisalley/pundit-matchers) gem.

**External Resources**

* [RailsApps Example Application: Pundit and Devise](https://github.com/RailsApps/rails-devise-pundit)
* [Migrating to Pundit from CanCan](http://blog.carbonfive.com/2013/10/21/migrating-to-pundit-from-cancan/)
* [Testing Pundit Policies with RSpec](http://thunderboltlabs.com/blog/2013/03/27/testing-pundit-policies-with-rspec/)
* [Using Pundit outside of a Rails controller](https://github.com/elabs/pundit/pull/136)
* [Straightforward Rails Authorization with Pundit](http://www.sitepoint.com/straightforward-rails-authorization-with-pundit/)

**License**

Licensed under the MIT license, see the separate LICENSE.txt file.