Rails-Devise-Pundit

Last updated 28 Jun 2014

GitHub Repository (https://github.com/RailsApps/rails-devise-pundit) · Issues (https://github.com/RailsApps/rails-devise-pundit/issues)

Pundit Quickstart Guide

Introduction

This guide shows how to use Pundit for authorization in a Rails application. Versions:

- Pundit 0.2
- Devise 3.2
- Rails 4.1

Pundit (https://github.com/elabs/pundit) is used for *authorization*, limiting access to pages of a web application. We use Devise (https://github.com/plataformatec/devise) for *authentication*, to verify a user's registered identity.

What's Here

You'll find what you need:

- build a starter application with Devise and Pundit
- add role-based authorization to a User model
- control access using Pundit

- add tests for Pundit

Is It for You?

This guide shows how to use Pundit in a web application. There are no in-depth tutorials about Pundit available elsewhere, so experienced Rails developers as well as beginners will find this guide helpful. If you've created simple Rails applications, such as the one in the book Learn Ruby on Rails (https://tutorials.railsapps.org/learn-ruby-on-rails), you'll have enough background to follow this guide. Before you begin, if you'd like to know more about Devise, see the Devise Quickstart Guide (https://tutorials.railsapps.org/tutorials/devise-quickstart).

Concepts

Almost every web application needs an authorization system, if there are parts of the web site that are restricted to some users. Most web sites set access restrictions based on roles; that is, users are grouped by privilege. The web application checks the user's role to determine if access is allowed. We call this *role-based authorization*.

In this tutorial we'll look at how to set up roles in a User model. We'll look at how to implement simple role-based authorization without Pundit, and then we'll look at using Pundit for role-based authorization. We'll also briefly discuss complex applications where role-based authorization is inadequate.

Role-based authorization can be implemented with simple conditional `if` statements in Rails controllers or views. In the simplest implementation, we check if a user has a specific role (such as administrator) and either allow access or redirect with an "Access Denied" message. As a framework, Rails allows you to add as much complexity to a controller as you wish. However, the Rails community has come to a consensus that complexity in controllers is a not a best practice; you'll often hear the advice, "Keep your controllers skinny." Given that advice, some developers attempt to implement access rules as methods in a model. Access rules don't belong in a model, given that a model is best used for retrieving and manipulating data, and not for logic that controls program flow through the application. Rather than build "fat models" with complex access rules for program flow, developers look for ways to keep both models and controllers free from authorization code.

Two approaches stand out. The gem CanCan (https://github.com/ryanb/cancan) and its successor CanCanCan (https://github.com/CanCanCommunity/cancancan) are popular, as is a newer gem, Pundit (https://github.com/elabs/pundit). CanCan provides a DSL (domain-specific language) that isolates all authorization logic in a single Ability class. As an application grows in complexity, the CanCan Ability class can grow unwieldy. Also, every authorization request requires evaluation of the full CanCan Ability class, adding performance overhead. Pundit is an authorization system that uses simple Ruby objects for access rules instead of a centralized Ability class. Pundit provides two helper methods and a set of conventions instead of a DSL. Overall, Pundit is simpler than CanCan but also offers the advantage of segregating access rules into a central location. With Pundit the central location is not a single Ability file. Instead, it is a folder named app/policies/ containing plain Ruby objects that implement access rules. Adding authorization to a controller action requires one line of code that calls a helper method. Pundit uses meta-programming magic to instantiate a policy object that matches an access rule to the controller action. Pundit policy objects are lightweight, adding authorization logic without as much overhead as CanCan.

Pundit is well-suited to the service-oriented architecture that is popular for large Rails applications, emphasizing object-oriented design with discrete Ruby objects providing specialized services. If you wish, you can create a single Pundit policy object for use with all your controllers. More often, developers create a policy object that corresponds with a specific model (for example, a UserPolicy class to match a User model) and will use the policy object to control access for actions in a specific controller (for example, a Users controller).

Pundit policy objects are often described as POROs, or "plain old Ruby objects." PORO simply means that a Pundit policy object doesn't inherit from other classes or include code mixed in from elsewhere. In contrast, a Rails model that inherits from Active Record is not a PORO; the model inherits behavior that is defined in a parent class. Because Pundit policy objects are POROs, the code is simple and easy to understand.

Starter App with Devise and Pundit

In this tutorial, we'll build the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) example application using the Rails Composer (http://railsapps.github.io/rails-composer/) tool. The Rails Composer tool creates a complete starter app including:

- Pundit for authorization
- Devise for authentication
- Bootstrap or Foundation front-end framework
- RSpec testing framework

The starter application will give you sign-up and sign-in pages nicely styled for Bootstrap or Foundation, plus a full test suite for your authentication and user management features, and examples of authorization with RSpec tests. Using Rails Composer, your starter app will include all this without extra work.

In this tutorial, we'll examine the code generated by Rails Composer. You'll see how to customize the application for your needs.

Building the Application

See the article Installing Rails (http://railsapps.github.io/installing-rails.html) for instructions about setting up Rails and your development environment. Rails 4.1 must be installed in your development environment.

To build the starter app, run the command:

```
$ rails new rails-devise-pundit -m https://raw.github.com/RailsApps/rails-composer/master/compo
ser.rb
```

You'll see a prompt:

```
question  Build a starter application?
      1)  Build a RailsApps example application
      2)  Contributed applications
      3)  Custom application
```

Enter "1" to select Build a RailsApps example application. You'll see a prompt:

```
question  Starter apps for Rails 4.1. More to come.
      1)  learn-rails
      2)  rails-bootstrap
      3)  rails-foundation
      4)  rails-omniauth
      5)  rails-devise
      6)  rails-devise-pundit
```

Choose rails-devise-pundit.

Choose these options to create the starter application for this tutorial:

- Web server for development? WEBrick (default)
- Web server for production? Same as development
- Database used in development? SQLite
- Template engine? ERB
- Test framework? RSpec with Capybara
- Continuous testing? None
- Front-end framework? Bootstrap 3.0
- Add support for sending email? Gmail
- Devise modules? Devise with default modules
- Use a form builder gem? None
- Set a robots.txt file to ban spiders? no
- Create a GitHub repository? no
- Use or create a project-specific rvm gemset? yes

If you have problems creating the starter app, ask for help on Stack Overflow (http://stackoverflow.com/questions/tagged/railsapps). Use the tag “railsapps” on Stack Overflow for extra attention. If you find a problem with the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) example application, open an issue on GitHub (https://github.com/RailsApps/rails-devise-pundit/issues).

Running the Application

After you create the application, switch to its folder to work directly in the application:

```
$ cd rails-devise-pundit
```

Launch the application by entering the command:

```
$ rails server
```

To see your application in action, open a web browser window and navigate to http://localhost:3000/ (http://localhost:3000).

Exploring the Application

You've got a starter application with a home page, a navigation bar, and the functionality needed to sign in and see a list of users.

The script that set up the starter app created the database and seeded it with an example user. You'll see a link on the home page for "Users: 1 registered." If you click the link, you'll see a flash message, "You need to sign in or sign up before continuing." The starter app is set up so you must be *authenticated* before you can access anything other than the home page.

To explore the application, sign in as an administrator with the email address `user@example.com` and the password `changeme`. You'll see a flash message, "Welcome! You have signed up successfully."

Click the link "Users" and you'll see a list of users. There is only one user listed right now, the administrator. If you want, you can click the link to "Edit Account" and change the name, email address, or password for the administrator.

Try signing out and creating another account. Click the "Sign up" link and enter a name, email address, and password to register as a new user. After you click the "Sign up" button, you'll see the home page with the message "Welcome! You have signed up successfully." You'll see two users are registered. Click the link "Users" to try to see the list of users. You will see "Access denied" because your new account does not have the privileges of an administrator.

You've seen a simple demonstration of role-based authorization. Pundit provides a mechanism to set rules for access. In this application, each user is assigned a role and Pundit checks the user's role to determine if access will be granted.

Let's consider which features come from Devise, and which are additional, using Pundit.

Devise provides basic features of authentication and user management. To learn how these features are implemented, and ways to customize them, see the Devise Quickstart Guide (https://tutorials.railsapps.org/tutorials/devise-quickstart) on the RailsApps site:

- Sign up (create account)
- Sign in
- "Forgot password?" (send password reset instructions)
- "Remember me" (stay signed in for two weeks)
- View account (also called "user profile")
- Edit account (change name, email, or password)
- Delete account

Rails Composer added additional features:

- User profile page
- A signed-in user can view, edit, or delete his or her own profile (but not others')

Finally, Rails Composer set up Pundit to control access:

- Only an administrator can see a page with a list of users
- Only an administrator can see any user's profile
- Only an administrator can delete any user

- Only an administrator can change a user's role

We'll examine these features in this guide.

Set Up Pundit

We've already built our rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) example application using Rails Composer, but before we examine the application, I'll describe how Rails Composer modifies a starter app to set up Pundit. If you don't use Rails Composer, you can follow these steps to install Pundit in another application. For now, read through the steps (while appreciating the work is done for you).

Authorization Elements

To add role-based authorization features to an application, you'll need to:

- add the Pundit gem
- add a Pundit initializer
- add a roles attribute to the User model
- use the `authorize` helper method to add restrictions to any controller action
- set up *policy objects* with access rules

Though Rails Composer has set this up already, we'll look at each element so you'll see what is required for Pundit.

Add Pundit to the Gemfile

Rails Composer adds Pundit to the basic Gemfile and then runs `bundle install`:

```
source 'https://rubygems.org'
.
.
.
gem 'pundit'
.
.
.
```

Pundit Generator

If you looked at the Pundit README (https://github.com/elabs/pundit), you may have noticed that the Pundit gem includes a generator that creates an example file. The generator is not very useful and you can ignore it. It creates one file `app/policies/application_policy.rb` that merely serves as an example.

Application Controller

The Pundit README (https://github.com/elabs/pundit) shows how you can modify the Application controller to incorporate Pundit in every controller. The README example also shows how to add code to display a message when access is not authorized. To minimize clutter in the Application controller, we'll use

a different technique to set up Pundit.

We won't make any changes to the default app/controllers/application_controller.rb file:

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
end
```

We'll use an initializer file instead.

Pundit Initializer

We need a way to add Pundit to every controller. Some developers like to add it to the Application controller file, because every controller that inherits from the Application controller will inherit Pundit. Instead, to segregate our Pundit-specific code, we will use an initializer file.

Here is the file config/initializers/pundit.rb generated by Rails Composer:

```
# config/initializers/pundit.rb
# Extends the ApplicationController to add Pundit for authorization.
# Modify this file to change the behavior of a 'not authorized' error.
# Be sure to restart your server when you modify this file.
module PunditHelper
  extend ActiveSupport::Concern

  included do
    include Pundit
    rescue_from Pundit::NotAuthorizedError, with: :user_not_authorized
  end

  private

  def user_not_authorized
    flash[:alert] = "Access denied."
    redirect_to (request.referrer || root_path)
  end

end

ApplicationController.send :include, PunditHelper
```

The statement `include Pundit` makes Pundit methods available to any controller that inherits from the Application controller. With `include Pundit`, we get an `authorize` helper method we can add to any controller action. You'll use the `authorize` helper method with any controller action that requires access control.

The statement that begins `rescue_from Pundit::NotAuthorizedError...` lets us define a user-friendly message to display when a user is not allowed to see a page. In development, without the `rescue_from Pundit::NotAuthorizedError...`, you will see an exception and stack trace. In production, the user will see a generic "something went wrong" page. Instead, we can provide a custom error message.

The `user_not_authorized` method, which is called when Pundit raises a `NotAuthorizedError`, gives us a place to define an error message. We simply display a flash message, "Access denied." You can modify the initializer file to change the error message. Later, in a chapter on "Custom Violation Messages," we'll take a closer look at customizing the error messages.

The rest of the file implements a Rails *concern*. Concerns are modules that can be mixed into models and controllers to add shared code. Typically, these modules go in a app/controllers/concerns/ folder and are added to a controller with an `include` keyword. In this case, we use the Ruby `send` method to add our mixin to the `ApplicationController` object, adding `include PunditHelper` without actually editing the Application controller code. If this seems incomprehensible, don't worry, as it is advanced Ruby metaprogramming that most beginners don't encounter.

Role-Based Authorization

Pundit can set access limits based on any criteria. For example, using only the `created_at` attribute found in the User model, you could create a rule that certain pages on your web site are only seen by users who registered more than a month ago. Or you could create a rule that checks the user's email address and only allows users with a `gmail.com` address to comment on a blog. Chances are you don't need to implement such odd rules.

More often, developers want to establish access privileges based on a user's role. For example, an administrator can view and delete any user's account. Or a user who has paid for a premium plan can access premium features. Roles are attributes associated with a user account, and often implemented in a User model. We'll look at ways to implement roles, but first let's consider situations where role-based authorization is not suitable.

Alternatives to Role-Based Authorization

Role-based authorization is suitable for simple applications without complex access rules. A big advantage is easy conceptualization; it is easy to imagine personas, each with different (but uniform) privileges. If all you need are role-based rules, use them.

You may encounter complex applications where role-based authorization is inadequate. In these cases, authorization is often based on matching requested activities with a database of privileges. For example, imagine an application that is used across a university to record and report student grades. A student can see his or her own grades for any class; a teaching assistant can enter a grade but not change it after the course ends but only for students in their own section; a professor can enter or change a grade for any student in the class until the next semester begins; the department chairperson can view but not change grades for any student enrolled in a department course; the registrar of records can view or change any grade for any student ever enrolled. Whew! In a real university, the requirements are even more complex, I'm sure. Not only do

roles overlap (a professor may also be a department chairperson) but privileges are finer-grained than roles. A user with the role of professor should only have grade-changing privileges for the students in his or her course and it is impractical to create a new role for every new course every semester. This is a use case for building access rules based on permissions attached to activities, not roles.

If you're building an application with this level of complexity, seek help from experienced developers. You'll be treading in the territory of large enterprises where initiatives such as User-Managed Access (http://en.wikipedia.org/wiki/User-Managed_Access) hold sway. Pundit can be used for these cases but this tutorial doesn't cover such complexity. We'll focus on the majority of applications where role-based authorization is optimal.

Implementing Roles

Despite the emphasis in Rails on conventions for the sake of consistency, there is no convention established in the framework for a "user". Developers are free to create a model for an Account, a Member, a Profile, or anything else that meets their requirements. In most cases, developers create a User model, which is a practice we follow in the RailsApps example applications and tutorials.

There is also no convention for implementing roles. We'll consider several approaches you may see in other applications, and then look closely at the approach we've selected for this application.

Single or Multiple Roles

Before you decide which approach you'll use to implement roles, consider whether your users will each have a single role, or if you will need to assign multiple roles to a single user. Let's consider some examples.

If a user can either be an ordinary user, or an administrator, and nothing else, each user has a single role. If a user can join with a bronze, silver, or gold plan, or be an administrator, you'll only need one role per user. With a single role per user, privileges can be cumulative. You can create access rules so gold users get all the privileges of bronze or silver users (plus more). A single role per user is all you need for these web applications.

If privileges are not cumulative, you may need multiple roles per user. Consider concertgoers. Some may pay extra to sit close to the stage; there's a "seating" role with values of "close" or "far." Some may win a contest for a backstage pass; whether they sit close or far, each user will be assigned a value of "access" or "no access" for the "backstage" role. Each user will have multiple roles. Take the time to map out your system of privileges before deciding how you'll implement roles.

Binary Role

The simplest use case is a user who can be either an administrator or an ordinary user. You can add a boolean attribute to the User model to indicate whether a user is an administrator or not. Then you can check the role with a simple method such as `@user.admin?`.

This approach has drawbacks when you need multiple roles. You could add another boolean attribute to indicate if a user has a premium plan, but as soon as you add more plans, the approach gets unwieldy, as you'll need to add a separate attribute to the model for every anticipated authorization level.

String Roles

For a user who can have only a single role, you could add a `role` attribute to the User model, and set a string representing a privilege level such as “admin,” “gold,” “silver,” or “bronze.” This approach requires only one column in the User data table. You may need some supporting code in the User model that makes sure only pre-defined roles can be used. You can check the role with `if @user.role == 'admin'` ; with a little extra code, you can implement methods in the User model such as `@user.is_admin?` .

You may encounter several limitations with this approach. First, though you can easily add new roles, you can’t easily rename roles once you’ve got registered users (you’d have to change many records in the database). Second, a user cannot have multiple roles; only one role is possible for each user. Finally, this approach requires extra code in the model to implement convenience methods such as `@user.is_admin?` .

Bitmask Roles

A model attribute can encode a role using a bitmask (http://en.wikipedia.org/wiki/Mask_(computing)) function. Instead of representing the role as a string, the role attribute can take an integer value. The integer itself means nothing; instead, by decoding the integer as a binary number, each bit represents a role. This is more compact than creating a separate database column for each role.

In the early days of computing, when machine memory was limited and code had to be compact, bitmasks were commonly used to store configuration settings or other data. Today bitmasks are a clever trick that is best avoided. To implement bitmasks to encode roles, you’ll have to add complex methods to your User model (or use the bitmask_attributes (https://github.com/joelmoss/bitmask_attributes) gem). There is also no way to set up database indexes or simple queries to retrieve encoded roles. It’s the worst kind of hack; there is no performance benefit and your code becomes much less readable. There are better alternatives.

Role Model

If your application requires that users have more than one role, a Role model provides the most flexible implementation. The User model and Role model will have a many-to-many association, so a user can have multiple roles and a role can be assigned to multiple users. You’ll need two database tables, one for the User model and another for the Role model. Additionally, to implement the many-to-many association, your database will need an intermediate “join” table named roles_users. Your models will implement the association using the has_and_belongs_to_many (http://guides.rubyonrails.org/association_basics.html#the-has-and-belongs-to-many-association) association:

An alternative approach uses the has_many :through (http://guides.rubyonrails.org/association_basics.html#the-has-many-through-association). This requires an intermediate class such as Assignment. Unless you need to interact with the intermediate object, has_and_belongs_to_many is more appropriate.

With a Role model, each user can be assigned multiple roles. You’ll be able to construct access rules with conditions such as `if @user.roles.include?('admin')` . If you want convenience methods such as `@user.is_admin?` , you’ll need extra code in the User model.

I won’t show you the code you need to implement roles using the Role model because there is a gem that provides this functionality. You don’t have to implement it yourself.

Rolify Gem

Use Florent Monbillard's Rolify (https://github.com/EppO/rolify) gem to add multiple roles to an application. The gem is well-documented on its wiki (https://github.com/EppO/rolify/wiki). The Rolify gem provides a generator that creates a Role model with a migration to create a roles table, a users_roles join table, and appropriate database indexes. To add roles to the User model, simply add a `rolify` method to the model.

Rolify provides a full set of convenience methods without any extra coding, so you can use methods such as:

- `user.add_role :admin` – to assign an admin role to a user
- `user.has_role? :admin` – to determine if a user is an administrator
- `user.remove_role :admin` – to remove a role
- `@users = User.with_role :admin` – obtain an array of all users with the admin role

Rolify is convenient and well-tested, so there is little reason to implement your own Role model, if your application requires users with more than one role.

Enum Roles

Most applications don't need to assign more than one role to a user. In our tutorial application, a single role for each user is sufficient, so we won't use the Rolify gem. Instead we'll use a new feature of Active Record, Enum (http://api.rubyonrails.org/classes/ActiveRecord/Enum.html), introduced in Rails 4.1. Enums are the simplest way to add roles to a User model, with advantages over all the approaches described above.

An enum, or enumerated type (http://en.wikipedia.org/wiki/Enumerated_type), is stored in the database as an integer but represented in code as a string. Enums give us all the functionality we need to implement user roles. We can define the names of the roles, and if necessary, change the names as needed (the integer values stored with each user record remain unchanged). Active Record will restrict the assignment of the attribute to a collection of predefined values, so we don't have to add any code to restrict the names of the roles to a defined set. Best of all, enums come with a set of convenience methods that allow us to directly query the role without any extra code. For an enum attribute named `role`, with the values `admin`, `vip`, and `user`, we can use these methods:

- `User.roles # => {"user"=>0, "vip"=>1, "admin"=>2}` – list all roles
- `user.admin!` – make the user an administrator
- `user.admin? # => true` – query if the user is an administrator
- `user.role # => "admin"` – find out the user's role
- `@users = User.admin` – obtain an array of all users with the admin role
- `user.role = 'foo' # ArgumentError: 'foo' is not a valid role` – we can't set invalid roles

Combined with Pundit, Active Record enums make it easy to add role-based authorization to a Rails application.

Selecting Roles in Forms

Before we examine how Rails Composer sets up a User model using enums for roles, let's explore how to set roles in forms. Strictly speaking, this has nothing to do with authorization, but if your application requires users to have roles, you may need forms that allow users or administrators to change roles.

Our starter application already contains an example of a form that allows an administrator to change a user's role.

Examine the file app/views/users/_user.html.erb:

```
<td>
  <%= link_to user.email, user %>
</td>
<td>
  <%= form_for(user) do |f| %>
    <%= f.select(:role, User.roles.keys.map {|role| [role.titleize,role]}) %>
    <%= f.submit 'Change Role', :class => 'button-xs' %>
  <% end %>
</td>
<td>
  <%= link_to("Delete user", user_path(user), :data => { :confirm => "Are you sure?" }, :method
 => :delete, :class => 'button-xs') unless user == current_user %>
</td>
```

This file is a partial that is included in the User index page. The page can only be viewed by an administrator (we'll see why when we examine the Users controller later in this tutorial). The partial displays cells in a table, one row for each user. The first cell shows the user's email address. The second cell contains a form that displays the user's role and allows the administrator to select a new one. The third cell contains a link (styled as a button) that allows the administrator to delete the user (the delete button will not appear for the signed-in user).

See RailsGuide: Action View Form Helpers (http://guides.rubyonrails.org/form_helpers.html#select-boxes-for-dealing-with-models) for an introduction to the `select` form helper. In this example, we're using `form_for(user)` to construct a form scoped to the User model. When used with a model, the syntax for the `select` form helper looks like this:

```
<%= f.select(attribute, [[key, value], [key, value], ...]) %>
```

The `select` form helper creates a dropdown menu for displaying a choice of roles. We could hardcode an array of key-value pairs like this:

```
<%= f.select(:role, [['User', 'user'], ['Vip', 'vip'], ['Admin', 'admin']]) %>
```

Instead, to avoid hardcoding the roles in the form, we obtain an array of roles from the User model and construct an array of key-value pairs using the `map` method:

```
<%= f.select(:role, User.roles.keys.map {|role| [role.titleize,role]}) %>
```

We use the string `titleize` method to capitalize the name of a role. When the administrator clicks the "Change Role" button, the User controller `update` action will update the database.

Next we'll look at how Rails Composer sets up a User model with enums for roles.

User Model

Rails Composer does all the work of setting up a User model with a `roles` attribute for role-based authorization. We'll look at what Rails Composer does for you. If you want to add roles to a User model in an existing application, you can follow these steps.

Migration

To add a `roles` attribute to an existing User model, Rails Composer generates a migration:

```
$ rails generate migration AddRoleToUsers role:integer
```

The migration will look like this:

```
class AddRoleToUsers < ActiveRecord::Migration
  def change
    add_column :users, :role, :integer
  end
end
```

You'll access roles by name, such as `user` , `vip` , or `admin` , but thanks to Active Record enum, the values are stored as integers in the Users table. The values that correspond to each integer will be defined in the User class.

You have the option of forcing a default role when a database record is created, with the statement `add_column :users, :role, :integer, default: 0` . However, rather than hiding the default in the migration, we'll add code to the User model to set a default role.

Rails Composer runs the migration for you:

```
$ rake db:migrate
```

You can see the result by examining the file db/schema.rb.

User Model

Rails Composer adds some code to the User model. The default User model created by the Devise generator looks like this:

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

Examine the file app/models/user.rb generated by Rails Composer and you'll see the code needed to implement roles:

```
class User < ActiveRecord::Base
  enum role: [:user, :vip, :admin]
  after_initialize :set_default_role, :if => :new_record?

  def set_default_role
    self.role ||= :user
  end

  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

This User model contains code to implement role-based authorization.

The `enum` method sets the `role` attribute with a set of predefined role values: `user` , `vip` , and `admin` . Rails Composer doesn't know what roles you want in your application; this is where you must customize your application for your own requirements. For example, if your application requires three premium membership levels—bronze, silver, and gold—as well as an ordinary user and administrator, change `enum role: [:user, :vip, :admin]` to:

- `enum role: [:user, :bronze, :silver, :gold, :admin]`

The `set_default_role` method sets a role of `user` when the User model is initialized. The statement `after_initialize :set_default_role, :if => :new_record?` calls the method to set the default when the model is initialized.

Refer to the documentation for Enum (http://api.rubyonrails.org/classes/ActiveRecord/Enum.html) to see other options for setting role values. For example, you can use a hash to explicitly map enum values to integers:

- `enum role: {user: 0, bronze: 1, silver: 2, gold: 3, admin: 4}`

Thanks to the power of the Active Record enum attribute, we need only a few lines of code in the User model to provide a complete roles implementation for Pundit to use.

Authorization Without Pundit

You might be surprised to learn that you don't need Pundit to implement role-based authorization. With a role attribute in your user model, you can add simple role-based conditionals to restrict access in controller actions. Pundit is useful when access rules are complex and you want to keep controllers simple. In the next chapter, we'll examine a Users controller that uses Pundit. First, let's consider how we can implement simple role-based authorization in the Users controller without Pundit.

Rails Composer has generated a app/controllers/users_controller.rb file that uses Pundit.

User Controller Without Pundit

Here's an alternate app/controllers/users_controller.rb file:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!

  def index
    unless current_user.admin?
      redirect_to :back, :alert => "Access denied."
    end
    @users = User.all
  end

  def show
    @user = User.find(params[:id])
    unless current_user.admin?
      unless @user == current_user
        redirect_to :back, :alert => "Access denied."
      end
    end
  end

end
```

To keep things simple, I've left out the `update` and `destroy` actions. If we included them, they'd look very similar to the `index` action.

Authentication

Authentication is enforced using Devise, with the method `before_filter :authenticate_user!` forcing the user to sign in before calling any of the controller actions. Devise provides a `current_user` method which supplies the user object for the signed-in user.

Admin?

We only need the method `current_user.admin?` to determine if the current user is an administrator. The `current_user.admin?` method uses a convenience method supplied by the Active Record enum that returns true or false when we ask if the user's role is `admin` .

Active Record automatically gives you convenience methods when you use enum. For example, if you have:

- `enum role: [:coach, :business, :first, :crew]`

You'll automatically get the following methods:

- `user.coach!` – assign the user's role
- `user.business!`
- `user.first!`
- `user.crew!`

- `user.coach?` – query the user's role
- `user.business?`
- `user.first?`
- `user.crew?`

We can thank the Rails maintainers for building this for us.

Index

We want to allow only an administrator to see the list of users on the User index page. The `index` action checks if the current user is an administrator. If the user is not an administrator, the action redirects to the previous page and displays a flash message, "Access denied." If the user is an administrator, an instance variable is assigned an array of users and the Rails render method displays the page (the render method is hidden and implicit).

Show

The `show` action uses simple conditionals to restrict access to the user profile. The user profile is specified by a request parameter and the `User.find` method retrieves the requested user object from the database. If the current user is an administrator, the conditional lets the admin view the user profile. Thus, an admin can view any user profile. If the current user is not an administrator, we want to allow the authenticated user to see their own profile, and exclude all other users. We use `@user == current_user` to check if the current user object is the same as the requested user object. If it is not, we know someone else is trying to view the person's profile and we display an "Access denied" message.

Security Not Obscurity

In our simple application, you might wonder if we really need to block a user from viewing another person's profile. After all, a user has to be an administrator to see a list of users, so there is no way for an ordinary user to click a link to another user's profile. However, a clever user can examine the URL when viewing their own profile: `http://localhost:3000/users/2` . It is not difficult to guess that another profile is available at `http://localhost:3000/users/1` . Even though there is no link on a web page to the other user's profile, we need to protect profiles from users who simply manipulate the URL. We should not rely on "security through obscurity" to protect the user profiles.

Why Use Pundit?

Looking at the `index` and `show` actions, we see that Pundit is not needed for simple role-based authorization. If you don't want to use Pundit in your application, you could add the conditional `if current_user.admin?` to every controller action, providing an "Access denied" alert and redirect for all non-administrative users. We use Pundit for convenience, because it allows us to add authorization to a controller action with the single method call `authorize` . Pundit provides the "Access denied" alert and redirect. More importantly, Pundit provides a mechanism for accessing complex authorization rules, allowing us to segregate complex rules in a discrete *policy object.*

If your application only contains a few administrator-only controller actions, use simple role-based authorization and don’t bother adding Pundit. If you’re building an application with multiple user levels and complex access rules, use Pundit. Pundit keeps controllers skinny.

Next, let’s look at the same Users controller using Pundit.

Pundit Authorization

Rails Composer has done the work of setting up Pundit for managing authorization. We have a `role` attribute available in the User model. Now we can add the `authorize` method to any controller action.

Let’s look at the User controller generated by Rails Composer.

User Controller

Examine the file app/controllers/users_controller.rb:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  after_action :verify_authorized

  def index
    @users = User.all
    authorize User
  end

  def show
    @user = User.find(params[:id])
    authorize @user
  end

  def update
    @user = User.find(params[:id])
    authorize @user
    if @user.update_attributes(secure_params)
      redirect_to users_path, :notice => "User updated."
    else
      redirect_to users_path, :alert => "Unable to update user."
    end
  end

  def destroy
    user = User.find(params[:id])
    authorize user
    user.destroy
    redirect_to users_path, :notice => "User deleted."
  end

  private

  def secure_params
    params.require(:user).permit(:role)
  end

end
```

You'll see a simple controller with the same RESTful actions as the previous example: `index` and `show`. In addition, we have RESTful actions for `update` and `destroy`, as well as a private `secure_params` method to accommodate Rails strong parameters security. The `index` and `show` actions generate the list of users and the individual user profile pages. The `update` and `destroy` actions accommodate the form we examined earlier, allowing the administrator to change a user's role or delete a user.

Authentication

Authentication is enforced using Devise, with the method `before_filter :authenticate_user!` forcing the user to sign in before calling any of the controller actions.

Authorization Protection

The statement `after_action :verify_authorized` isn't strictly necessary. It is a second line of defense, offered by Pundit, that ensures you haven't forgotten to confirm authorization in every controller action where you want it. If you include `after_action :verify_authorized` and a controller action isn't protected with the `authorize` method, the user will see an exception when the unprotected action is called.

If you want to make sure every controller action includes authorization, use `after_action :verify_authorized`. If some actions don't require authorization, specify the exceptions. For example, you can specify `after_action :verify_authorized, except: :show` to require authorization for all actions except `show`.

Index

The `index` action is protected with Pundit using the `authorize User` method call. We add the keyword `authorize` to any controller action that requires authorization for access. The `authorize` method takes an argument that tells Pundit where to find access rules.

Authorize Method

The keyword `authorize` is a helper method that provides a shortcut to a longer statement that implements the actual authorization. We never see the full statement because we use the helper method, but if we were to use it, it would look like this:

```
raise "not authorized" unless UserPolicy.new(current_user, User).index?
```

The `authorize` helper method finds a UserPolicy class and instantiates it, passing the `current_user` object and either the User class or an instance of the User model, and calling an `index?` method to return true or false. You may wonder why we can provide either the User class (as in the `index` action) or the `@user` instance variable (as in the `show` action). We'll explore the differences soon, in the section "Policy Lookup Variations" below.

Devise automatically provides a `current_user` object in every authenticated controller. If you are not using Devise, or you want to use a different user to determine role-based authorization, you can add a `pundit_user` method to your controller:

```
def pundit_user
  User.find_by_other_means # just an example (not a real method)
end
```

The `authorize` method is the gateway to an access rule that will be interpreted before the user is allowed to complete the controller action. Pundit's access rules are called *policies* and defined in a *policy object.* Pundit provides a strict naming convention for policy objects:

- Policy objects should be kept in a folder app/policies.
- A policy object must correspond to a model class, using the name of the model suffixed with the word "Policy," as we'll see in the example `UserPolicy`.

By providing the name of a model as an argument to the `authorize` method, Pundit will instantiate an appropriate policy object. In this case, we'll use the file app/policies/user_policy.rb for a UserPolicy class (we'll examine the policy class in the next chapter). When we want to use the `authorize` method in our controller, we call the method with `authorize User` so Pundit will look for access rules in the UserPolicy class.

Pundit is smart about looking up access rules; not only will it look for a UserPolicy class (because we call `authorize User`), it will look for a specific method named `index?` that corresponds to the name of the controller action. The name of the controller action doesn't need to be specified; it is determined automatically. If you don't want Pundit to automatically use a policy definition with the same name as the controller action, you can pass a second argument to the `authorize` method. For example, you could call `authorize User, :forbidden?` to call `UserPolicy.forbidden?` instead of `UserPolicy.index?`. The `forbidden?` method isn't automatically provided by Pundit; like any other access rule, if you want a `forbidden?` method, you will have to define it in a policy object.

Pundit's policy lookup function can be confusing. Given that the policy object is named UserPolicy, and we will use it for authorization from the Users controller, you might wrongly assume that the name of the policy object will always match the name of the controller. That is not the case. Imagine that we use an IP address geolocation service to identify the country where a browser request originates. We put our geolocation lookup code in a model named Location. Then we can use `authorize Location` to determine access rules for any action in the Users controller. Using a LocationPolicy object we could only allow access for users from Iceland, for example.

Show

The `show` action uses the same `authorize` mechanism to restrict access. Glancing at the controller, we can't tell who is allowed to see the index page or the show page (we have to examine the policy object to find out). But the controller code is clean, with the `authorize` method indicating access rules are in place.

Comparing the `index` and `show` actions, you'll see a small difference in the `authorize` method. The `index` action calls `authorize User`, passing the User class as an argument. The `show` action calls `authorize @user`, passing an instance variable as an argument. Pundit is smart enough to determine that `@user` is an instance of the User class. When we provide an instantiated object as an argument, Pundit will use a policy object that corresponds to the name of the object's class. Calling `authorize @user` from the User controller `show` method will execute access rules defined in a `UserPolicy.show?` method.

Policy Lookup Variations

Pundit will use a UserPolicy object when `authorize` is called from a controller action with any of these arguments:

- `authorize User` – the User class
- `authorize @user` – an instance variable that is an instance of the User class
- `authorize user` – a simple variable that is an instance of the User class
- `authorize @users` – an array of User objects

- `authorize foo` – where the variable `foo` is an instance of the User class

These variations accommodate the full range of objects you'll have available in a typical RESTful controller. At first glance, the mix of different objects is confusing. But it provides great flexibility.

Rails beginners often wonder about the difference between `@user` (an instance variable) and `user` (a simple variable). The instance variable `@user` will be available in a view and the simple variable `user` is only available in the method where it is assigned. The book Learn Ruby on Rails (https://tutorials.railsapps.org/tutorials/learn-rails) goes into more detail.

For a deeper look at the Pundit policy lookup mechanism, and all its possibilities, see the chapter ahead, "Pundit Policy Lookup."

Update

The `update` action accommodates submission of a form that changes a user's role. We use the same syntax as the `show` action for the authorization call: `authorize @user`, supplying an instance variable to Pundit to obtain the UserPolicy object.

Destroy

The `destroy` action allows the administrator to delete a user. The `@user` instance variable is not needed for use in a view, so we assign the user object to a simple variable `user`. We check authorization with `authorize user`.

In the next chapter, we'll examine a Pundit policy object.

Pundit Policy Object

Policy objects are the core of authorization with Pundit. The primary purpose of the Pundit gem is to make it convenient to use simple Ruby objects for defining access rules. The Pundit gem makes it easy to move authorization code out of controllers and into separate classes, organized in the app/policies folder.

We've seen how to set up a Users controller to obtain authorization from a policy object. Now let's look at the UserPolicy class that defines the access rules.

UserPolicy

Examine the file app/policies/user_policy.rb:

```
class UserPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @user = model
  end

  def index?
    @current_user.admin?
  end

  def show?
    @current_user.admin? or @current_user == @user
  end

  def update?
    @current_user.admin?
  end

  def destroy?
    return false if @current_user == @user
    @current_user.admin?
  end

end
```

Policies Folder

The Pundit documentation recommends placing policy objects in the app/policies folder. This is not strictly necessary (you could place the policy objects in the lib folder or anywhere else) but it makes sense to follow the suggested convention. Newcomers to Rails are often uncomfortable when asked to create a new folder that is not part of the standard Rails directory structure. Rest assured, it is perfectly okay to create new folders. In fact, it is good practice, as a Rails application grows in complexity, to add plain old Ruby objects to perform various services, in one or more new folders (this is often called a "service-oriented architecture").

Class Definition

Notice the class definition `class UserPolicy`. It doesn't inherit from any parent class. It is a plain old Ruby object, which means there are no inherited methods other than what is found in any simple Ruby object. We have to implement everything we need. That's a benefit; there is no hidden API or domain specific language to learn. Fortunately, the boilerplate needed to implement a policy object is very simple, so we don't have to add much code to turn a simple Ruby object into a policy object.

The name of the class must correspond to an existing model class (one that inherits from ActiveRecord or ActiveModel). The name of the model class is combined with "Policy" to form the class name. This allows Pundit to find and instantiate the class from the `authorize` method.

Attributes

The policy class must accommodate two attributes. First, Pundit will always initialize the object by providing a `current_user` object. This accommodates a wide range of access rules that are conditioned on attributes of the current user, such as role. Second, the Pundit `authorize` method always provides a model object that corresponds to the name of the policy object (in this case, a User model). Our plain old Ruby object needs methods to define the current user and the model. We could do something like:

```
def current_user
  @current_user
end

def model
  @model
end
```

It is easier to use the standard Ruby `attr_reader` accessor method to do the same thing in a single statement:

```
attr_reader :current_user, :model
```

Elsewhere, you may see examples of Pundit policy objects that inherit from the Struct (http://www.ruby-doc.org/core-2.0/Struct.html) class. Struct eliminates the need for `attr_reader` or `initialize`. Struct is a shortcut that is not always a good practice (http://thepugautomatic.com/2013/08/struct-inheritance-is-overused/).

Initialization

We need an initialization method:

```
def initialize(current_user, model)
  @current_user = current_user
  @user = model
end
```

Most policy objects will need the `current_user` attribute, especially if the application uses role-based authorization.

Some simple policy objects may not need to use the `model` attribute at all. If you don't need data or methods from a model for your access rules, you don't need the model. If we're building a policy object for another model, `LocationPolicy` for example, we'd modify the initialization method accordingly, with `@location = model`.

Comparing Controller and Policy Object

It can be confusing to look at the `authorize` method used in a controller and compare it to the initialization method in the policy object. The `authorize` method (used in a controller) takes one required parameter, `model`, and one optional parameter, `action`:

```
authorize User, :forbidden?
```

The policy object initialization method used by Pundit internally (you never see it), looks like this:

```
UserPolicy.new(current_user, model)
```

Obviously, the method signatures are different. It helps to notice the difference, and realize that the `authorize` method doesn't map directly to the policy object initialization method. In the controller `authorize` method, the first argument is a model and an optional second argument is the name of a method in the policy object. When the policy object is initialized, `current_user` is the first argument (obtained automatically), and the second argument is the model (passed from the `authorize` method call).

Improved Initialization

Since we are using Devise, we have `before_filter :authenticate_user!` in every controller that contains actions requiring authentication or authorization. If you are not using Devise, you can improve the initialization of any policy object by checking if the current user is signed in:

```
def initialize(current_user, model)
  raise Pundit::NotAuthorizedError, "must be logged in" unless current_user
  @current_user = current_user
  @user = model
end
```

This is a "belt and suspenders" approach that adds a second layer of protection if you are already using Devise to require authentication.

Access Rules

You'll need to define a method that corresponds to each controller action that requires authorization. Use the name of the controller action combined with a `?` (question mark) character. By convention in Ruby, method names ending in question marks are expected to return true or false. Here we provide access rules for the controller actions:

```
def index?
  @current_user.admin?
end

def show?
  @current_user.admin? or @current_user == @user
end

def update?
  @current_user.admin?
end

def destroy?
  return false if @current_user == @user
  @current_user.admin?
end
```

We've been calling these methods "access rules", which is descriptive of their function. There is nothing special about these methods. As long as the method has the name of a controller action followed by a question mark, and returns a boolean, it serves as an access rule for Pundit authorization. Here we've provided simple conditionals to implement the access rules we used in our example of role-based authorization without Pundit.

The `index?` and `update?` methods are similar. If the current user is an administrator, we return "true" to allow access.

The `show?` method allows the signed-in user to see his or her own profile, and allows administrators to see any user's profile.

The `destroy?` method allows administrators to delete any user account except their own. The method returns false if someone tries to delete their own account.

This is a simple example of a Pundit policy object. You'll have to judge for yourself whether your application is sufficiently complex to warrant use of Pundit. If you anticipate your application will grow in complexity, it is a good idea to use Pundit.

Custom Violation Messages

When we looked at how to set up Pundit, we saw an initializer file that provides an "Access denied" alert when a user is not authorized for a particular request.

If we don't implement exception handling for unauthorized access attempts, in production, the user will see a generic "something went wrong" page. In development, you will see an exception and stack trace. For better usability, we provide an error message. In this chapter, we'll see how to customize the error message.

Here is the file config/initializers/pundit.rb generated by Rails Composer:

```
# config/initializers/pundit.rb
# Extends the ApplicationController to add Pundit for authorization.
# Modify this file to change the behavior of a 'not authorized' error.
# Be sure to restart your server when you modify this file.
module PunditHelper
  extend ActiveSupport::Concern

  included do
    include Pundit
    rescue_from Pundit::NotAuthorizedError, with: :user_not_authorized
  end

  private

  def user_not_authorized
    flash[:alert] = "Access denied."
    redirect_to (request.referrer || root_path)
  end

end

ApplicationController.send :include, PunditHelper
```

Here we create a Rails flash message, “Access denied.” Then we redirect to the previous page or the home page (if there is no previous page). The `request.referrer` method provides the URL of the previous page. We can also use `request.referer` (one less “r” character) because Rails accommodates the misspelling that was set in stone in 1996 (http://en.wikipedia.org/wiki/HTTP_referer) by the standards document RFC 1945.

If we want to display more than “Access denied,” we can customize the `user_not_authorized` method or add custom exceptions. We can either modify the initializer file, providing the same message for all Pundit access violations, or we can modify individual controller files, providing custom messages for each.

Custom Message for a Controller

If you want a customized message for a particular controller, you can customize the `user_not_authorized` method. For example, we could add the following to the Users controller:

```
def user_not_authorized
  flash[:alert] = "No way! This operation is not allowed."
  redirect_to (request.referrer || root_path)
end
```

This displays an identical message for violation of access rules for any action in the controller. It overrides the `user_not_authorized` method in the initializer file.

Custom Message Using Pundit Information

A standard Pundit exception message provides details about the policy object and method that caused an exception:

- `policy` – the policy object
- `record` – the model name
- `query` – the controller action

You can create a custom alert using this information:

```
def user_not_authorized(exception)
  policy_name = exception.policy.class.to_s
  flash[:alert] = "Access denied by #{policy_name} for #{exception.record} #{exception.query.ch
op}"
  redirect_to (request.referrer || root_path)
end
```

You'd see a message like this:

```
Access denied by UserPolicy for User index
```

This is not a very helpful error message for a user. The Pundit README (https://github.com/elabs/pundit) contains a better example that shows how to use the Rails localization facility to display informative messages using the Pundit exception information. Here's an example using localization:

```
def user_not_authorized(exception)
  policy_name = exception.policy.class.to_s.underscore

  flash[:error] = I18n.t "pundit.#{policy_name}.#{exception.query}",
    default: 'You cannot perform this action.'
  redirect_to (request.referrer || root_path)
end
```

To provide localization for the English language, you can create a file config/locales/en.yml:

```
en:
  pundit:
    user_policy:
      index?: 'No way! You are not an administrator!'
      show?: "We won't let you see that profile."
```

Notice that we use double quotation marks when the text includes an apostrophe.

Custom Message for a Controller Action

As shown above, your custom error messages can be defined in a locale file. You can also define error messages within the policy objects, using custom exceptions. Either approach lets you define error messages for authorization violations for specific controller actions.

Here's a new version of the file app/policies/user_policy.rb:

```
class UserPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @user = model
  end

  def index?
    unless @current_user.admin?
      raise AuthorizationError, 'Access for administrators only.'
    end
  end

  def show?
    unless @current_user.admin? or @current_user == @user
      raise AuthorizationError, 'Access denied to user profile.'
    end
  end

  def update?
    @current_user.admin?
  end

  def destroy?
    return false if @current_user == @user
    @current_user.admin?
  end

  class AuthorizationError < StandardError
  end

end
```

Notice we've created a custom exception class:

```
class AuthorizationError < StandardError
end
```

We catch and display our custom `AuthorizationError` in the Users controller. Here's a new version of the file app/controllers/users_controller.rb:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  after_action :verify_authorized

  rescue_from UserPolicy::AuthorizationError, with: :user_not_authorized

  def index
    @users = User.all
    authorize User
  end

  def show
    @user = User.find(params[:id])
    authorize @user
  end

  def update
    @user = User.find(params[:id])
    authorize @user
    if @user.update_attributes(secure_params)
      redirect_to users_path, :notice => "User updated."
    else
      redirect_to users_path, :alert => "Unable to update user."
    end
  end

  def destroy
    user = User.find(params[:id])
    authorize user
    user.destroy
    redirect_to users_path, :notice => "User deleted."
  end

  private

  def secure_params
    params.require(:user).permit(:role)
  end

  def user_not_authorized
    flash[:alert] = $!
    redirect_to (request.referrer || root_path)
  end

end
```

We've added a statement:

```
rescue_from UserPolicy::AuthorizationError, with: :user_not_authorized
```

And modified the `user_not_authorized` method:

```
def user_not_authorized
  flash[:alert] = $!.message
  redirect_to (request.referrer || root_path)
end
```

We set `flash[:alert]` to `$!.message`. The cryptic Ruby (http://jimneath.org/2010/01/04/cryptic-ruby-global-variables-and-their-meanings.html) dollar-bang symbol contains the exception that was most recently raised. This is a trick to get the message we set in an exception.

To make this custom message available in every controller, we could modify the config/initializers/pundit.rb initializer file.

Which is Best?

There is no recommended best practice for implementing custom error messages; your approach will depend on your particular application. If your application is localized for multiple languages, you should stick to that approach for authorization error messages. If you need to let users know why they've been denied access, and you are translating for multiple languages, define your authorization error messages in config/locales/. If you are not translating for locales, it might be easier to create custom exceptions in the policy object, perhaps in combination with a generic error message.

In general, if "Access Denied" is sufficient for all authorization violations, keep it simple and don't add custom error messages.

Authorization in Views

We've seen how to add authorization to controllers; let's see how we add authorization to views. Just as with controllers, we can use either simple role-based authorization or we can use Pundit for more complex access rules.

Our starter application contains an example of authorization in a view.

Simple Role-based Authorization

Examine the file app/views/layouts/_navigation_links.html.erb:

```
<%# add navigation links to this file %>
<% if user_signed_in? %>
  <li><%= link_to 'Edit account', edit_user_registration_path %></li>
  <li><%= link_to 'Sign out', destroy_user_session_path, :method=>'delete' %></li>
<% else %>
  <li><%= link_to 'Sign in', new_user_session_path %></li>
  <li><%= link_to 'Sign up', new_user_registration_path %></li>
<% end %>
<% if user_signed_in? %>
  <% if current_user.admin? %>
    <li><%= link_to 'Users', users_path %></li>
  <% end %>
<% end %>
```

We use the `user_signed_in?` method supplied by Devise to toggle display of links to authentication pages.

We use simple role-based authorization to restrict display of one link:

```
.
.
.
  <% if current_user.admin? %>
    <li><%= link_to 'Users', users_path %></li>
  <% end %>
.
.
.
```

Only users with an administrator role will see the link to the Users index page. In this case, simple role-based authorization is all we need.

Authorization With Pundit

Pundit provides a `policy` helper method that is similar to the `authorize` helper that we use in our controller.

If simple role-based authorization is not sufficient, we can improve the file app/views/layouts/_navigation_links.html.erb to use Pundit authorization:

```
.
.
.
  <% if policy(User).index? %>
    <li><%= link_to 'Users', users_path %></li>
  <% end %>
.
.
.
```

We use the `policy(...)` helper, indicating we want to use the policy object that corresponds to the User model, and calling the `index?` method to find out if access is allowed. We've already defined an access rule for the User index page that we use in the controller; we can use the same rule to display the link. The access rule defined by `UserPolicy.index?` returns true when the current user is an administrator.

Pundit gives us the flexibility to use any instance of the User model to look up the policy object. In our navigation partial, we don't have the variable `user` or `@user`, but we have the `current_user` provided by Devise. Here's an alternative version, using the `current_user`:

```
.
.
.
  <% if policy(current_user).index? %>
    <li><%= link_to 'Users', users_path %></li>
  <% end %>
.
.
.
```

This only works if the user is signed in, as `current_user` will not be available otherwise. If you look at the file, you'll see we've wrapped the condition in `<% if user_signed_in? %>`, so we only try to display the link to signed-in users. If we want to simplify the code in the navigation partial and eliminate the `<% if user_signed_in? %>` condition, we can modify the UserPolicy object in the file app/policies/user_policy.rb:

```
.
.
.
  def index?
    return false if @current_user.nil?
    @current_user.admin?
  end
.
.
.
```

This access rule allows us to control display of the link to the Users index page even when the user is not signed in, eliminating the need to wrap the navigation link in the `<% if user_signed_in? %>` condition.

Scoped Database Queries

Pundit adds convenience when you need to restrict the information returned by a database query.

Example

Imagine you want administrators to see a list of all the users. And you want ordinary users to only see other ordinary users, and no administrators. You could implement this restriction in the User controller `index` action:

```
def index
  if current_user.admin?
    @users = User.all
  else
    @users = User.where("role = ?", User.roles[current_user.role])
  end
end
```

In this implementation, when the current user is an administrator, the `@users` instance variable will contain a list of all users. If the current user is an ordinary user, we perform a search query that returns only users in the same role. Notice that we don't restrict access to the view; any user can see a list of users, but the displayed dataset is limited by the user's role.

Defining Scope

Rails provides a mechanism to return a predefined database query. In earlier versions of Rails, this was called *named scope_. Beginning with Rails 3.0, we simple call it scope. For more detail, see RailsGuide: Active Record Query Interface (http://guides.rubyonrails.org/active_record)*querying.html#scopes.

Pundit makes it easy to restrict database queries based on access rules, using the Pundit `policy_scope` helper method. Here's a policy object that will respond to the `policy_scope` method:

```
class UserPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @user = model
  end

  def index?
    # @current_user.admin?
    true
  end

  def show?
    @current_user.admin? or @current_user == @user
  end

  def update?
    @current_user.admin?
  end

  def destroy?
    return false if @current_user == @user
    @current_user.admin?
  end

  class Scope < Struct.new(:current_user, :model)
    def resolve
      if current_user.admin?
        model.all
      else
        model.where(role: current_user.role)
      end
    end
  end

end
```

Notice that we've changed the access rule for the `index?` method. Unlike the implementation we've used elsewhere in the tutorial application, we're allowing any user to view the User index page. In the examples below, we don't call `authorize User`, but if we did, the `index?` rule would allow access to anyone.

To create a scope, Pundit expects us to define a `Scope` class within our `UserPolicy` class. In this case, rather than implementing an `attr_reader` and `initialize` method, we follow the Pundit maintainers' recommendation and subclass a `Struct` class to get an object prebuilt with accessor and initialize methods.

Pundit requires us to define a `resolve` method that returns an array or other collection (anything that can be iterated with `each`, including an `ActiveRecord::Relation` object that is returned by an Active Record database query).

Calling Scope

Here's a new version of our User controller `index` action that uses the Pundit `Scope` we've defined:

```
def index
  @users = UserPolicy::Scope.new(current_user, User).resolve
end
```

In this example, you see the full signature of the class and methods. To get an array of `@users`, we need the `UserPolicy::Scope`. We call `new`, passing the `current_user` object and the `User` model class to instantiate the `Scope` object. Then we call `resolve` on the `Scope` object to get the search query results.

The Pundit gem supplies a shortcut, giving us the `policy_scope` helper method that does the same thing with less code:

```
def index
  @users = policy_scope(User)
end
```

The `policy_scope` helper method returns the same result. Implicitly, the `current_user` object is supplied to the `UserPolicy::Scope` object. The supplied argument will usually be a model, but it can be any ActiveRecord class or an ActiveRecord::Relation object.

Requiring Scope

Earlier we saw the `verify_authorized` helper method that we can use in controllers to ensure that every controller action is protected with an `authorize` directive. A similar helper method, `verify_policy_scoped`, is available to make sure any database queries in a controller are protected by the `policy_scope` helper method.

Here is an example:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  after_action :verify_authorized
  after_action :verify_policy_scoped, :only => :index
  .
  .
  .
end
```

In this case, we double check that we are using the `policy_scope` helper method in the controller `index` action.

Scope in Views

You can use the Pundit `policy_scope` helper method in views. For example, we could eliminate the `index` action entirely in the file app/controllers/users_controller.rb:

```
def index
end
```

And replace the file app/views/users/index.html.erb with this:

```
<div class="container">
  <div class="row">
    <h3>Users</h3>
    <div class="column">
      <table class="table">
        <tbody>
          <% policy_scope(User).each do |user| %>
            <tr>
              <%= render user %>
            </tr>
          <% end %>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Here the `UserPolicy::Scope` class does all the work of returning a collection of users consistent with the restrictions we've implemented in the `resolve` method.

Why Use Pundit Scope?

The Pundit scope mechanism is a powerful way to restrict database queries based on access rules defined in the `UserPolicy::Scope` class. The examples we've examined are trivial and it would be easier to stick a few conditional statements in a controller action if this was all we needed. Keep in mind that Pundit doesn't exist to reduce your total lines of code; rather, it exists to reduce the lines of code in your controllers, so you can keep controllers lean and simple.

Pundit makes authorization more robust by ensuring that, for each model, all the access rules that restrict views or datasets are in a single place, the policy class. If you make sure that every controller action is restricted by `authorize`, and every database query is restricted by `policy_scope`, you can be confident that authorization is controlled by the access rules you've defined in the policy class.

Strong Parameters

Rails 4.0 introduced "strong parameters" to prevent a class of security exploits called "mass-assignment vulnerabilities." Rails won't let us initialize a model with just any parameters submitted on a form. Suppose we are creating a new user and one of the user attributes is a flag allowing administrator access. A malicious

hacker could create a fake form that provides a user name and sets the administrator status to “true.” Rails forces us to “white list” each of the parameters used to initialize the model.

Our example Users controller contains a method to sanitize parameters submitted from forms:

```
def secure_params
  params.require(:user).permit(:role)
end
```

As generated by Rails Composer, it only allows the administrator to change the user’s role (not name or email address).

Imagine that we add a form to the application that allows a user to change a name and email address. We would need to modify the `secure_params` method:

```
def secure_params
  params.require(:user).permit(:name, :email)
end
```

These permitted parameters only allow a user to change name and email address. We don’t want an ordinary user changing their role.

Now imagine that we want to allow an administrator to change the user’s role and we want to allow the user to change his or her name and email address. Pundit provides a `permitted_attributes` helper method that lets us define the attributes we allow to be changed.

We can create a new UserPolicy class in the file app/policies/user_policy.rb:

```
class UserPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @user = model
  end

  def index?
    @current_user.admin?
  end

  def show?
    @current_user.admin? or @current_user == @user
  end

  def update?
    @current_user.admin?
  end

  def destroy?
    return false if @current_user == @user
    @current_user.admin?
  end

  def permitted_attributes
    if @current_user.admin?
      [:role]
    else
      [:name, :email]
    end
  end

end
```

The `permitted_attributes` allows an administrator to change a user's role. Anyone else can change a name and email address but cannot change a role.

We use the Pundit `permitted_attributes` helper method in the Users controller, replacing the conventional `secure_params` method:

```
def secure_params
  params.require(:user).permit(*policy(@user || User).permitted_attributes)
end
```

Pundit allows us to retrieve a policy object without a helper method. The policy object is available anywhere in the controller. This line of code contains the Ruby splat (http://pivotallabs.com/ruby-pearls-vol-1-the-splat/) operator. Ordinary people will recognize the "splat" as a "star" or asterisk character. As used here, the splat operator converts an array returned by the `UserPolicy.permitted_attributes` method to arguments expected

by the `permit` method. We call the policy object with either the `@user` instance variable or the User class, depending on what's available in the controller action that calls `secure_params`. It's a complex bit of code, but it allows us to accommodate strong parameters, segregating the authorization logic in the policy object.

Testing Authorization With RSpec

Testing becomes especially important when you are building an application that behaves differently depending on who is signed in. It is more difficult to rely on manual testing to reveal failures, so automated testing is essential. We'll look at testing using the popular RSpec framework. If you are new to testing, the RSpec Quickstart Guide (https://tutorials.railsapps.org/tutorials/rspec-quickstart) gives an introduction and explains how to get started.

The Pundit README (https://github.com/elabs/pundit#rspec) briefly describes test helpers that are included in the Pundit gem. We'll look at how Rails Composer sets up the test helpers and then examine our authorization test suite.

Pundit Test Helpers

To enable the Pundit test helpers, you must add `require 'pundit/rspec'` to the RSpec support system.

You can add `require 'pundit/rspec'` to the spec/rails_helper.rb file:

```
# This file is copied to spec/ when you run 'rails generate rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require 'spec_helper'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'pundit/rspec'
.
.
.
```

Rails Composer avoids making changes to the spec/rails_helper.rb file, in favor of creating a separate file.

Rails Composer adds a file spec/support/pundit.rb containing one line:

```
require 'pundit/rspec'
```

Either approach has the same result.

The Pundit test helpers add a custom example group (https://www.relishapp.com/rspec/rspec-core/v/3-0/docs/example-groups/basic-structure-describe-it) named "permissions." The new example group works like `describe` or `feature` to set up tests ("examples" in RSpec terminology) of a policy object. The Pundit test helpers also add the `permit` custom matcher which executes an access rule in the policy object. You'll see how to use `permissions` and `permit` in the following example.

UserPolicy Test Suite

The Pundit maintainers recommend placing all tests of policy objects in the spec/policies/ folder.

Rails Composer generates a test suite to accompany the UserPolicy class.

Examine the file spec/policies/user_policy_spec.rb:

```
describe UserPolicy do
  subject { UserPolicy }

  let (:current_user) { FactoryGirl.build_stubbed :user }
  let (:other_user) { FactoryGirl.build_stubbed :user }
  let (:admin) { FactoryGirl.build_stubbed :user, :admin }

  permissions :index? do
    it "denies access if not an admin" do
      expect(subject).not_to permit(current_user)
    end
    it "allows access for an admin" do
      expect(subject).to permit(admin)
    end
  end

  permissions :show? do
    it "prevents other users from seeing your profile" do
      expect(subject).not_to permit(current_user, other_user)
    end
    it "allows you to see your own profile" do
      expect(subject).to permit(current_user, current_user)
    end
    it "allows an admin to see any profile" do
      expect(subject).to permit(admin)
    end
  end

  permissions :update? do
    it "prevents updates if not an admin" do
      expect(subject).not_to permit(current_user)
    end
    it "allows an admin to make updates" do
      expect(subject).to permit(admin)
    end
  end

  permissions :destroy? do
    it "prevents deleting yourself" do
      expect(subject).not_to permit(current_user, current_user)
    end
    it "allows an admin to delete any user" do
      expect(subject).to permit(admin, other_user)
    end
  end

end
```

The RSpec framework provides a domain-specific language used for testing. Learning RSpec is like learning another language, since it has its own specialized keywords.

The directive `describe UserPolicy` sets up a block of tests (“examples” in RSpec terminology). Specify the

policy object to be tested with `subject { UserPolicy }`; this value is passed to the `expect` method in each test.

We create three different users with FactoryGirl (https://github.com/thoughtbot/factory_girl). These users are specifically needed for testing our UserPolicy. For tests of other policy objects, you might need to instantiate other objects in the set-up phase of the tests. We use RSpec's `let` helper method to cache the three users before executing any tests. We could use FactoryGirl's `create`, `build`, or `build_stubbed` methods to instantiate the User object; we use the `build_stubbed` method because it builds the object in memory only, making tests run faster.

We have a set of tests (a "group of examples" in RSpec terminology) for each access rule in the policy object. When we run RSpec, the `subject { UserPolicy }` instantiates the UserPolicy object and the `permissions` keyword executes the access rule for each test in the block. The keyword `it` sets up each individual test.

We can see how the RSpec syntax is used in the first test of the `index?` access rule.

We will test an access rule from the file app/policies/user_policy.rb:

```
def index?
  @current_user.admin?
end
```

Here is our test from the file spec/policies/user_policy_spec.rb:

```
permissions :index? do
  it "denies access if not an admin" do
    expect(UserPolicy).not_to permit(current_user)
  end
  .
  .
  .
end
```

You should write a clear description of the test to follow `it`. You should pass `subject` as an argument to `expect()`; if not, you can write `expect(UserPolicy)` but it is considered better practice to use `subject` to avoid duplication. Finally, call the `permit` method and pass a user object as an argument. You can pass a single user object, which will serve as the current user in the access rule. You can pass an (optional) second user object, if the access rule expects it. This test is easier to understand. We expect the `index?` method of the UserPolicy object not to permit the current user to access the page if he or she is not an administrator.

The next test is very similar, but we use the `admin` object created by FactoryGirl:

```
permissions :index? do
  .
  .
  .
  it "allows access for an admin" do
    expect(UserPolicy).to permit(admin)
  end
end
```

This time, we expect the `index?` method of the UserPolicy object to permit the admin to access the page.

We'll look at another test to see how we can pass a second user object to the access rule. We want to test this access rule:

```
def show?
  @current_user.admin? or @current_user == @user
end
```

Here is our test:

```
permissions :show? do
  it "prevents other users from seeing your profile" do
    expect(subject).not_to permit(current_user, other_user)
  end
  .
  .
  .
end
```

We expect the `show?` method of the UserPolicy object not to permit the current user to view the profile of the other user. However, in the next test, we expect the current user to see his or her own profile:

```
permissions :show? do
  .
  .
  .
  it "allows you to see your own profile" do
    expect(subject).to permit(current_user, current_user)
  end
  .
  .
  .
end
```

Finally, we test the `destroy?` rule:

```
def destroy?
  return false if @current_user == @user
  @current_user.admin?
end
```

Here are our tests:

```
permissions :destroy? do
  it "prevents deleting yourself" do
    expect(subject).not_to permit(current_user, current_user)
  end
  it "allows an admin to delete any user" do
    expect(subject).to permit(admin, other_user)
  end
end
```

In the first test, we provide the `current_user` object twice, which verifies that you cannot delete yourself. In the second test, we pass two different user objects, one of which is the administrator, to verify the administrator can delete anyone.

Run RSpec

When you run the `rspec` command, you'll run all the test cases in all the files in the spec/ folder.

Run all tests using the `rspec` command:

```
$ rspec
```

Alternatively, you can run the rake task `rake spec`.

Rails Composer also generated a test suite for Devise. All the Devise tests have a `devise` tag. If you want to exclude the Devise tests when you run RSpec:

```
$ rspec --tag ~devise
```

By adding the argument `--tag ~devise` to the `rspec` command, you will exclude all tests marked with the `devise` tag. Notice that you prepend the `~` (tilde) character to indicate tests with the tag `devise` tag should be excluded.

Pundit Policy Lookup

Pundit looks for a policy object when `authorize` is called from a controller action. We already saw that Pundit will find a UserPolicy if given any of these arguments:

- `authorize User` – the User class
- `authorize @user` – an instance variable that is an instance of the User class
- `authorize user` – a simple variable that is an instance of the User class

- `authorize @users` – an array of User objects

These variations accommodate all the objects you'll have available in a typical RESTful controller. But what if no user object is available? Pundit offers great flexibility in its policy lookup scheme.

Pundit Source Code

Let's examine the Pundit source code to see what it can do. Here's the find method (https://github.com/elabs/pundit/blob/master/lib/pundit/policy_finder.rb) that identifies the policy class:

```
def find
  if object.respond_to?(:policy_class)
    object.policy_class
  elsif object.class.respond_to?(:policy_class)
    object.class.policy_class
  else
    klass = if object.respond_to?(:model_name)
      object.model_name
    elsif object.class.respond_to?(:model_name)
      object.class.model_name
    elsif object.is_a?(Class)
      object
    else
      object.class
    end
    "#{klass}Policy"
  end
end
```

Demonstration

Let's make a new Access class in the file app/models/access.rb:

```
class Access
end
```

We'll also make a new AccessPolicy class in the file app/policies/access_policy.rb:

```
class AccessPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @model = model
  end

  def show?
    false
  end

end
```

For a demonstration, we'll set a simple access rule for `show?` to deny all access.

We'll change the Users controller `show` action in the file app/controllers/users_controller.rb:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  after_action :verify_authorized
  .
  .
  .
  def show
    @user = User.find(params[:id])
    authorize Access
  end
  .
  .
  .
end
```

In this case, the source code `object.is_a?(Class)` will use the `Access` class name to construct `"#{klass}Policy"` and instantiate the `AccessPolicy` object.

Let's make a new version of the Access class that inherits from `ActiveRecord::Base` :

```
class Access < ActiveRecord::Base
end
```

Try using an instantiated `Access` object in the Users controller:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  after_action :verify_authorized
  .
  .
  .
  def show
    @user = User.find(params[:id])
    authorize Access.new
  end
  .
  .
  .
end
```

Any class that inherits from `ActiveRecord::Base` automatically defines `model_name`. The source code `object.respond_to?(:model_name)` will use the `model_name` attribute to construct `"#{klass}Policy"` and instantiate the `AccessPolicy` object.

Pundit will also use `policy_class` or `model_name` attributes if they are defined in the object. Try this:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  after_action :verify_authorized
  .
  .
  .
  def show
    @user = User.find(params[:id])
    authorize Foo.new
  end
  .
  .
  .
end
```

And use either of two versions of the `Foo` class. You can create the file app/models/foo.rb:

```
class Foo

  def self.policy_class
    AccessPolicy
  end

end
```

```
class Foo

def self.model_name
  'Access'
end

end
```

In the first case, the source code `object.respond_to?(:policy_class)` will find that `policy_class` is defined and instantiate the `AccessPolicy` object.

In the second case, the source code `object.respond_to?(:model_name)` will find that `model_name` is defined and use the `model_name` attribute to construct `"#{klass}Policy"` and instantiate the `AccessPolicy` object.

This exercise demonstrates that you can pass any object to Pundit to find the policy object. Since classes are objects in Ruby, you can pass a class, too. The Pundit gem contains the logic to find the policy object and apply the appropriate access rule.

Universal Policy Object

We can use Pundit's flexible policy lookup to create a policy object that can be used anywhere in an application. If you don't want to create a separate policy object for each model, you can create a generic policy object.

Create a file app/models/access.rb:

```
class Access
end
```

And create a policy object in the file app/policies/access_policy.rb:

```
class AccessPolicy
  attr_reader :current_user, :model

  def initialize(current_user, model)
    @current_user = current_user
    @model = model
  end

  def index?
    false
  end

  def show?
    false
  end

  def create?
    false
  end

  def new?
    false
  end

  def update?
    false
  end

  def edit?
    false
  end

  def destroy?
    false
  end

end
```

You can use this policy object in any controller with the helper method `authorize Access`.

This policy object will block access to any controller for any user, so you will want to modify it to accommodate your own access rules.

You can also inherit from this policy object to create other policy objects. For example, you can create a UserPolicy object by subclassing `AccessPolicy`:

```
class UserPolicy < AccessPolicy

  def index?
    @current_user.admin?
  end

  def show?
    @current_user.admin? or @current_user == @user
  end

  def update?
    @current_user.admin?
  end

  def destroy?
    return false if @current_user == @user
    @current_user.admin?
  end

end
```

Inheriting from `AccessPolicy` eliminates the need to set up the `attr_reader` and `initialize` methods. You can override the methods that need special handling and inherit the rest. You'd use this UserPolicy class in a controller the same way you did previously. For example:

```
def show
  @user = User.find(params[:id])
  authorize @user
end
```

Pundit gives you a great deal of flexibility because it's all just Ruby.

Comments

Credits

Daniel Kehoe wrote the guide.

Did You Like This Guide?

Was the guide useful to you? Follow @rails_apps (http://twitter.com/rails_apps) on Twitter and tweet some praise. I'd love to know you were helped out by the article.

You can also find me on Facebook (https://www.facebook.com/daniel.kehoe.sf) or Google+ (https://plus.google.com/+DanielKehoe/).

Jordan • 5 days ago

I am getting this: I am assuming that I am getting something mixed up with Devise in my app because my Pundit code is looking the same as yours. I will keep trying to debug this. If anyone has any thoughts that would be great!

Daniel Kehoe RailsApps ➦ Jordan • 5 days ago

You can clone the git repo for rails-devise-pundit. The code is known to work, so compare with a file compare tool.

Jordan • 5 days ago

This is very helpful, but I'm having a hard time with moving forward. I am wondering why I am getting
undefined method `authorize'

```
def index
@users = User.all
# Look for rules in the UserPolicy class
authorize User
end
```

...on the index page? I have the root set to root 'users#index' and I am using Devise.

Scott • a month ago

Great guide! I've been looking for instructions on Pundit for the longest time now.. One question however : If I make the Devise example app by hand (not with Rails Composer) following the "Devise QuickStart" instructions, will I then be able to follow this tutorial's steps and add Pundit authorization to the app no problem?

specifically:---> I'm confused by the 'User Pages' step in the Devise QuickStart, where it says "Finally, Rails Composer uses the RailsApps Pages gem to add a User controller, views, routes, and RSpec tests to the application." ..... should choose the ($ rails generate pages:users) option? Or the ($ rails generate pages:authorized) option? Since Pundit will not have been installed by this point yet

Daniel Kehoe RailsApps Scott • a month ago

You can make the Devise app by hand, then run ($ rails generate pages:users) which should reproduce the rails-devise example (check with a file-compare tool). Then install Pundit and run ($ rails generate pages:authorized) which should reproduce the rails-devise-pundit example app (again, check with a file-compare tool). You'll be missing RSpec tests. Use the rails_apps_testing gem to add those if you want.

• Reply • Share ›

Scott Daniel Kehoe • a month ago

You're awesome ;) Thanks for the quick reply! I'll definitely be renewing my subscription to RailsApps

Sort of off topic, but would you mind pointing me in the direction of a good tutorial explaining how to add things to Users, like the ability to post a classified item ad/listing (basically like Craigslist)? I realize I'll need to abstract the information from whatever tutorial I use since most explain how to make a first app like Twitter with simple text-based 140 character posts rather than actually creating a unique