django-logical-rules Documentation

Release 1.0

Benjamin W Stookey

December 03, 2013

Contents

When you need logic...

A tool to manage logical rules throughout your application. Logical rules are more powerful than permission or rule tables because they are written in python. Register a rule once and work with it throughout your app, from templates to generic view mixins. Instead of cluttering your models with rule-style and permission-style methods define those rules in rules.py and then get easy access to them in your views and templates.

CHAPTER 1

Usage

1.1 Template Tags

Once you have created a rule, it's easy to use anywhere in your templates:

```
{% load logical_rules_tags %}
{% testrule user_can_edit_mymodel object request.user %}
        <p>You are the owner!</p>
{% endtestrule %}
```

Note: *Don't use quotes around the rule name in the template.*

1.2 Direct Calling

```
import logical_rules
if logical_rules.site.test_rule(rule['name'], arg1, arg2):
        print "passed"
else:
        print "failed"
```

1.3 RulesMixin

If you are extending Django's class-based generic views, you might find this mixin useful. It allows you to define rules that should be applied before rendering a view. Here's an example usage:

```
class MyView(RulesMixin, DetailView):

   def update_logical_rules(self):
      super(MyView, self).update_logical_rules()
      self.add_logical_rule({
         'name': 'user_can_edit_mymodel',
         'param_callbacks': [
            ('object', 'get_object'),
            ('user', 'get_request_user')
```

```
        ]
    })
```

`param_callbacks` are our technique for getting the parameters for your rule. These are assumed to be methods on your class. `get_request_user()` is defined in RuleMixin since it's so common. `get_object()` is a method on the DetailView class.

Rule dictionaries can have other properties, like `redirect_url` and `response_callback`. If `redirect_url` is defined, then the view will return an `HttpResponseRedirect` to that URL. If `response_callback` is defined, then the view will return the result of that method.

Messaging integration is possible with `message` and `message_level` options.

Finally, we've added two commonly used rules. As an optional substitute for `login_required`, we have `user_is_authenticated` and to test a generic expression, we have `evaluate_expression`.

CHAPTER 2

Tutorial

Some basic examples can be found under *Usage*, but here is a more extended walkthrough...

2.1 Views

Often, we'll have several views that share common permissions. This is pretty easy to handle with mixins:

```
class TeamLevelMixin(RulesMixin):

        def update_logical_rules(self):
                super(TeamLevelMixin, self).update_logical_rules()
                self.add_logical_rule({
                        'name': 'user_is_on_team',
                        'param_callbacks': [
                                ('team', 'get_object'),
                                ('user', 'get_request_user')
                        ]
                })
```

Above, I've created a general mixin that can be used wherever I need to use group permissions. This assumes I have a `Team` model and a rule called `user_is_on_team` defined. The `get_request_user` is built in to `RulesMixin` and `get_object` is a custom method that will be provided by `DetailView` below.

Now I'll use that mixin in a few other views:

```
class TeamDetail(TeamLevelMixin, DetailView):
        template_name = 'teams/detail.html'
        model = Team
        context_object_name = 'team'

class TeamPlayers(TeamLevelMixin, TeamDetail):
        template_name = 'team/players.html'
```

2.2 Templates

Great, but what about something more complicated? What if some users can delete other players? That probably means we'll need a delete button in the template and a view to remove them. That's where the template tag comes in.

Here's an example using `teams/players.html`:

```
{% load logical_rules_tags %}
<html>
        <body>
                <h1>Players</h1>
                <table>
                        <tr>
                                <th>First Name</th>
                                <th>Last Name</th>
                                <th>Actions</th>
                        </tr>
                        {% for player in team.players %}
                                <tr>
                                        <td>{{ player.first_name }}</td>
                                        <td>{{ player.last_name }}</td>
                                        <td>
                                                <a href="{% url 'player_profile_view' player %}">Pro
                                                {% testrule user_can_admin_team team request.user %}
                                                        <a href="{% url 'remove_player_from_team' pla
                                                {% endtestrule %}
                                        </td>
                                </tr>
                        {% endfor %}
                </table>
        </body>
</html>
```

2.3 Rules

As you've probably figured out, there are two rules defined here: `user_is_on_team` and `user_can_admin_team`:

```
import logical_rules

def user_is_on_team(team, user):
        """ Is there a player object for this user and this team """
        try:
                player = Player.objects.get(team=team, user=user)
                return True
        except Player.DoesNotExist:
                return False
logical_rules.site.register("user_is_on_team", user_is_on_team)

def user_can_admin_team(team, user):
        """ Is this player the captain? """
        return team.captain.user == user
logical_rules.site.register("user_can_admin_team", user_can_admin_team)
```

For giggles, let's add another, `team_is_published`. Here's how your updated *rules.py* file might look:

```
...

def team_is_published(team):
        """ A team is only published if they're in good standing """
        return team.in_good_standing
logical_rules.site.register("team_is_published", team_is_published)
```

Since this is all written in python, you could easily nest rules and add much more logic here.

2.4 Multiple Rules

Above, we still have to create the view that handles deletion. This view will extend TeamLevelMixin, but also needs another rule to be sure the user is the captain of the team. So, we'll extend TeamLevelMixin and update the logical rules on the view:

```
class TeamLevelMixin(RulesMixin):
        """
                New definition of this mixin that tests if the team is published
        """
        def update_logical_rules(self):
                super(TeamLevelMixin, self).update_logical_rules()
                self.add_logical_rule({
                        'name': 'team_is_published',
                        'param_callbacks': [
                                ('team', 'get_object')
                        ],
                        'response_callback': "redirect_to_reg"
                })
                self.add_logical_rule({
                        'name': 'user_is_on_team',
                        'param_callbacks': [
                                ('team', 'get_object'),
                                ('user', 'get_request_user')
                        ]
                })

class DeletePlayer(DeleteView):
        template_name = 'teams/delete_player.html'
        model = Player
        context_object_name = 'player'

        def update_logical_rules(self):
                super(DeletePlayer, self).update_logical_rules()
                self.add_logical_rule({
                        'name': 'team_is_published',
                        'param_callbacks': [
                                ('team', 'get_team')
                        ],
                        'response_callback': "redirect_to_reg"
                })
                self.add_logical_rule({
                        'name': 'user_can_admin_team',
                        'param_callbacks': [
                                ('team', 'get_team'),
                                ('user', 'get_request_user')
                        ],
                })

        def get_team(self):
                """ Needed as a param callback now """
                return self.get_object.team;

        def redirect_to_reg(self)
                """
```

```
            Redirects the captain to the registration page so the team can
            be in good standing again
    """
    return HttpResponseRedirect(reverse('teams.registration', args=(self.get_object(),)))
```

This example uses the `response_callback` parameter to redirect the user to the registration page if the team is not in good standing. We updated `TeamLevelMixin` to use the `team_is_published` rule and redirect if they aren't. Rules are executed in the order they are added to the view, so `team_is_published` will be executed first and could result in a redirect.

We couldn't use `TeamLevelMixin` for the `DeletePlayer` view because the parameters would have been pointing to the `get_object` method and that would have returned a `Player`. Of course, we could simply create a `get_team` method that wraps `get_object` in the `TeamDetail` class and then we wouldn't have to add it again. Lots of ways to approach this.

This is a lot of code for one or two views, but the real power of the rules is that they can be used everywhere and when you have 10 views that use the same rule logic and then need that same logic in your templates, this can be very handy. Performance can become an issue, so you may want to include some caching in your rules where possible.

CHAPTER 3

Installation

Use pip to install from PyPI:

```
pip install django-logical-rules
```

Add `storages` to your settings.py file:

```
INSTALLED_APPS = (
    ...
    'django-logical-rules',
    ...
)
```

If you want to use the messaging features, install Django messages framework.

CHAPTER 4

rules.py

Simply add **rules.py** to your app and use them throughout your app. Here's what a rule looks like:

```
import logical_rules

def user_can_edit_mymodel(object, user):
        """
                Confirms a user can edit a specific model
                ...owners only!
        """
        return object.owner == user
logical_rules.site.register("user_can_edit_mymodel", user_can_edit_mymodel)
```

CHAPTER 5

Configuration

To include your models in the registry you will need to do run the autodiscover, a bit like django.contrib.admin (I generally put this in my **urls.py**):

```
import logical_rules
logical_rules.autodiscover()
```

CHAPTER 6

Performance

Performance varies mainly around how you write your rules. Often it's a good idea to use caching in your rules when a permission isn't changing frequently.

CHAPTER 7

Contributing

Think this needs something else? To contribute to `django-logical-rules` create a fork on Bitbucket. Clone your fork, make some changes, and submit a pull request.

Bugs are great contributions too! Feel free to add an issue on Bitbucket: