django-rules-light Documentation

Release 0.1.3

James Pic

Mar 05, 2018

Contents

This is a simple alternative to django-rules. The core difference is that it uses as registry that can be modified on runtime, instead of database models.

One of the goal is to enable developpers of external apps to make rules, depend on it, while allowing a project to override rules.

Example `your_app/rules_light_registry.py`:

```
# Everybody can read a blog post (for now!):
rules_light.registry['blog.post.read'] = True

# Require authentication to create a blog post, using a shortcut:
rules_light.registry['blog.post.create'] = rules_light.is_authenticated

def is_staff_or_mine(user, rule, obj):
    return user.is_staff or obj.author == user

# But others shouldn't mess with my posts !
rules_light.registry['blog.post.update'] = is_staff_or_mine
rules_light.registry['blog.post.delete'] = is_staff_or_mine
```

Example `your_app/views.py`:

```
@rules_light.class_decorator
class PostDetailView(generic.DetailView):
    model = Post

@rules_light.class_decorator
class PostCreateView(generic.CreateView):
    model = Post

@rules_light.class_decorator
class PostUpdateView(generic.UpdateView):
    model = Post

@rules_light.class_decorator
class PostDeleteView(generic.DeleteView):
    model = Post
```

You might want to read the tutorial for more.

CHAPTER 1

What's the catch ?

The catch is that this approach does not offer any feature to get secure querysets.

This means that the developper has to:

- think about security when making querysets,
- override eventual external app ListViews,

CHAPTER 2

Requirements

- Python 2.7+ (Python 3 supported)
- Django 1.4+

CHAPTER 3

Quick Install

- Install module: `pip install django-rules-light`,
- Add to `settings.INSTALLED_APPS`: `rules_light`,
- Add in `settings.MIDDLEWARE_CLASSES`: `rules_light.middleware.Middleware`,
- Add in `urls.py`: `rules_light.autodiscover()` if you have `admin.autodiscover()` in there too (Django < 1.7),

You might want to read the tutorial.

There is also a lot of documentation, from the core to the tools, including pointers to debug, log and test your security.

CHAPTER 4

Contributing

Run tests with the tox command. Documented patches passing all tests have more chances getting merged in, see community guidelines for details.

CHAPTER 5

Resources

You could subscribe to the mailing list ask questions or just be informed of package updates.

- Mailing list graciously hosted by Google
- Git graciously hosted by GitHub,
- Documentation graciously hosted by RTFD,
- Package graciously hosted by PyPi,
- Continuous integration graciously hosted by Travis-ci

Contents:

5.1 Tutorial

5.1.1 Install

Either install the last release:

```
pip install django-rules-light
```

Either install a development version:

```
pip install -e git+https://github.com/yourlabs/django-rules-light.git#egg=django-
↪rules-light
```

That should be enough to work with the registry.

Middleware

To enable the middleware that processes `rules_light.Denied` exception, add to `setings.MIDDLEWARE_CLASSES`:

```
MIDDLEWARE_CLASSES = (
    # ...
    'rules_light.middleware.Middleware',
)
```

See *docs on middleware* for more details.

Autodiscovery

To enable autodiscovery of rules in the various apps installed in your project, add to `urls.py` (as early as possible):

```
import rules_light
rules_light.autodiscover()
```

See *docs on registry* for more details.

Logging

To enable logging, add a `rules_light` logger for example:

```
LOGGING = {
    # ...
    'handlers': {
        # ...
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
        },
    },
    'loggers': {
        'rules_light': {
            'handlers': ['console'],
            'propagate': True,
            'level': 'DEBUG',
        }
    }
}
```

See *docs on logging* for more details on logging.

Debug view

Add to `settings.INSTALLED_APPS`:

```
INSTALLED_APPS = (
    'rules_light',
    # ....
)
```

Then the view should be usable, install it as such:

```
url(r'^rules/', include('rules_light.urls')),
```

See *docs on debugging* for more details on debugging rules.

5.1.2 Creating Rules

Declare rules

Declaring rules consist of filling up the `rules_light.registry` dict. This dict uses rule "names" as keys, ie. `do_something`, `some_app.some_model.create`, etc, etc ... For values, it can use booleans:

```
# Enable read for everybody
rules_light.registry['your_app.your_model.read'] = True

# Disable delete for everybody
rules_light.registry['your_app.your_model.delete'] = False
```

Optionnaly, use the Python dict method `setdefault()` in default rules. For example:

```
# Only allow everybody if another (project-specific) callback was not set
rules_light.registry.setdefault('your_app.your_model.read', True)
```

It can also use callbacks:

```
def your_custom_rule(user, rule_name, model, *args, **kwargs):
    if user in model.your_custom_stuff:
        return True  # Allow user !

rules_light.registry['app.model.read'] = your_custom_rule
```

See *docs on registry* for more details.

Mix rules, DRY security

Callbacks may also be used to decorate each other, using `rules_light.make_decorator()` will transform a simple rule callback, into a rule callback that can also be used as decorator for another callback.

Just decorate a callback with `make_decorator()` to make it reusable as decorator:

```
@rules_light.make_decorator
def some_condition(user, rule, *args, **kwargs):
    # do stuff

rules_light.registry.setdefault('your_app.your_model.create', some_condition)

@some_condition
def extra_condition(user, rule, *args, **kwargs):
    # do extra stuff

rules_light.registry.setdefault('your_app.your_model.update', extra_condition)
```

This will cause `some_condition()` to be evaluated first, and if it passes, `extra_condition()` will be evaluated to, for the update rule.

See docs on decorator for more details.

5.1.3 Using rules

The rule registry is in charge of using rules, using the `run()` method. It should return True or False.

Run

For example with this:

```
def some_condition(user, rulename, *args, **kwargs):
    # ...

rules_light.registry['your_app.your_model.create'] = some_condition
```

Doing:

```
rules_light.run(request.user, 'your_app.your_model.create')
```

Will call:

```
some_condition(request.user, 'your_app.your_model.create')
```

Kwargs are forwarded, for example:

```
rules_light.run(request.user, 'your_app.your_model.create',
    with_widget=request.GET['widget'])
```

Will call:

```
some_condition(request.user, 'your_app.your_model.create',
    with_widget=request.GET['widget'])
```

See *docs on registry* for more details.

Require

The `require()` method is useful too, it does the same as `run()` except that it will raise `rules_light.Denied`. This will block the request process and will be catched by the middleware if installed.

See *docs on registry* for more details.

Decorator

You can decorate a class based view as such:

```
@rules_light.class_decorator
class SomeCreateView(views.CreateView):
    model=SomeModel
```

This will automatically require `'some_app.some_model.create'`.

See *docs on class decorator* for more usages of the decorator.

Template

In templates, you can run rules using '{% rule %}' templatetag.

Usage:

```
{% rule rule_name [args] [kwargs] as var_name %}
```

This is an example from the test project:

```
{% load rules_light_tags %}

<ul>
{% for user in object_list %}
    {% rule 'auth.user.read' user as can_read %}
    {% rule 'auth.user.update' user as can_update %}

    <li>
    <a href="{% url 'auth_user_detail' user.username %}">{{ user }} (has perm: {{ can_
↪read|yesno:'Yes,No' }})</a>
    <a href="{% url 'auth_user_update' user.username %}">update (has perm: {{ can_
↪update|yesno:'Yes,No'}})</a>
    </li>
{% endfor %}
</ul>
```

5.1.4 Tips and tricks

Override rules

If your project wants to change the behaviour of `your_app` to allows users to create models and edit the models they have created, you could add after `rules_light.autodiscover()`:

```
def my_model_or_staff(user, rulename, obj):
    return user.is_staff or user == obj.author

rules_light.registry['your_app.your_model.create'] = True
rules_light.registry['your_app.your_model.update'] = my_model_or_staff
rules_light.registry['your_app.your_model.delete'] = my_model_or_staff
```

As you can see, a project can **completely** change the security logic of an app, which should enpower creative django developers hehe ...

See *docs on registry* for more details.

Take a shortcut

django-rules-light comes with a predefined `is_staff` rule which you could use in `your_app/rules_light_registry.py`:

```
import rules_light

# Allow all users to see your_model
rules_light.registry.setdefault('your_app.your_model.read', True)

# Allow admins to create and edit models
rules_light.registry.setdefault('your_app.your_model.create', rules_light.is_staff)
rules_light.registry.setdefault('your_app.your_model.update', rules_light.is_staff)
rules_light.registry.setdefault('your_app.your_model.delete', rules_light.is_staff)
```

See *docs on shortcuts*.

Test security

See *security testing docs*.

5.2 Rule registry

5.2.1 API

The rule registry is in charge of keeping and executing security rules.

It is the core of this app, everything else is optionnal.

This module provides a variable, `registry`, which is just a module-level, default RuleRegistry instance.

A rule can be a callback or a variable that will be evaluated as bool.

class `rules_light.registry.`**RuleRegistry**
Dict subclass to manage rules.

logger The standard logging logger instance to use.

as_text(*user*, *name*, **args*, ***kwargs*)
Format a rule to be human readable for logging

require(*user*, *name*, **args*, ***kwargs*)
Run a rule, raise `rules_light.Denied` if returned False.

Log denials with warn-level.

run(*user*, *name*, **args*, ***kwargs*)
Run a rule, return True if whatever it returns evaluates to True.

Also logs calls with the info-level.

`rules_light.registry.`**require**(*user*, *name*, **args*, ***kwargs*)
Proxy `rules_light.registry.require()`.

`rules_light.registry.`**run**(*user*, *name*, **args*, ***kwargs*)
Proxy `rules_light.registry.run()`.

`rules_light.registry.`**autodiscover**()
Check all apps in INSTALLED_APPS for stuff related to rules_light.

For each app, autodiscover imports `app.rules_light_registry` if available, resulting in execution of `rules_light.registry[...] = ...` statements in that module, filling registry.

Consider a standard app called 'cities_light' with such a structure:

```
cities_light/
    __init__.py
    models.py
    urls.py
    views.py
    rules_light_registry.py
```

With such a rules_light_registry.py:

```
import rules_light

rules_light.register('cities_light.city.read', True)
rules_light.register('cities_light.city.update',
    lambda user, rulename, country: user.is_staff)
```

When autodiscover() imports cities_light.rules_light_registry, both *'cities_light.city.read'* and *'cities_light.city.update'* will be registered.

5.2.2 Examples

```
import rules_light

rules_light.registry['auth.user.read'] = True
rules_light.registry['auth.user.update'] = lambda user, *args: user.is_staff
```

Even django-rules-light's view uses a permission, it is registered in `rules_light/rules_light_registry.py` and thus is picked up by `rules_light.autodiscover()`:

```
from __future__ import unicode_literals

import rules_light


rules_light.registry['rules_light.rule.read'] = rules_light.is_staff
```

Of course, you could use any callable instead of the lambda function.

5.3 Class decorator

5.3.1 API

class rules_light.class_decorator.**class_decorator**

Can be used to secure class based views.

If the view has `model=YourModel`, it will support:

- `CreateView`, it will decorate `get_form()`, to run `rules_light.require('yourapp.yourmodel.create')`,
- `UpdateView`, it will decorate `get_object()`, to run `rules_light.require('yourapp.yourmodel.update', obj)`,
- `DeleteView`, it will decorate `get_object()`, to run `rules_light.require('yourapp.yourmodel.delete', obj)`,
- `DetailView`, it will decorate `get_object()`, to run `rules_light.require('yourapp.yourmodel.read', obj)`,
- others views, if the rule name is specified in the decorator for example `@class_decorator('some_rule')`, then it will decorate `dispatch()`,
- Else it raises an exception.

5.3.2 Examples

5.4 Middleware

The role of the middleware is to present a user friendly error page when a rule denied process of the request by raising `Denied`.

class `rules_light.middleware.`**Middleware**

Install this middleware by adding *rules_light.middleware.Middleware'* to `settings.MIDDLEWARE_CLASSES`.

process_exception(*request*, *exception*)

Render `rules_light/exception.html` when a `Denied` exception was raised.

5.4.1 Template

```
{% extends 'rules_light/base.html' %}

{% load i18n %}

{% block body %}
        <div class='rules_light' >
                {% trans 'You do not have permission to do that.' %}

                {% if settings.LOGIN_URL %}
                <a href="{{ settings.LOGIN_URL }}?next={{ request.path_info|urlencode␣
↪}}">{% trans 'Try logging in' %} {% if request.user.is_authenticated %}{% trans
↪'with other credentials' %}{% endif %}</a>
                {% endif %}
        </div>
{% endblock %}
```

5.5 Shortcuts

It is trivial to take shortcuts because the rule registry is a simple dict.

You can reuse your rules several times in standard python:

```
def my_model_or_is_staff(user, rule, model, obj=None):
    return user.is_staff or (obj and obj.author == user)

rules_light.registry.setdefault('your_app.your_model.create',
    my_model_or_is_staff)
rules_light.registry.setdefault('your_app.your_model.update',
    my_model_or_is_staff)
rules_light.registry.setdefault('your_app.your_model.delete',
    my_model_or_is_staff)
```

This module provides some shortcut(s). Shortcuts are also usable as decorators too (see `make_decorator`):

```
@rules_light.is_authenticated
def my_book(user, rule, book):
    return book.author == user
```

(continues on next page)

(continued from previous page)

```
rules_light.registry.setdefault('your_app.your_model.update', my_book)
```

5.6 Logging

Everything is logged in the `rules_light` logger:

- rule registered is logged with `DEBUG` level,
- rule `run()` is logged with `INFO` level,
- `require()` failure is logged with `WARN` level.

5.6.1 Install

Example `settings.LOGGING` that will display all logged events in the console, as well as denials in malicious.log.

See http://docs.djangoproject.com/en/dev/topics/logging for more details on how to customize your logging configuration.

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'filters': {
        'require_debug_false': {
            '()': 'django.utils.log.RequireDebugFalse'
        }
    },
    'handlers': {
        'mail_admins': {
            'level': 'ERROR',
            'filters': ['require_debug_false'],
            'class': 'django.utils.log.AdminEmailHandler'
        },
        'console': {
            'level': 'DEBUG',
            'class': 'logging.StreamHandler',
        },
        'malicious': {
            'level': 'WARN',
            'class': 'logging.FileHandler',
            'filename': 'malicious.log',
        },
    },
    'loggers': {
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': True,
        },
        'rules_light': {
            'handlers': ['console', 'malicious'],
            'propagate': True,
            'level': 'DEBUG',
```

(continues on next page)

(continued from previous page)

```
        },
    }
}
```

5.7 Debugging

Two tools are provided to debug issues with your registry:

- the *logger logs* everything (and it likes to log malicious users too),
- the url provides a live rule registry browser (see below).

As usual, resort to `ipdb`, for example in `rules_light.RuleRegistry.run()` place:

```
import ipdb; ipdb.set_trace()
```

5.7.1 The registry browser

class rules_light.views.**RegistryView**(***kwargs*)

Expose the rule registry for debug purposes.

Install it as such:

```
url(r'^rules/$', RegistryView.as_view(), name='rules_light_registry'),
```

Or just:

```
url(r'^rules/', include('rules_light.urls')),
```

Note: view requires `'rules_light.rule.read'` which is enabled for admins by default.

Constructor. Called in the URLconf; can contain helpful extra keyword arguments, and other things.

get_context_data()

Add the registry to the context.

5.8 Security testing

It is important to test your security. Here is an example:

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index