django-prbac Documentation

Release 1.0.1

Dimagi

Sep 10, 2020

Contents

The django-prbac package provides the basic components of parameterized role-based access control for Django. The entirety of the system is contained in two classes of objects:

1. `Role` (representing users, groups, capabilities, and privileges)
2. `Grant` (representing memberships, containment, and permissions)

If you are familiar with role-based access control (RBAC) then this is a minor, though foundational, enhancement to the non-parameterized version. It will often make the role graph much smaller and simpler, and will definitely allow much more powerful administration of the graph.

Contents:

CHAPTER 1

Installation and Set Up

There are no special steps to set up *django_prbac* beyond those for any Django app.

The first step is to install the Python package via *pip* or *easy_install*:

```
$ pip install django-prbac
```

Then add *django_prbac* to the *INSTALLED_APPS* in your settings module (this will be *settings.py* for default projects):

```
# in setting.py
INSTALLED_APPS = [
    ...
    'django_prbac',
]
```

Set up the database by running the migrations:

```
$ python manage.py migrate django_prbac
```

If you wish, you can run the tests to check the health of your installation. This will not modify any of your data:

```
$ python manage.py test django_prbac --settings django_prbac.mock_settings
```

CHAPTER 2

Django PRBAC Tutorial

Models of class **|Role|** represent capabilities, which may intuitively map to users, privileges, groups, and collections of privileges.

Models of class **|Grant|** represent adding a user to a group, including a group in another group, and granting privileges to a user or group.

2.1 Users

This library does not replace or modify the Django user system – too many projects muck around with that, so it is safer and more flexible to leave it alone. Instead, you may give each user a corresponding **|Role|**:

```
>>> from django.contrib.auth.models import User
>>> for user in User.objects.all():
        Role.objects.create(
            name=user.username,
            slug=user.username,
            description='Role for django user: %s' % user.username
        )
```

This is very easy to automate with triggers or via the *UserProfile* feature of Django.

2.2 Privileges

A privilege is an actual thing that a user may do in the system. It is up to you to decide what these are and give them meaningful names and descriptions. For example, perhaps there is a granular permission of "may view reports":

```
>>> may_view_reports = Role.objects.create(name='may_view_reports', slug='may_view_
↪reports', description='May view reports')

>>> biyeun = Role.objects.get(name='biyeun')
```

(continues on next page)

(continued from previous page)

```
>>> kenn = Role.objects.get(name='kenn')

>>> Grant.objects.create(from_role=biyeun, to_role=may_view_reports)

>>> biyeun.has_privilege(may_view_reports)
True

>>> kenn.has_privilege(may_view_reports)
False
```

All of this is normal for RBAC (without parameterization) but with PRBAC we can make this privilege more granular:

```
>>> may_view_report = Role.objects.create(name='may_view_report', slug='may_view_
↪report', parameters=set(['report_name']))

>>> Grant.objects.create(from_role=biyeun, to_role=may_view_report, assignment={
↪'report_name': 'active_users'})
>>> Grant.objects.create(from_role=kenn, to_role=may_view_report, assignment={'report_
↪name': 'submissions'})

>>> biyeun.has_privilege(may_view_report.instantiate({'report_name': 'active_users'}))
True

>>> biyeun.has_privilege(may_view_report.instantiate({'report_name': 'submissions'}))
False

>>> kenn.has_privilege(may_view_report.instantiate({'report_name': 'active_users'}))
False

>>> kenn.has_privilege(may_view_report.instantiate({'report_name': 'submissions'}))
True
```

2.3 Groups

A group of users may be represented as a |Role| as well:

```
>>> dimagineers = Role.objects.create(name='dimagineers', slug='dimagineers',␣
↪description='Dimagi Engineers')

>>> Grant.objects.create(from_role=kenn, to_role=dimagineers)
>>> Grant.objects.create(from_role=biyeun, to_role=dimagineers)
```

Now both *kenn* and *biyeun* are members of role *dimagineers*.

```
>>> kenn.has_privilege(dimagineers)
True
>>> biyeun.has_privilege(dimagineers)
True
```

But groups can also be useful when parameterized, for granting a variety of parameterized privileges to a group of people.

```
>>> may_edit_report = Role.objects.create(
        name='may_edit_report',
```

(continues on next page)

(continued from previous page)

```
        description='May edit report',
        slug='may_edit_report',
        parameters=set(['report_name']),
    )
```

```
>>> report_superusers = Role.objects.create(
        name='report_superusers',
        description='Report Superusers',
        slug='report_superusers',
        parameters=set(['report_name']),
    )
```

```
>>> Grant.objects.create(from_role=report_superusers, to_role=may_edit_report)
>>> Grant.objects.create(from_role=report_superusers, to_role=may_view_report)
>>> Grant.objects.create(
        from_role=kenn,
        to_role=report_superusers,
        assignment={'report_name': 'dashboard'},
    )
```

```
>>> kenn.has_privilege(may_view_report.instantiate({'report_name': 'dashboard'}))
True
>>> kenn.has_privilege(may_edit_report.instantiate({'report_name': 'dashboard'}))
True
```

CHAPTER 3

django_prbac Package

3.1 django_prbac Package

3.2 admin Module

3.3 arbitrary Module

3.4 decorators Module

3.5 exceptions Module

exception django_prbac.exceptions.**PermissionDenied**
Bases: Exception

3.6 fields Module

class django_prbac.fields.**StringListField**(*verbose_name=None*, *name=None*, *primary_key=False*, *max_length=None*, *unique=False*, *blank=False*, *null=False*, *db_index=False*, *rel=None*, *default=<class 'django.db.models.fields.NOT_PROVIDED'>*, *editable=True*, *serialize=True*, *unique_for_date=None*, *unique_for_month=None*, *unique_for_year=None*, *choices=None*, *help_text=''*, *db_column=None*, *db_tablespace=None*, *auto_created=False*, *validators=()*, *error_messages=None*)

Bases: django.db.models.fields.TextField

A Django field for lists of strings

formfield(***kwargs*)
The default form field is a StringListFormField.

from_db_value(*value*, *expression*, *connection*, **args*, ***kwargs*)

get_prep_value(*value*)
Converts the value, which must be a string list, to a comma-separated string, quoted appropriately. This format is private to the field type so it is not exposed for customization or any such thing.

is_string_list(*value*)

to_python(*value*)
Best-effort conversion of "any value" to a string list.

It does not try that hard, because curious values probably indicate a mistake and we should fail early.

value_to_string(*obj*)
Return a string value of this field from the passed obj. This is used by the serialization framework.

class django_prbac.fields.**StringSetField**(*verbose_name=None*, *name=None*, *primary_key=False*, *max_length=None*, *unique=False*, *blank=False*, *null=False*, *db_index=False*, *rel=None*, *default=<class 'django.db.models.fields.NOT_PROVIDED'>*, *editable=True*, *serialize=True*, *unique_for_date=None*, *unique_for_month=None*, *unique_for_year=None*, *choices=None*, *help_text=''*, *db_column=None*, *db_tablespace=None*, *auto_created=False*, *validators=()*, *error_messages=None*)

Bases: *django_prbac.fields.StringListField*

A Django field for set of strings.

from_db_value(*value*, *expression*, *connection*, **args*, ***kwargs*)

get_prep_value(*value*)
Converts the value, which must be a string list, to a comma-separated string, quoted appropriately. This format is private to the field type so it is not exposed for customization or any such thing.

is_string_set(*value*)

to_python(*value*)
Best-effort conversion of "any value" to a string set. Mostly strict, but a bit lenient to allow lists to be passed in by form fields.

value_to_string(*obj*)
Return a string value of this field from the passed obj. This is used by the serialization framework.

3.7 mock_settings Module

Settings just for running the django-prbac tests or checking out the admin site.

3.8 models Module

3.9 urls Module

3.10 Subpackages

3.10.1 migrations Package

0001_initial Module

3.10.2 tests Package

tests Package

test_decorators Module

test_fields Module

class django_prbac.tests.test_fields.**TestStringListField**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Test suite for django_prbac.fields.StringListField

test_get_prep_value_convert()

test_is_string_list()

test_to_python_already_done()

test_to_python_convert()

class django_prbac.tests.test_fields.**TestStringSetField**(*methodName='runTest'*)
Bases: django.test.testcases.TestCase

Test suite for django_prbac.fields.StringSetField

test_get_prep_value_convert()

test_is_string_set()

test_to_python_already_done()

test_to_python_convert()

test_models Module

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index