django-object-authority Documentation

Release 0.0.5

Tomeu Canyelles

Jun 19, 2017

Contents

Django-object-authority provides an authentication backend for Django applications. Furthermore expose a set of features for filtering queryset.

Documentation:

CHAPTER 1

Overview

Django provides an authentication system to authorize users to create, modify or delete objects of a type, that is, the user can perform this action on any element of the class in which it has such permissions. This package extends these permissions and adds read permissions.

The main function of the same is to control the access on specific elements for a concrete action.

Features

1. New authentication backend for django apps.
2. New authentication backend for django rest framework.
3. Mechanism to auto-register object permissions.
4. Mixin to use in list views that filter your queryset according an authorization filter.
5. Base filter makes you easier filter your queryset according user's permissions.
6. Command to create custom permission of application and/or specific models.

Inspiration

This package is base on the default implementation of Django authentication.

There are some other plugins like django-object-permission, django-guardian, django-authority. Another framework has inspired in this development has been dry-rest-permissions.

CHAPTER 2

Installation

You can install it with `pip`:

```
pip install django-object-authority
```

...or clone the project from github:

```
git clone git@github.com:APSL/django-object-authority.git
```

CHAPTER 3

Configuration

There are two modes of usage the library:

1. As a *mixins* that provide you a set of features.
2. Application that autodiscover your objects permissions to apply them to your _Django_ application.

As third party application

First of all you should add *django_object_authority* to you *INSTALLED_APPS* settings.

```
INSTALLED_APPS = (
    ...
    'django_object_authority',
)
```

Is needed override *AUTHENTICATION_BACKENDS* setting to add *ObjectAuthorityBackend*.

```
AUTHENTICATION_BACKENDS = [
    'django.contrib.auth.backends.ModelBackend',
    'django_object_authority.backends.ObjectAuthorityBackend',
]
```

For each model you want to custom the permission level is needed define a *authorizations.py* file and register the permission *class*.

```
@register(MyModel)
class MyModelAuthority(BaseUserObjectAuthorization):

    def has_add_permission(self, user, obj):
        return obj.owner == user
```

If you don't override all *BaseUserObjectAuthorization* defined methods. The default behaviour is defined as a setting variable [*Settings* section].

BaseObjectAuthorization only implements *has_object_permission* method which check the object permission as default resource.

As *mixins*

You can use it only installing the package [*Installation* section] and include the mixin in your views.

```
from django.views.generic import ListView
from django_object_authority.mixins import AuthorizationMixin

class MyListView(AuthorizationMixin, ListView):
    ...
    authorization_filter_class = MyAuthorityFilter
    ...
```

CHAPTER 4

Settings

Settings for Django object authority are all namespaced in the *OBJECT_AUTHORITY* setting. For example your project's *settings.py* file might look like this:

```
OBJECT_AUTHORITY_AUTHORIZE_BY_DEFAULT = True
```

AUTHORIZE_BY_DEFAULT Boolean value that define the default behaviour for authorization user object action.

Default `True`

AUTHORIZE_ONLY_OBJECTS Boolean value for validate *class* permission before validate *object* permissions.

Default: `False`

FULL_PERMISSION_FOR_STAFF Boolean value to allow all access if the user requester is **staff**.

Default: `False`

FULL_PERMISSION_FOR_SUPERUSERS Boolean value to allow all access if the user requester is **superuser**.

Default: `True`

CHAPTER 5

Management commands

The library provide a *Django* command line to create or update new permission `django.contrib.auth.Permission` instances.

How to use it

By default the command creates a new `Permission` if it does not exist. Otherwise it will update it.

```
python manage.py create_update_permissions -n my_custom_permission_1 my_custom_
↪permission_2
```

This execution will create/update all permissions defined on `-n` parameter for each non auto-created*[0] model of each registered application.

If you want to filter the applications and/or models for creation/update you can use `-a` and/or `-m` respective.

Use case

This feature could be very useful to configure your object permission according those *Django* permissions.

```
@register(MyModel)
class MyModelAuthority(ObjectAuthorization):

    def has_view_permission(self, user, obj):
        codename = '{}.{}_{}'.format(obj._meta.app_label, 'my_custom_permission_1',␣
↪obj._meta.model_name))
        return codename in user.get_all_permissions()
```

[0] All models that django creates as a many to many relationship result.

CHAPTER 6

User Guide

1. Include `django-object-authority` in your requirements file or install via *pip* [*Installation* section].

2. Add `django_object_authority` to the `INSTALLED_APPS` and add `django_object_authority.backends.ObjectAuthorityBackend` to the default django `AUTHENTICATION_BACKENDS` [*Configuration* section].

3. Define and register the permissions for your models [*As third party application* section].

```
# authorization.py
from django_object_authority import register
from django_object_authority import BaseUserObjectAuthorization

from .models import Book, Article


@register(Book)
class BookAuthority(BaseUserObjectAuthorization):

    def has_change_permission(self, user, obj):
        return book.author == user

    def has_delete_permission(self, user, obj):
        return False


@register(Article)
class ArticleTeamAuthority(BaseUserObjectAuthorization):

    def has_view_permission(self, user, obj):
        return obj.book.team.filter(user=user).exists()

    def has_change_permisssion(self, user, obj):
        return self.has_view_permission(user, obj) or obj.owner == user

    def has_delete_permission(self, user, obj):
```

```
        return obj.owner == user
```

4. Define your CRUD base views for check user permissions.

```
# base_views.py
from django.core.exceptions import PermissionDenied
from django.db import models
from django.views import generic


class ViewMixin(object):

    def get_codename(self, perm):
        return '{}.{}_{}'.format(self.model._meta.app_label, perm, self.model._meta.
↪model_name)

    def has_view_permission(self, request, obj=None):
        return request.user.has_perm(self.get_codename('view'), obj)


    def has_change_permission(self, request, obj=None):
        return request.user.has_perm(self.get_codename('change'), obj)

    def has_add_permission(self, request):
        return request.user.has_perm(self.get_codename('add'), obj)

    def has_delete_permission(self, request, obj=None):
        return request.user.has_perm(self.get_codename('delete'), obj)

class CreateBaseView(ViewMixin, generic.CreateView):
    ...

    def get(self, request, *args, **kwargs):
        if not self.has_add_permission(self.request, None):
            raise PermissionDenied
        return super(CreateModelView, self).get(request, *args, **kwargs)


class RetrieveBaseView(ViewMixin, generic.DetailView):
    ...

    def get_object(self):
        obj = super(BaseDetailView, self).get_object()
        if not self.has_add_permission(self.request, obj):
            raise PermissionDenied
        return obj


class UpdateBaseView(ViewMixin, generic.UpdateView):
    ...

    def get_object(self):
        obj = super(BaseDetailView, self).get_object()
        if not self.has_change_permission(self.request, obj):
            raise PermissionDenied
        return obj
```

```
class DeleteBaseView(ViewMixin, generic.UpdateView):
    ...

    def get_object(self):
        obj = super(BaseDetailView, self).get_object()
        if not self.has_delete_permission(self.request, obj):
            raise PermissionDenied
        return obj


class ListBaseView(ViewMixin, generic.ListView):
    ...

    def dispatch(self, request, *args, **kwargs):
        if not self.has_view_permission(self.request):
            raise PermissionDenied
        return super(ListModelView, self).dispatch(request, *args, **kwargs)
```

5. Create som custom permission for after filtering [*Management commands* section].

```
python manage.py create_update_permissions -a main -m book -n my_team
```

6. Define your filter class for accurate list according the authorization over each element [*As mixins* section].

```
# authorization_filters.py
class BookAuthorityFilter(AuthorityBaseFilter):
    permission_codes = ('my_team', )

    def filter_by_my_team(self, queryset, user):
        return queryset.filter(team=user.team)
```

7. Custom your list views to filter queryset according the permissions.

```
# views.py
from django_object_authority.mixins import AuthorizationMixin
from .base_views import ListBaseView

class BookListView(AuthorizationMixin, ListBaseView):
    authorization_filter_class = BookAuthorityFilter
```

CHAPTER 7

Indices and tables

- genindex
- modindex
- search