django-role-permissions Documentation

Release 0.1

Filipe Ximenes

Dec 02, 2018

Contents

Contents:

CHAPTER 1

Setup

1.1 Installation

Install from PyPI with `pip`:

```
pip install django-role-permissions
```

1.2 Configuration

Add `rolepermissions` to you `INSTALLED_APPS`

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

CHAPTER 2

Quick Start

Create a `roles.py` file in the same folder as your `settings.py` and two roles:

```
from rolepermissions.roles import AbstractUserRole

class Doctor(AbstractUserRole):
    available_permissions = {
        'create_medical_record': True,
    }

class Nurse(AbstractUserRole):
    available_permissions = {
        'edit_patient_file': True,
    }
```

Add a reference to your roles module to your settings:

```
ROLEPERMISSIONS_MODULE = 'myapplication.roles'
```

When you create a new user, set its role using:

```
>>> from rolepermissions.roles import assign_role
>>> user = User.objects.get(id=1)
>>> assign_role(user, 'doctor')
```

and check its permissions using

```
>>> from rolepermissions.checkers import has_permission
>>>
>>> has_permission(user, 'create_medical_record')
True
>>> has_permission(user, 'edit_patient_file')
False
```

You can also change users permissions:

```
>>> from rolepermissions.permissions import grant_permission, revoke_permission
>>>
>>> revoke_permission(user, 'create_medical_record')
>>> grant_permission(user, 'edit_patient_file')
>>>
>>> has_permission(user, 'create_medical_record')
False
>>> has_permission(user, 'edit_patient_file')
True
```

CHAPTER 3

Roles

3.1 Roles File

Create a `roles.py` file anywhere inside your django project and reference it in the project settings file.

`my_project/roles.py`

```
from rolepermissions.roles import AbstractUserRole

class Doctor(AbstractUserRole):
    available_permissions = {
        'create_medical_record': True,
    }

class Nurse(AbstractUserRole):
    available_permissions = {
        'edit_patient_file': True,
    }
```

`settings.py`

```
ROLEPERMISSIONS_MODULE = 'my_project.roles'
```

Each class that imports `AbstractUserRole` is a role on the project and has a snake case string representation. For example:

```
from rolepermissions.roles import AbstractUserRole

class SystemAdmin(AbstractUserRole):
    available_permissions = {
        'drop_tables': True,
    }
```

will have the string representation: `system_admin`.

3.2 Available Role Permissions

The field `available_permissions` lists what permissions the role can be granted. The boolean referenced on the `available_permissions` dictionary is the default value to the referred permission.

CHAPTER 4

Object permission checkers

4.1 permissions.py file

You can add a `permissions.py` file to each app. This file should contain registered object permission checker functions.

`my_app/permissions.py`

```
from rolepermissions.permissions import register_object_checker
from my_project.roles import SystemAdmin

@register_object_checker()
def access_clinic(role, user, clinic):
    if role == SystemAdmin:
        return True

    if user.clinic == clinic:
        return True

    return False
```

when requested the object permission checker will receive the role of the user, the user and the object being verified and should return `True` if the permission is granted.

4.2 Checking object permission

Use the *has_object_permission* method to check for object permissions.

CHAPTER 5

Utils

5.1 Shortcuts

get_user_roles(*user*)

Returns the user's roles.

```
from rolepermissions.roles import get_user_roles

role = get_user_roles(user)
```

assign_role(*user*, *role*)

Assigns a role to the user. Role parameter can be passed as string or role class object.

```
from rolepermissions.roles import assign_role

assign_role(user, 'doctor')
```

remove_role(*user*, *role*)

Removes a role from a user. Role parameter can be passed as string or role class object.

```
from rolepermissions.roles import remove_role

remove_role(user, 'doctor')
```

WARNING: Any permissions that were explicitly granted to the user that are also defined to be granted by this role will be revoked when this role is revoked.

Example:

```
>>> class Doctor(AbstractUserRole):
...     available_permissions = {
...         "operate": False,
```

(continues on next page)

(continued from previous page)

```
...     }
>>>
>>> class Surgeon(AbstractUserRole):
...     available_permissions = {
...         "operate": True,
...     }
>>>
>>> grant_permission(user, "operate")
>>> remove_role(user, Surgeon)
>>>
>>> has_permission(user, "operate")
False
```

In the example, the user no longer has the `"operate"` permission, even though it was set explicitly before the `Surgeon` role was removed.

`clear_roles`(*user*)

Clear all of a user's roles.

```
from rolepermissions.roles import clear_roles

clear_roles(user)
```

`available_perm_status`(*user*)

Returns a dictionary containing all permissions available across all the specified user's roles. Note that if a permission is granted in one role, it overrides any permissions set to `False` in other roles. Permissions are the keys of the dictionary, and values are `True` or `False` indicating if the permission is granted or not.

```
from rolepermissions.permissions import available_perm_status

permissions = available_perm_status(user)

if permissions['create_medical_record']:
    print('user can create medical record')
```

`grant_permission`(*user*, *permission_name*)

Grants a permission to a user. Will raise a `RolePermissionScopeException` for a permission that is not listed in the user's roles' `available_permissions`.

```
from rolepermissions.permissions import grant_permission

grant_permission(user, 'create_medical_record')
```

`revoke_permission`(*user*, *permission_name*)

Revokes a permission from a user. Will raise a `RolePermissionScopeException` for a permission that is not listed in the user's roles' `available_permissions`.

```
from rolepermissions.permissions import revoke_permission

revoke_permission(user, 'create_medical_record')
```

5.2 Permission and role verification

The following functions will always return `True` for users with supperuser status.

has_role (*user*, *roles*)

Receives a user and a role and returns `True` if user has the specified role. Roles can be passed as object, snake cased string representation or inside a list.

```
from rolepermissions.checkers import has_role
from my_project.roles import Doctor

if has_role(user, [Doctor, 'nurse']):
    print 'User is a Doctor or a nurse'
```

has_permission (*user*, *permission*)

Receives a user and a permission and returns `True` is the user has ths specified permission.

```
from rolepermissions.checkers import has_permission
from my_project.roles import Doctor
from records.models import MedicalRecord

if has_permission(user, 'create_medical_record'):
    medical_record = MedicalRecord(...)
    medical_record.save()
```

has_object_permission (*checker_name*, *user*, *obj*)

Receives a string referencing the object permission checker, a user and the object to be verified.

```
from rolepermissions.checkers import has_object_permission
from clinics.models import Clinic

clinic = Clinic.objects.get(id=1)

if has_object_permission('access_clinic', user, clinic):
    print 'access granted'
```

5.3 Template tags

To load template tags use:

```
{% load permission_tags %}
```

***filter* has_role**

Receives a camel case representation of a role or more than one separated by coma.

```
{% load permission_tags %}
{% if user|has_role:'doctor,nurse' %}
    the user is a doctor or a nurse
{% endif %}
```

***filter* can**

Role permission filter.

```
{% load permission_tags %}
{% if user|can:'create_medical_record' %}
    <a href="/create_record">create record</a>
{% endif %}
```

***tag* can**

If no user is passed to the tag, the logged user will be used in the verification.

```
{% load permission_tags %}

{% can "access_clinic" clinic user=user as can_access_clinic %}
{% if can_access_clinic %}
    <a href="/clinic/1/">Clinic</a>
{% endif %}
```

CHAPTER 6

Mixins and Decorators

6.1 Decorators

Decorators require that the current logged user attend some permission grant. They are meant to be used on function based views.

has_role_decorator(*role*)

Accepts the same arguments as `has_role` function and raises PermissionDenied in case it returns `False`. You can pass an optional key word argument `redirect_to_login` to overhide the `ROLEPERMISSIONS_REDIRECT_TO_LOGIN` setting.

```
from rolepermissions.decorators import has_role_decorator

@has_role_decorator('doctor')
def my_view(request, *args, **kwargs):
        ...
```

has_permission_decorator(*permission_name*)

Accepts the same arguments as `has_permission` function and raises PermissionDenied in case it returns `False`. You can pass an optional key word argument `redirect_to_login` to overhide the `ROLEPERMISSIONS_REDIRECT_TO_LOGIN` setting.

```
from rolepermissions.decorators import has_permission_decorator

@has_permission_decorator('create_medical_record')
def my_view(request, *args, **kwargs):
        ...
```

6.2 Mixins

Mixins require that the current logged user attend some permission grant. They are meant to be used on class based views.

class HasRoleMixin(object)

Add `HasRoleMixin` mixin to the desired CBV (class based view) and use the `allowed_roles` attribute to set the roles that can access the view. `allowed_roles` attribute will be passed to `has_role` function, and PermissionDenied will be raised in case it returns `False`. You can set an optional `redirect_to_login` attribute to overhide the `ROLEPERMISSIONS_REDIRECT_TO_LOGIN` setting.

```
from django.views.generic import TemplateView
from rolepermissions.mixins import HasRoleMixin

class MyView(HasRoleMixin, TemplateView):
        allowed_roles = 'doctor'
        ...
```

class HasPermissionsMixin(object)

Add `HasPermissionsMixin` mixin to the desired CBV (class based view) and use the `required_permission` attribute to set the roles that can access the view. `required_permission` attribute will be passed to `has_permission` function, and PermissionDenied will be raised in case it returns `False`. You can set an optional `redirect_to_login` attribute to overhide the `ROLEPERMISSIONS_REDIRECT_TO_LOGIN` setting.

```
from django.views.generic import TemplateView
from rolepermissions.mixins import HasPermissionsMixin

class MyView(HasPermissionsMixin, TemplateView):
        required_permission = 'create_medical_record'
        ...
```

CHAPTER 7

Admin Integration

Use Django User Admin Site to manage roles and permissions interactively.

7.1 Permission Names

Permissions defined in `roles.py` are given 'human-friendly' names.

All such permissions are assigned to the `auth | user` Content Type.

Permission names are a Title Case version of the snake_case or camelCase permission codename, so...

- `create_medical_record` is named `auth | user | Create Medical Record`
- `enterSurgery` is named `auth | user | Enter Surgery`

7.2 RolePermissions User Admin

Assign / remove roles when editing Users in the Django User Admin Site.

RolePermissionsUserAdmin()
Custom `django.contrib.auth.admin.UserAdmin` that essentially adds the following logic. To be used with standard django User model:

- `remove_role(user, group)` is called for each Group, removed via the Admin, that represents a role.
- `assign_role(user, group)` is called for each Group, added via the Admin, that represents a role.

Opt-in with `setting`: *ROLEPERMISSIONS_REGISTER_ADMIN* = True

RolePermissionsUserAdminMixin()
Mixin the functionality of `RolePermissionsUserAdmin` to your own custom `UserAdmin` class. To be used with custom User model:

```
class MyCustomUserAdmin(RolePermissionsUserAdminMixin, django.contrib.auth.admin.
↪UserAdmin):
    ...
```

Warning: `remove_role` removes every permission associated with a removed `Group`, regardless of how those permissions were originally assigned. See *remove_role()*

7.3 Management Commands

```
django-admin sync_roles
```

Ensures that `django.contrib.auth.models` `Group` and `Permission` objects exist for each role defined in `roles.py`

This makes the roles and permissions defined in code immediately acccessible via the Django User Admin

Note: `sync_roles` never deletes a `Group` or `Permission`.

If you remove a role or permission from `roles.py`, the corresponding `Group` / `Persission` continues to exist until it is manually removed.

```
django-admin sync_roles --reset_user_permissions
```

Additionally, update every User's permissions to ensure they include all those defined by their current roles.

Warning: `--reset_user_permissions` is primarily intended for development, not production!

Changing which permissions are associated with a role in `roles.py` does NOT change any User's actual permissions! `--reset_user_permissions` simply clears each User's roles and then re-assign them. This guarantees that Users will have all permissions defined by their role(s) in `roles.py`, but in no way does this imply that any permissions previously granted to the User have been revoked!

CHAPTER 8

Settings

8.1 Redirect to the login page

Instead of getting a Forbidden (403) error when the user has no permission, you can make the request be redirected to the login page. Add the following variable to your django `settings.py`:

`settings.py`

```
ROLEPERMISSIONS_REDIRECT_TO_LOGIN = True
```

8.2 Register User Admin

Replaces the default `django.contrib.auth.admin.UserAdmin` with *RolePermissionsUserAdmin* so you can manange roles interactively via the Django User Admin Site.

Add the following variable to your django `settings.py`:

`settings.py`

```
ROLEPERMISSIONS_REGISTER_ADMIN = True
```

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index