## Contents

1 Setup ................................................. 3
   1.1 Installation ..................................... 3
   1.2 Configuration .................................... 3

2 Quick Start ........................................... 5

3 Roles ..................................................... 7
   3.1 Roles File ......................................... 7
   3.2 Available Role Permissions ....................... 8

4 Object permission checkers ......................... 9
   4.1 permissions.py file .............................. 9
   4.2 Checking object permission ...................... 9

5 Utils .................................................... 11
   5.1 Shortcuts .......................................... 11
   5.2 Permission and role verification .................. 13
   5.3 Template tags ..................................... 13

6 Mixins and Decorators .............................. 15
   6.1 Decorators ......................................... 15
   6.2 Mixins .............................................. 16

7 Admin Integration .................................... 17
   7.1 Permission Names .................................. 17
   7.2 RolePermissions User Admin ...................... 17
   7.3 Management Commands ............................. 18

8 Settings ............................................... 19
   8.1 Redirect to the login page ....................... 19
   8.2 Register User Admin ............................... 19
   8.3 Disable superuser superpowers ................. 19

9 Indices and tables .................................... 21

Index .................................................... 23
CHAPTER 1

Setup

1.1 Installation

Install from PyPI with pip:

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

1.2 Configuration

Add `rolepermissions` to your `INSTALLED_APPS`

```
INSTALLED_APPS = (  
    ...  
    'rolepermissions',  
    ...  
)
```
Create a roles.py file in the same folder as your settings.py and two roles:

```python
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:

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

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

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

and check its permissions using

```python
>>> 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
Roles

3.1 Roles File

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

```python
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

```python
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:

```python
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.
### 4.1 permissions.py file

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

```python
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.
5.1 Shortcuts

**get_user_roles** *(user)*

Returns the user’s roles.

```python
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.

```python
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.

```python
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:

```python
>>> class Doctor(AbstractUserRole):
...     ...     available_permissions = {
...     ...         "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.

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

```
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.

```python
grant_permission(user, 'create_medical_record')
```

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

Revolves a permission from a user. Will raise a RolePermissionScopeException for a permission that is not listed in the user’s roles’ available_permissions.

```python
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.

```python
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 if the user has the specified permission.

```python
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.

```python
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:

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

**filter* has_role**

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

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

**filter* can**

Role permission filter.

5.2. Permission and role verification 13
{% 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 %}
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 override the ROLEPERMISSIONS_REDIRECT_TO_LOGIN setting.

```python
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 override the ROLEPERMISSIONS_REDIRECT_TO_LOGIN setting.

```python
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 override the ROLEPERMISSIONS_REDIRECT_TO_LOGIN setting.

```python
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 override the ROLEPERMISSIONS_REDIRECT_TO_LOGIN setting.

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

class MyView(HasPermissionsMixin, TemplateView):
    required_permission = 'create_medical_record'
    ...
```
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:
Warning: remove_role removes every permission associated with a removed Group, regardless of how those permissions were originally assigned. See remove_role()
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:

```
ROLEPERMISSIONS_REDIRECT_TO_LOGIN = True
```

8.2 Register User Admin

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

Add the following variable to your django settings.py:

```
ROLEPERMISSIONS_REGISTER_ADMIN = True
```

8.3 Disable superuser superpowers

By default Django superusers have all roles and permissions. You can disable this behavior and make them respect their roles and permissions.

Superusers still can add any role or permission to them through Django Admin.
ROLEPERMISSIONS_SUPERUSER_SUPERPOWERS = False
CHAPTER 9

Indices and tables

- genindex
- modindex
- search
<table>
<thead>
<tr>
<th>Index</th>
</tr>
</thead>
</table>

| A | assign_role() *(built-in function)*, 11  
|   | available_perm_status() *(built-in function)*, 12 |
| C | clear_roles() *(built-in function)*, 12 |
| G | get_user_roles() *(built-in function)*, 11  
|   | grant_permission() *(built-in function)*, 12 |
| H | has_object_permission() *(built-in function)*, 13  
|   | has_permission() *(built-in function)*, 13  
|   | has_permission_decorator() *(built-in function)*, 15 |
|   | has_role() *(built-in function)*, 13  
|   | has_role_decorator() *(built-in function)*, 15 |
| R | remove_role() *(built-in function)*, 11  
|   | revoke_permission() *(built-in function)*, 12  
|   | RolePermissionsUserAdmin() *(built-in function)*, 17 |
|   | RolePermissionsUserAdminMixin() *(built-in function)*, 17 |