Django Improved Permissions Documentation

Gabriel de Biasi

May 04, 2018

Contents:

Django Improved Permissions (DIP) is a django application made to make django's default permission system more robust. Here are some highlights:

- Object-level Permissions
- Role Assignment
- Permissions Inheritance
- Cache
- Customizable Permissions per User Instance

CHAPTER 1

Setup

1.1 Installation

We are in PyPI. Just use the following command within your development environment:

```
pip install django-improved-permissions
```

1.2 Configuration

We use some apps that are already present in Django: `auth` and `contenttypes`. Probably they are already declared, but just make sure so we don't have any issues later.

```
# settings.py

INSTALLED_APPS = (
...
'django.contrib.auth',
'django.contrib.contenttypes',
...
)
```

Now, you need to add our app inside your Django project. To do this, add `improved_permissions` into your `INSTALLED_APPS`:

```
# settings.py

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

Note: We are almost there! We use some tables in the database to store the permissions, so you must run `./manage.py migrate improved_permissions` in order to migrate all models needed.

Yeah, all set to start! Let's go to the next page to get a quick view of how everything works.

CHAPTER 2

Quick Start

The entire DIP permissions system works based on roles. In other words, if you want to have permissions between a user and a certain object, you need to define a role for this relationship.

2.1 Creating your first Role class

First, supose that you have the following model in your `models.py`:

```
# myapp/models.py

from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=256)
    content = models.TextField(max_length=1000)

    class Meta:
        permissions = (
            ('read_book', 'Can Read Book'),
            ('review_book', 'Can Review Book')
        )
```

Note: Notice that the permission statements inside models is exactly like the Django `auth` system.

Now, create a new file inside of any app of your project named `roles.py` and implement as follows:

```
# myapp/roles.py

from improved_permissions.roles import Role
from myapp.models import Book

```

(continues on next page)

(continued from previous page)

```
class Author(Role):
    verbose_name = "Author"
    models = [Book]
    deny = ['myapp.review_book']

class Reviewer(Role):
    verbose_name = "Reviewer"
    models = [Book]
    allow = ['myapp.review_book']
```

Ready! You can now use the DIP functions to assign, remove, and check permissions.

Every time your project starts, we use an `autodiscover` in order to validate and register your Role classes automatically. So, don't worry about to do anything else.

2.2 Using the first shortcuts

Once you implement the role classes, you are ready to use our shortcuts. For example, let's create a `Book` object and an `Author` role for a user:

```
from django.contrib.auth.models import User
from improved_permissions.shortcuts import assign_role, has_permission, has_role

from myapp.models import Book
from myapp.roles import Author, Reviewer


john = User.objects.get(pk=1)
book = Book.objects.create(title='Nice Book', content='Such content.')

has_role(john, Author, book)
>>> False
has_permission(john, 'myapp.read_book', book)
>>> False

assign_role(john, Author, book)

has_role(john, Author, book)
>>> True
has_permission(john, 'myapp.read_book', book)
>>> True
has_permission(john, 'myapp.review_book', book)
>>> False
```

You just met the shortcuts `assign_role`, `has_role` and `has_permission`. If you don't get how they work, no problem. First, let's understand all about implementing `Role` classes in the next section.

CHAPTER 3

Role Class

Role classes must be implemented by inheriting the `Role` class present in `improved_permissions.roles`.

Whenever you start your project, they are automatically validated and can already be used. If something is wrong, `RoleManager` will raise an exception with a message explaining the cause of the error.

Note: Only Role classes inside modules called `roles.py` are automatically validated. See here to change this behavior.

3.1 Required attributes

The `Role` class has some attributes that are required to be properly registered by our `RoleManager`. The description of these attributes is in the following table:

Attribute	Type	Description
`verbose_name`	`str`	Used to print some information about the role.
`models`	`list` or `ALL_MODELS`	Defines which models this role can be attached.
`allow` or `deny`	`list`	Defines which permissions should be allowed or denied. You must define only one of them.

3.2 Optional Attributes

The `Role` class also has other attributes, which are considered as optional. When they are not declared, we assign default values for these arguments.

Attribute	Type	Default	Description
`unique`	`bool`	`False`	Only one User instance is allowed to be attached to a given object using this role.
`ranking`	`int`	`0`	Used in order to solve permissions conflit. More about this in the examples.
`inherit`	`bool`	`False`	Allows this role to inherit permissions from its child models. Read about this feature here.
`inherit_allow` or `inherit_deny`	`list`	`[]`	Specifies which inherit permissions should be allowed or denied. You must define only one of them.

3.2.1 Unique Roles

You will probably arrive in a case where an object can only have one User instance assigned to a particular role. But, it is as easy as learning python to do this using DIP. For example:

```
class CarOwner(Role):
    verbose_name = 'Owner of a Car'
    models = [Car]
    deny = []
```

Now, let's test this on the terminal:

```
my_car = Car.objects.create(name='My New Car')

# Giving a new car to john!
john = User.objects.get(pk=1)
assign_role(john, CarOwner, my_car)

# That's right.
has_role(john, CarOwner, my_car)
>>> True

# Oh, no...
bob = User.objects.get(pk=2)
assign_role(bob, CarOwner, my_car)

# And now?
has_role(bob, CarOwner, my_car)
>>> True # Noooooo :(
```

To fix this, let's change the implementation of the role class and add the `unique` attribute.

```
class CarOwner(Role):
    verbose_name = 'Owner of a Car'
    models = [Car]
    deny = []
    unique = True
```

Let's test this again:

```
my_car = Car.objects.create(name='My New Car')

# Giving a new car to john!
```

(continues on next page)

(continued from previous page)

```
john = User.objects.get(pk=1)
assign_role(john, CarOwner, my_car)

# That's right.
has_role(john, CarOwner, my_car)
>>> True

# And now we are protected :D
bob = User.objects.get(pk=2)
assign_role(bob, CarOwner, my_car)
>>> InvalidRoleAssignment: 'The object "Car" already has a "CarOwner" attached and it␣
↪is marked as unique.'
```

3.2.2 Role Ranking

There are several cases that can lead your project to have permissions conflicts. We have a basic scenario to show you how this happens and how you can use role ranking to solve it. For example:

```
class Teacher(Role):
    verbose_name = 'Teacher'
    models = [User]
    deny = ['user.update_user']

class Advisor(Role):
    verbose_name = 'Advisor'
    models = [User]
    deny = []
```

Note that these roles have conflicting permissions if both are assigned to the same User instance. To solve this conflict problem, you can assign an integer value to `ranking`, present in the Role class. This value will be used to sort the permissions to be used by the DIP.

In other words, the lower the `ranking` value, more important this role is. So, let's work using ranking now:

```
class Teacher(Role):
    verbose_name = 'Teacher'
    models = [User]
    deny = ['user.update_user']
    ranking = 1

class Advisor(Role):
    verbose_name = 'Teacher'
    models = [User]
    deny = []
    ranking = 0
```

Now let's test this on the terminal:

```
john = User.objects.get(pk=1)
bob = User.objects.get(pk=2)

assign_role(john, Advisor, bob)
assign_role(john, Teacher, bob)

# Now has_permission returns True using
```

(continues on next page)

(continued from previous page)

```
# the role "Advisor" by Role Ranking.
has_permission(john, 'user.update_user', bob)
>>> True
```

3.3 Role Classes using ALL_MODELS

If you need a role that manages any model of your project, you can define the `models` attribute using `ALL_MODELS`. These classes are `inherit=True` by default because they don't have their own permissions, only inherited permissions. For example:

```
# myapp/roles.py

from improved_permissions.roles import ALL_MODELS, Role

class SuperUser(Role):
    verbose_name = 'Super Man Role'
    models = ALL_MODELS
    deny = []
    inherit_deny = []
```

Because this class is not attached to a specific model, you can use the shortcuts without defining objects. For example:

```
from myapp.models import Book
from myapp.roles import SuperUser

john = User.objects.get(pk=1)
book = Book.objects.create(title='Nice Book', content='Such content.')

# You shouldn't pass an object during assignment.
assign_role(john, SuperUser)

# This line will raise an InvalidRoleAssignment exception
assign_role(john, SuperUser, book)

# You can check with and without an object.
has_permission(john, 'myapp.read_book')
>>> True
has_permission(john, 'myapp.read_book', book)
>>> True
```

3.4 Public Methods

The role classes have some class methods that you can call if you need them.

`get_verbose_name(): str`

Returns the `verbose_name` attribute. Example:

```
from myapp.roles import Author, Reviewer

Author.get_verbose_name()
>>> 'Author'
```

(continues on next page)

(continued from previous page)

```
Reviewer.get_verbose_name()
>>> 'Reviewer'
```

is_my_model(model): bool

Checks if the role can be attached to the argument `model`. The argument can be either the model class or an instance. Example:

```
from myapp.models import Book
from myapp.roles import Author

Author.is_my_model('some data')
>>> False
Author.is_my_model(Book)
>>> True
my_book = Book.objects.create(title='Nice Book', content='Nice content.')
Author.is_my_model(my_book)
>>> True
```

get_models(): list

Returns a list of all model classes which this role can be attached. If the `models` attribute was defined using `ALL_MODELS`, this method will return a list of all valid models of the project. For example:

```
from myapp.models import Book
from myapp.roles import Author, SuperUser

Author.get_models()
>>> [Book]
SuperUser.get_models()
>>> [Book, User, Permission, ContentType, ...] # all models known by Django
```

In the next section, we describe all existing shortcuts in this app.

CHAPTER 4

Shortcuts

These functions are the heart of this app. Everything you need to do in your project is implemented in the `shortcuts` module.

Note: Do not rush your project using the shortcuts directly. We have an easiest way to use these shorcuts using **mixins** in your models. Click here to check it out.

4.1 Checkers

has_role(*user*, *role_class*, *obj=None*)

Returns True if the user has the role to the object.

has_permission(*user*, *permission*, *obj=None*)

Returns True if the user has the permission.

4.2 Assigning and Revoking

assign_role(*user*, *role_class*, *obj=None*)

Assign the role to the user.

assign_roles(*users_list*, *role_class*, *obj=None*)

Assign the role to all users in the list.

remove_role(*user*, *role_class*, *obj=None*)

Remove the role and your permissions of the object from the user.

remove_roles(*users_list=None*, *role_class*, *obj=None*)

Remove the role and your permissions of the object from all users in the list.

4.3 Getters

get_role(*user*, *obj=None*)

Get the unique role class of the user related to the object.

get_roles(*user*, *obj=None*)

Get all role classes of the user related to the object.

get_user(*role_class=None*, *obj=None*)

Get the unique user instance according to the object.

get_user(*role_class=None*, *obj=None*)

Get the unique user instance according to the object.

get_users(*role_class=None*, *obj=None*)

Get all users instances according to the object.

get_objects(*user*, *role_class=None*, *model=None*)

Get all objects related to the user.

CHAPTER 5

Mixins

We've implemented three mixins to make it easier to use the shortcuts in your project. All mixins are located in `improved_permisions.mixins`.

5.1 RoleMixin

Mixin for objects.

5.2 UserRoleMixin

Mixin for users.

5.3 PermissionMixin

Mixin for views.

CHAPTER 6

Permissions Inheritance

The DIP allows you to implement inheritance permissions for your objects. For example, a librarian does't need to have explicit permissions to all their books in his library as long as the books make it clear that the library is an object in which it "belongs" to.

6.1 Class RoleOptions

The class `RoleOptions` works just like the `Meta` class in the Django models, helping us to define some attributes related to that specific model. This class has the following attributes:

Attribute	Type	Description
`permission_parents`	list of str	List of `ForeignKey` or `GenericForeignKey` fields on the model to be considered as *parent* of the model.
`unique_together`	bool	If `True`, this model only allows one assignment to any User instance.

6.2 Working with the inheritance

Let's go back to that first example, the model `Book`. We are going to implement another model named `Library` and create a `ForeignKey` field in `Book` to create a relationship between them. So, our `models.py` will be something like that:

```
# myapp/models.py

from django.db import models

class Library(models.Model):
    name = models.CharField(max_length=256)
```

(continues on next page)

(continued from previous page)

```
class Book(models.Model):
    title = models.CharField(max_length=256)
    content = models.TextField(max_length=1000)
    my_library = models.ForeignKey(Library)

    class Meta:
        permissions = (
            ('read_book', 'Can Read Book'),
            ('review_book', 'Can Review Book')
        )
```

We need to say to DIP that the `my_library` represents a parent of the `Book` model. In other words, any roles related to the `Library` model with `inherit=True` will be elected to search for more permissions.

The way to do this is implementing another inner class in the model, the class `RoleOptions` and defining the list `permission_parents`:

```
# myapp/models.py

from django.db import models
from improved_permissions.mixins import RoleMixin


class Library(models.Model, RoleMixin):
    name = models.CharField(max_length=256)


class Book(models.Model, RoleMixin):
    title = models.CharField(max_length=256)
    content = models.TextField(max_length=1000)
    my_library = models.ForeignKey(Library)

    class Meta:
        permissions = (
            ('read_book', 'Can Read Book'),
            ('review_book', 'Can Review Book')
        )

    class RoleOptions:
        permission_parents = ['my_library']
```

Let's create a new role in order to represent the `Library` instances.

```
# myapp/roles.py

from improved_permissions.roles import Role
from myapp.models import Library

class LibraryManager(Role):
    verbose_name = 'Library Manager'
    models = [Library]
    allow = []
    inherit = True
    inherit_allow = ['myapp.read_book']
```

After that, the field `my_library` already represents a parent model of the `Book`. Now, let's go to the terminal to make some tests:

```
# Django Shell

from django.contrib.auth.models import User
from improved_permissions.shortcuts import assign_role, has_permission
from myapp.models import Book, Library
from myapp.roles import LibraryManager

john = User.objects.get(pk=1)

library = Library.objects.create(name='Important Library')
book = Book.objects.create(title='New Book', content='Much content', my_
↪library=library)

# John has nothing :(
has_permission(john, 'myapp.read_book', book)
>>> False

# John receives an role attached to "library".
assign_role(john, LibraryManager, library)

# Now, we got True by permission inheritance.
has_permission(john, 'myapp.read_book', book)
>>> True
```

6.3 Unique roles to a given object

There is a scenario where a model has several roles related to it, but a single user must be assigned to only one of them. In order to allow this behavior, we have the boolean attribute called `unique_together`.

Let's say that one user must not be the `Author` and the `Reviewer` of a given `Book` instance at same time. Let's see on the terminal:

```
# Django Shell

from django.contrib.auth.models import User
from improved_permissions.shortcuts import assign_role, has_permission
from myapp.models import Book
from myapp.roles import Author, Reviewer

john = User.objects.get(pk=1)
book = Book.objects.create(title='New Book', content='Much content', my_
↪library=library)

# John is the Author.
assign_role(john, Author, book)

# And also the Reviewer.
assign_role(john, Reviewer, book)

# We cannot allow that :(
has_permission(john, 'myapp.read_book', book)
>>> True
has_permission(john, 'myapp.review_book', book)
>>> True
```

Now, let's change the class `RoleOptions` inside of `Book`:

```
# myapp/models.py

from django.db import models
from improved_permissions.mixins import RoleMixin

class Book(models.Model, RoleMixin):
    title = models.CharField(max_length=256)
    content = models.TextField(max_length=1000)
    my_library = models.ForeignKey(Library)

    class Meta:
        permissions = (
            ('read_book', 'Can Read Book'),
            ('review_book', 'Can Review Book')
        )

    class RoleOptions:
        permission_parents = ['my_library']

        # new feature here!
        # --------------------
        unique_together = True
        # --------------------
```

Going back to the terminal to see the result:

```
# Django Shell

from django.contrib.auth.models import User
from improved_permissions.shortcuts import assign_role, has_permission
from myapp.models import Book
from myapp.roles import Author, Reviewer

john = User.objects.get(pk=1)
book = Book.objects.create(title='New Book', content='Much content', my_
↪library=library)

# John is the Author.
assign_role(john, Author, book)

# Can be the Reviewer now?
assign_role(john, Reviewer, book)
>>> InvalidRoleAssignment: 'The user "john" already has a role attached to the object
↪"book".'
```

Yeah! Now we are safe.

CHAPTER 7

API Reference

TODO

CHAPTER 8

Help

So do I.

8.1 Need further help?

oi.

8.2 Contributing

Feel free to create new issues if you have suggestions or find some bugs.

8.3 Commercial Support

This project is used in products of SSYS clients.

We are always looking for exciting work, so if you need any commercial support, feel free to get in touch: contato@ssys.com.br

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index