Last updated April 11th, 2022

Permissions in Django

Oluwole Majiyagbe

Django comes with a powerful permission system out-of-the-box.

In this article, we'll look at how to assign permissions to users and groups in order to authorize them to perform specific actions.

Objectives

By the end of this article, you will be able to:

1. Explain how Django's permissions and groups work
2. Harness the power of Django's built-in permission system

Authentication vs Authorization

This article is concerned with authorization.

- **Authentication** is the process of confirming if a user has access to a system. Typically, a username/email and password is used to authenticate a user.
- **Authorization:** pertains to what the "authenticated" user can do in a system.

Put another way, authentication answers the question 'who are you?' while authorization answers 'what can you do?'.

User-level Permissions

When `django.contrib.auth` is added to the `INSTALLED_APPS` setting in the *settings.py* file, Django automatically creates `add`, `change`, `delete` and `view` permissions for each Django model that's created.

Permissions in Django follow the following naming sequence:

`{app}.{action}_{model_name}`

Notes:

- `app` is the name of the Django app the associated model resides in
- `action`: is `add`, `change`, `delete`, or `view`
- `model_name`: is name of the model in lowercase

Let's assume we have the following model in an app called "blog":

```
from django.db import models


class Post(models.Model):
    title = models.CharField(max_length=400)
    body = models.TextField()
```

By default, Django will create the following permissions:

1. `blog.add_post`
2. `blog.change_post`
3. `blog.delete_post`
4. `blog.view_post`

You can then check if a user (via a Django user object) has permissions with the `has_perm()` method:

```
from django.contrib.auth import get_user_model
from django.contrib.auth.models import User, Permission
from django.contrib.contenttypes.models import ContentType

from blog.models import Post

content_type = ContentType.objects.get_for_model(Post)
post_permission = Permission.objects.filter(content_type=content_type)
print([perm.codename for perm in post_permission])
# => ['add_post', 'change_post', 'delete_post', 'view_post']

user = User.objects.create_user(username="test", password="test", email="test@user.com")

# Check if the user has permissions already
print(user.has_perm("blog.view_post"))
# => False

# To add permissions
for perm in post_permission:
    user.user_permissions.add(perm)

print(user.has_perm("blog.view_post"))
# => False
# Why? This is because Django's permissions do not take
# effect until you allocate a new instance of the user.

user = get_user_model().objects.get(email="test@user.com")
print(user.has_perm("blog.view_post"))
# => True
```

Superusers will always have permission set to `True` even if the permission does not exist:

```
from django.contrib.auth.models import User

superuser = User.objects.create_superuser(
    username="super", password="test", email="super@test.com"
)

# Output will be true
print(superuser.has_perm("blog.view_post"))

# Output will be true even if the permission does not exists
print(superuser.has_perm("foo.add_bar"))
```

> A superuser is a user type in Django that has every permission in the system. Whether custom permissions or Django-created permissions, superusers have access to every permission.
>
> A staff user is just like any other user in your system BUT with the added advantage of being able to access the Django Admin interface. The Django Admin interface is only accessible to superusers and staff users.

Group-level Permissions

Having to assign permissions each time to users is tedious and not scalable. There might be instances where you would want to add new permissions to a set of users. Here's where Django groups come into play.

What's a group?

- English Definition: A group is a set of objects which are classed together.
- Django Definition: Group models are a generic way of categorizing users so you can apply permissions, or some other label, to those users. A user can belong to any number of groups.

With Django, you can create groups to class users and assign permissions to each group so when creating users, you can just assign the user to a group and, in turn, the user has all the permissions from that group.

To create a group, you need the `Group` model from `django.contrib.auth.models`.

Let's create groups for the following roles:

- `Author`: Can view and add posts
- `Editor`: Can view, add, and edit posts
- `Publisher`: Can view, add, edit, and delete posts

Code:

```
from django.contrib.auth.models import Group, User, Permission
from django.contrib.contenttypes.models import ContentType
from django.shortcuts import get_object_or_404

from blog.models import Post

author_group, created = Group.objects.get_or_create(name="Author")
editor_group, created = Group.objects.get_or_create(name="Editor")
publisher_group, created = Group.objects.get_or_create(name="Publisher")

content_type = ContentType.objects.get_for_model(Post)
post_permission = Permission.objects.filter(content_type=content_type)
print([perm.codename for perm in post_permission])
# => ['add_post', 'change_post', 'delete_post', 'view_post']

for perm in post_permission:
    if perm.codename == "delete_post":
        publisher_group.permissions.add(perm)

    elif perm.codename == "change_post":
        editor_group.permissions.add(perm)
        publisher_group.permissions.add(perm)
    else:
        author_group.permissions.add(perm)
        editor_group.permissions.add(perm)
        publisher_group.permissions.add(perm)

user = User.objects.get(username="test")
user.groups.add(author_group)  # Add the user to the Author group

user = get_object_or_404(User, pk=user.id)

print(user.has_perm("blog.delete_post")) # => False
print(user.has_perm("blog.change_post")) # => False
print(user.has_perm("blog.view_post")) # => True
print(user.has_perm("blog.add_post")) # => True
```

Enforcing Permissions

Aside for the Django Admin, permissions are typically enforced at the view layer since the user is obtained from the request object.

To enforce permissions in class-based views, you can use the PermissionRequiredMixin from `django.contrib.auth.mixins` like so:

```python
from django.contrib.auth.mixins import PermissionRequiredMixin
from django.views.generic import ListView

from blog.models import Post

class PostListView(PermissionRequiredMixin, ListView):
    permission_required = "blog.view_post"
    template_name = "post.html"
    model = Post
```

`permission_required` can either be a single permission or an iterable of permissions. If using an iterable, a user must have ALL the permissions before they can access the view:

```python
from django.contrib.auth.mixins import PermissionRequiredMixin
from django.views.generic import ListView

from blog.models import Post

class PostListView(PermissionRequiredMixin, ListView):
    permission_required = ("blog.view_post", "blog.add_post")
    template_name = "post.html"
    model = Post
```

For function-based views, use the `permission_required` decorator:

```python
from django.contrib.auth.decorators import permission_required

@permission_required("blog.view_post")
def post_list_view(request):
    return HttpResponse()
```

You can also check for permissions in your Django templates. With Django's auth context processors, a perms variable is available by default when you render your template. The `perms` variable actually contains all permissions in your Django application.

For example:

```
{% if perms.blog.view_post %}
  {# Your content here #}
{% endif %}
```

Model-level Permissions

You can also add custom permissions to a Django model via the model Meta options.

Let's add an `is_published` flag to the `Post` model:

```python
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=400)
    body = models.TextField()
    is_published = models.Boolean(default=False)
```

Next, we'll set a custom permission called `set_published_status`:

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=400)
    body = models.TextField()
    is_published = models.Boolean(default=False)

    class Meta:
        permissions = [
            (
                "set_published_status",
                "Can set the status of the post to either publish or not"
            )
        ]
```

In order to enforce this permission, we can use the `UserPassesTestMixin` Django provided mixin in our view, giving us the flexibility to explicitly check whether a user has the required permission or not.

Here's what a class-based view might look like that checks whether a user has permission to set the published status of a post:

```
from django.contrib.auth.mixins import UserPassesTestMixin
from django.shortcuts import render
from django.views.generic import View

from blog.models import Post

class PostListView(UserPassesTestMixin, View):
    template_name = "post_details.html"

    def test_func(self):
        return self.request.user.has_perm("blog.set_published_status")

    def post(self, request, *args, **kwargs):
        post_id = request.POST.get('post_id')
        published_status = request.POST.get('published_status')

        if post_id:
            post = Post.objects.get(pk=post_id)
            post.is_published = bool(published_status)
            post.save()

        return render(request, self.template_name)
```

So, with `UserPassesTestMixin`, you need to override the `test_func` method of the class and add your own test. Do note that the return value of this method must always be a boolean.

Object-level Permissions

If you're using Django REST Framework, it already has object-level permissions built into the base permission class. `BasePermission` has `has_permission`, which is basically for list views, and `has_object_permission`, which checks if the user has permission to access a single model instance.

> For more on permissions in Django REST Framework, review Permissions in Django REST Framework.

If you're not using Django REST Framework, to implement object-level permissions, you can use a third-party, like:

- Django Guardian
- Rules

> For more permissions-related packages, check out Django Packages.

Conclusion

In this article, you've learnt how to add permissions to a Django model and check for permissions. If you have a set number of user types, you can create each user type as a group and give the necessary permissions to the group. Then, for every user that is added into the system and into the required group, the permissions are automatically granted to each user.

django

Oluwole Majiyagbe

Oluwole is a software engineer with a passion for new tech and a thirst for learning. When he's not coding, he can be found playing video games or reading mostly fiction novels.

CONTRIBUTORS

Michael Herman

SHARE THIS TUTORIAL

Twitter Reddit Hacker News Facebook

TUTORIAL TOPICS

api **architecture** **aws** **devops** **django** **django rest framework** **docker** **fastapi** **flask** **front-end** **heroku** **kubernetes** **machine learning** **python** **react** **task queue** **testing** **vue** **web scraping**