**Chapter 14: Permissions**

Currently there are no permissions set on our Bookstore project. Any user, even one not logged in, can visit any page and perform any available action. While this is fine for prototyping, implementing a robust permissions structure is a must before deploying a website to production.

Django comes with built-in authorization options for locking down pages to either logged in users, specific groups, or users with the proper individual permission.

**Logged-In Users Only**

Confusingly there are multiple ways to add even the most basic permission: restricting access only to logged-in users. It can be done in a raw way using the login\_required() decorator, or since we are using class-based views so far via the LoginRequired mixin.

Let’s start by limiting access to the Books pages only to logged-in users. There is a link for it in the navbar so this is not the case of a user accidentally finding a URL (which also can happen); in this case the URL is quite public.

First import LoginRequiredMixin at the top which we will then add to our BookListView class. It is important that LoginRequiredMixin come before ListView in order to work properly. Mixins are powerful but can be a little tricky in practice. As the official docs note, “not all mixins can be used together, and not all generic class based views can be used with all other mixins..

The second update is adding a login\_url for the user to be redirected to. This is the URL name for log in which, since we’re using django-allauth is account\_login. If we were using the traditional Django authentication system then this link would be called simply login.

The structure for BookDetailView is the same: add LoginRequiredMixin before DetailView and a login\_url route as well.

# books/views.py

from django.contrib.auth.mixins import LoginRequiredMixin # new

from django.views.generic import ListView, DetailView

from .models import Book

# Create your views here.

class BookListView(LoginRequiredMixin, ListView): # new (LoginRequiredMixin)

model = Book

context\_object\_name = "book\_list"

template\_name = "books/book\_list.html"

login\_url = "account\_login" # new

class BookDetailView(LoginRequiredMixin, DetailView): # new

model = Book

context\_object\_name = "book"

template\_name = "books/book\_detail.html"

login\_url = "account\_login" # new

And that’s it! If you now log out of your account and click on the “Books” link it will automatically redirect you to the Log In page. However if you are logged in, the Books page loads normally. Even if you somehow knew the UUID of a specific book page you’ll still be redirected to the Log In page!

**Permissions**

Django comes with a basic permissions system that is controlled through the Django admin. To demonstrate it we need to create a new user account. Navigate back to the Admin homepage and then click on “+ Add” next to Users.

We’ll call this new user special and set a password of testpass123. Click on the “Save” button. The second page allows us to set an “Email address” to special@email.com. Scrolling down further on the page to the bottom there are options to set Groups as well as User permissions. This is a long list of defaults Django provides. For now we won’t use them since we’ll create a custom permission in the next section so just click on the “Save” button in the lower right corner so that our email address is updated for the user account.

**Custom Permissions**

Setting custom permissions is a much more common occurrence in a Django project. We can set them via the Meta class on our database models.

For example, let’s add a special status so that an author can read all books. In other words they have access to the DetailView. We could be much more specific with the permissions, restricting them per book, but this is a good first step.

In the books/models.py file we’ll add a Meta class and set both the permission name and a description which will be visible in the admin.

# books/models.py

…

class Book(models.Model):

id = models.UUIDField(

primary\_key=True,

default=uuid.uuid4,

editable=False)

title = models.CharField(max\_length=200)

author = models.CharField(max\_length=200)

price = models.DecimalField(max\_digits=6, decimal\_places=2)

cover = models.ImageField(upload\_to="covers/", blank=True)

class Meta: # new

Permissions = [

("special\_status", "Can read all books"),

]

def \_\_str\_\_(self):

return self.title

def get\_absolute\_url(self):

return reverse("book\_detail", args=[self.id])

…

|  |
| --- |
| The order of the inner classes and methods here is deliberate. It follows the Model style section from the Django documentation. |

Since we have updated our database model we must create a new migrations file and then migrate the database to apply it.

docker-compose exec web python manage.py makemigrations books

Migrations for 'books':

books/migrations/0004\_alter\_book\_options.py

- Change Meta options on book

docker-compose exec web python manage.py migrate

Operations to perform:

Apply all migrations: account, accounts, admin, auth, books, contenttypes, sessions, si\

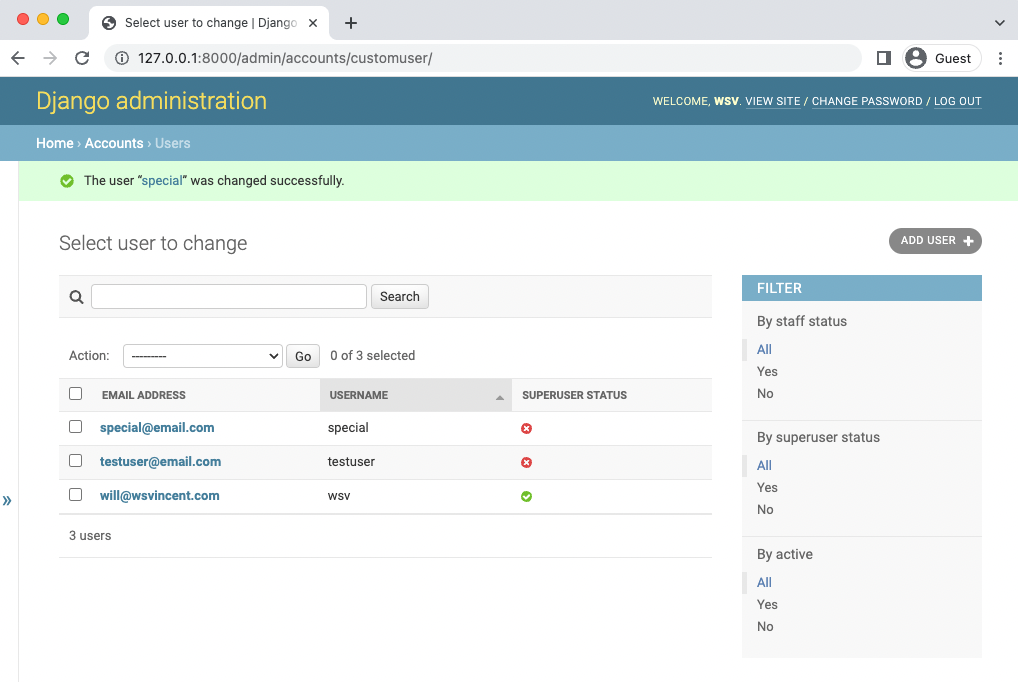
tes

Running migrations:

Applying books.0004\_alter\_book\_options... OK

**User Permissions**

Now we need to apply this custom permission to our new special user. Thanks to the admin this is not a difficult task. Navigate to the Users section where the three existing users are listed: special@email.com, testuser@email.com, and your superuser account (mine has an email address of [will@wsvincent.com](mailto:will@wsvincent.com).)



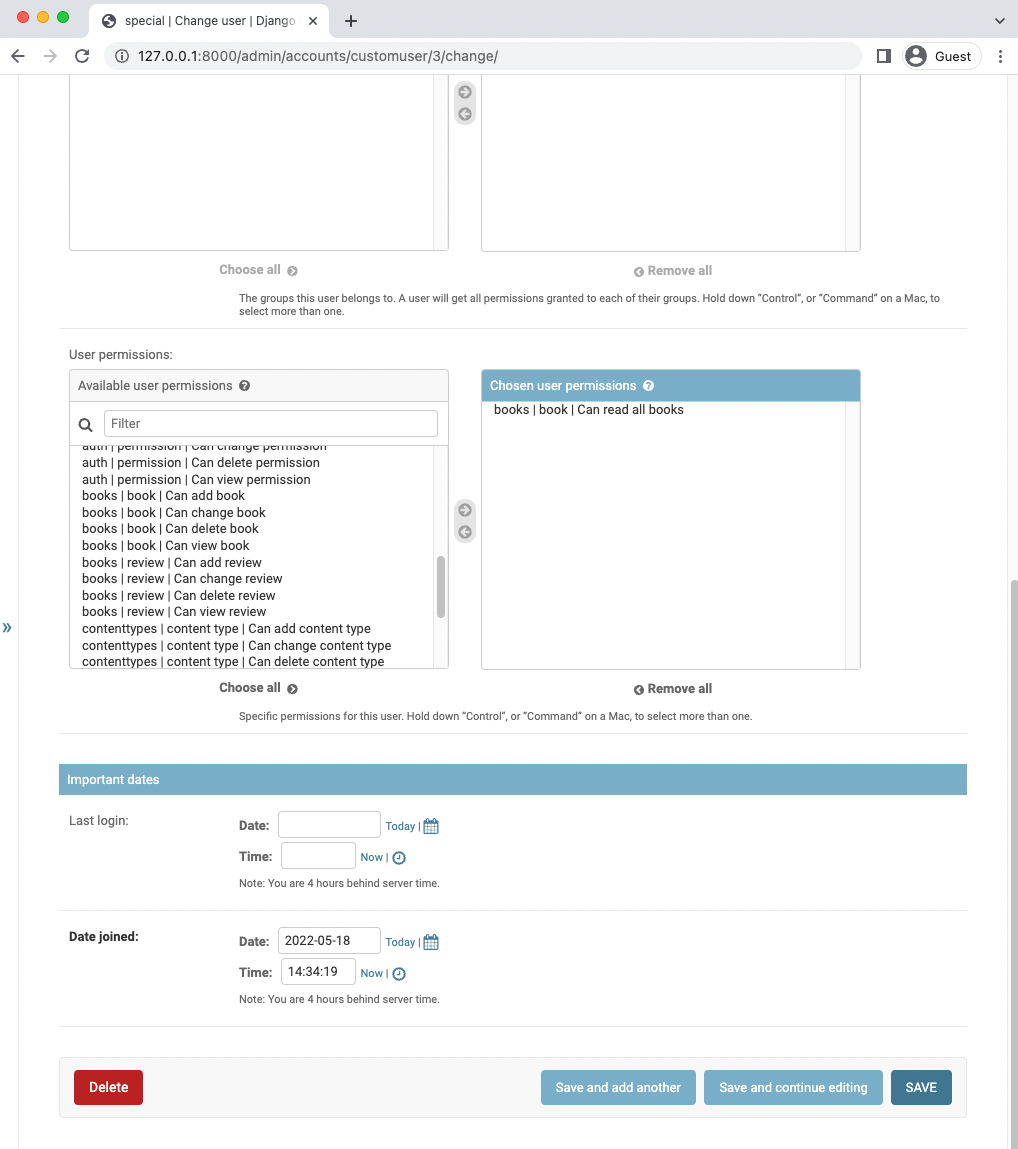
Special User

Click on the special@email.com user and then scroll down to User permissions near the bottom of the page. Within it search for books | book | Can read all books and select it.



Can read all books

Click on the -> arrow to add it to “Chosen user permissions.” Don’t forget to click the “Save” button at the bottom of the page.



Add Permission

**PermissionRequiredMixin**

The last step is to apply the custom permission using the PermissionRequiredMixin. One of the many great features of class-based views is we can implement advanced functionality with very little code on our part and this particular mixin is a good example of that.

We will add PermissionRequiredMixin to our list of imports on the top line. Then add it to DetailView after LoginRequiredMixin but before DetailView. The order should make sense: if a user isn’t already logged in it makes no sense to do the additional check of whether they have permission. Finally add a permission\_required field which specifies the desired permission. In our case its name is special\_status and it exists on the books model.

# books/views.py

from django.contrib.auth.mixins import (

LoginRequiredMixin,

PermissionRequiredMixin

)

from django.views.generic import ListView, DetailView

from .models import Book

# Create your views here.

class BookListView(LoginRequiredMixin, ListView):

model = Book

context\_object\_name = "book\_list"

template\_name = "books/book\_list.html"

login\_url = "account\_login"

class BookDetailView(

LoginRequiredMixin,

PermissionRequiredMixin,

DetailView):

model = Book

context\_object\_name = "book"

template\_name = "books/book\_detail.html"

login\_url = "account\_login"

permission\_required = "books.special\_status"

It is possible to add multiple permissions via the permission\_required field, though we are not doing so here.

To try out our work, log out of the admin. This is necessary because the superuser account is used for the admin and by default has access to everything. Not a great user account to test with!

Log in to the Bookstore site using the testuser@email.com account and then navigate to the Books page listing the three available titles. If you then click on any one of the books, you’ll see a “403 Forbidden” error because permission was denied.



**403 Error Page**

Now go back to the homepage at http://127.0.0.1:8000/ and log out. Then log in using the special@email.com account. Navigate again to the Books page and each individual book page is accessible.

**Groups & UserPassesTestMixin**

The third permissions mixin available is UserPassesTestMixin which restricts a view’s access only to users who pass a specific test.

And in large projects Groups, which are Django’s way of applying permissions to a category of users, become prominent. If you look on the Admin homepage there is a dedicated Groups section where they can be added and have permissions set. This is far more efficient than adding permissions on a per-user basis.

An example of groups is if you have a premium section on your website, a user upgrading could switch them into the premium group and then have access to however many specific extra permissions that involves.

**Tests**

It’s a good idea to run tests whenever a code change has been made. After all, the whole point of testing is to check that we did not inadvertently cause another part of the application to fail.

docker-compose exec web python manage.py test

...

Ran 17 tests in 0.249s

FAILED (failures=2)

It turns out we do have some failing tests! Specifically, test\_book\_list\_view and test\_book\_detail\_view both complain of a 302 status code, meaning a redirection, rather than a 200 for success. This is because we’ve just added the requirement that log in is required to view the list of books and for a detail page the user must have a special\_status permission.

The first step is to import Permission near the top of the file from the built-in auth models. Then within our BookTests in books/tests.py add the special\_status permission to the setUpTestData method so it is available for all our tests.

Then we will transform the existing test\_book\_list\_view test into two different tests: one

for logged in users (test\_book\_list\_view\_for\_logged\_in\_user) and one for logged out users (test\_book\_list\_view\_for\_logged\_out\_user). Finally we will update and rename the existing test\_book\_detail\_view to test\_book\_detail\_view\_with\_permissions and have it check if a user has the correct permission.

Here is what the fully updated code looks like:

# books/tests.py

from django.contrib.auth import get\_user\_model

from django.contrib.auth.models import Permission

from django.test import TestCase

from django.urls import reverse

from .models import Book, Review

# Create your tests here.

class BookTests(TestCase):

@classmethod

def setUpTestData(cls):

cls.user = get\_user\_model().objects.create\_user(

username="reviewuser",

email="reviewuser@email.com",

password="testpass123"

)

cls.special\_permission = Permission.objects.get(

codename="special\_status"

)

cls.book = Book.objects.create(

title="Harry Potter",

author="JK Rowling",

price="25.00",

)

cls.review = Review.objects.create( # new

book=cls.book,

author=cls.user,

review="An excellent review",

)

def test\_book\_listing(self):

self.assertEqual(f"{self.book.title}", "Harry Potter")

self.assertEqual(f"{self.book.author}", "JK Rowling")

self.assertEqual(f"{self.book.price}", "25.00")

def test\_book\_list\_view\_for\_logged\_in\_user(self):

self.client.login(email="reviewuser@email.com", password="testpass123")

response = self.client.get(reverse("book\_list"))

self.assertEqual(response.status\_code, 200)

self.assertContains(response, "Harry Potter")

self.assertTemplateUsed(response, "books/book\_list.html")

def test\_book\_list\_view\_for\_logged\_out\_user(self):

self.client.logout()

response = self.client.get(reverse("book\_list"))

self.assertEqual(response.status\_code, 302)

self.assertRedirects(

response, "%s?next=/books/" % (reverse("account\_login")))

response = self.client.get(

"%s?next=/books/" % (reverse("account\_login")))

self.assertContains(response, "Log In")

def test\_book\_detail\_view\_with\_permission(self):

self.client.login(email="reviewuser@email.com", password="testpass123")

self.user.user\_permissions.add(self.special\_permission)

response = self.client.get(self.book.get\_absolute\_url())

no\_response = self.client.get("/books/12345/")

self.assertEqual(response.status\_code, 200)

self.assertEqual(no\_response.status\_code, 404)

self.assertContains(response, "Harry Potter")

self.assertContains(response, "An excellent review")

self.assertTemplateUsed(response, "books/book\_detail.html")

**Git**

Make sure to create a new Git commit for the changes in this chapter.

Bring docker down on chapter - docker-compose down

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter 14. Permissions”

As always you can compare your code again the official source code on Github.

**Conclusion**

Permissions and groups are a highly subjective topic that can vary widely from project to project. The basic approach, however, remains the same and mimic what we’ve covered here: start by restricting access and only gradually loosed it to users as needed. In our case we restricted access to only logged in users and from there added additional permissions around what groups of logged in users could view.

In the next chapter we’ll add search functionality to our Bookstore site.

The end.