**Chapter 13: File & Image Uploads**

We previously configured static assets such as images in Chapter 6, but user-uploaded files, such as book covers, are somewhat different. To start with, Django refers to the former as static whereas anything uploaded by a user, whether it be a file or an image, is referred to as media.

The process for adding this feature for files or images is similar, but for images the Python image processing library Pillow must be installed which includes additional features such as basic validation.

Let’s install pillow using our by-now-familiar pattern of adding it to the requirements.txt file.

requirements.txt

asgiref==3.5.2

Django==4.0.4

psycopg2-binary==2.9.3

sqlparse==0.4.2

django-crispy-forms==1.14.0

crispy-bootstrap5==0.6

django-allauth==0.50.0

environs[django]==9.5.0

pillow==9.0.1

Then stop our Docker container, rebuild the Docker image so it now contains pillow, and start up the container up again.

docker-compose down

docker-compose up -d –build

**Media Files**

Fundamentally the difference between static and media files is that we can trust the former, but we definitely can’t trust the latter by default. There are always security concerns when dealing with user-uploaded content. Notably, it’s important to validate all uploaded files to ensure they are what they say they are. There are a number of nasty ways a malicious actor can attack a website that blindly accepts user uploads.

To start let’s add two new configurations to the django\_project/settings.py file. By default MEDIA\_URL and MEDIA\_ROOT are both empty and not displayed so we need to configure them:

• MEDIA\_ROOT is the absolute file system path to the directory for user-uploaded files

• MEDIA\_URL is the URL we can use in our templates for the files

We can add both of these settings after STATICFILES\_STORAGE near the bottom of the django\_project/settings.py file. We’ll use the common convention of calling both media. Don’t forget to include the trailing slash / for MEDIA\_URL!

# django\_project/settings.py

MEDIA\_URL = "/media/" # new

MEDIA\_ROOT = BASE\_DIR / "media" # new

Next add a new directory called media and a subdirectory called covers within it.

mkdir media

mkdir media/covers

And finally, since user-uploaded content is assumed to exist in a production context, to see media items locally we need to update django\_project/urls.py to show the files locally. This involves importing both settings and static at the top and then adding an additional line at the bottom.

# django\_project/urls.py

from django.conf import settings # new

from django.conf.urls.static import static # new

from django.contrib import admin

from django.urls import path, include

urlpatterns = [

# Django admin

path('admin/', admin.site.urls),

# User management

path("accounts/", include("allauth.urls")),

# Local apps

path("", include("pages.urls")),

path("books/", include("books.urls")),

] + static(

settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT

) # new

**Models**

With our generic media configuration out of the way we can now turn to our models. To store these images we’ll use Django’s ImageField which comes with some basic image processing validation included.

The name of the field is cover and we specify the location of the uploaded image will be in MEDIA\_ROOT/covers (the MEDIA\_ROOT part is implied based on our earlier settings.py config).

# 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/") # new

|  |
| --- |
| If we wanted to allow uploads of a regular file rather than an image file the only difference could be to change ImageField to FileField. |

Since we’ve updated the model it’s time to create a migrations file.

docker-compose exec web python manage.py makemigrations books

You are trying to add a non-nullable field 'cover\_image' to book

without a default; we can't do that (the database needs something to populate

existing rows).

Please select a fix:

1) Provide a one-off default now (will be set on all existing rows with a

null value for this column)

2) Quit, and let me add a default in models.py

Select an option:

Oops! What happened? We’re adding a new database field, but we already have three entries in our database for each book. Yet we failed to set a default value for cover.

To fix this type 2 to quit. We’ll add a blank field set to True for existing images.

# books/models.py

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

|  |
| --- |
| It’s common to see blank and null used together to set a default value on a field. A gotcha is that the field type – ImageField vs. CharField and so on – dictates how to use them properly so closely read the documentation for future use. |

Now we can create a migrations file without errors.

docker-compose exec web python manage.py makemigrations books

Migrations for 'books':

books/migrations/0003\_book\_cover.py

- Add field cover to book

And then apply the migration to our database.

docker-compose exec web python manage.py migrate

Operations to perform:

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

tes

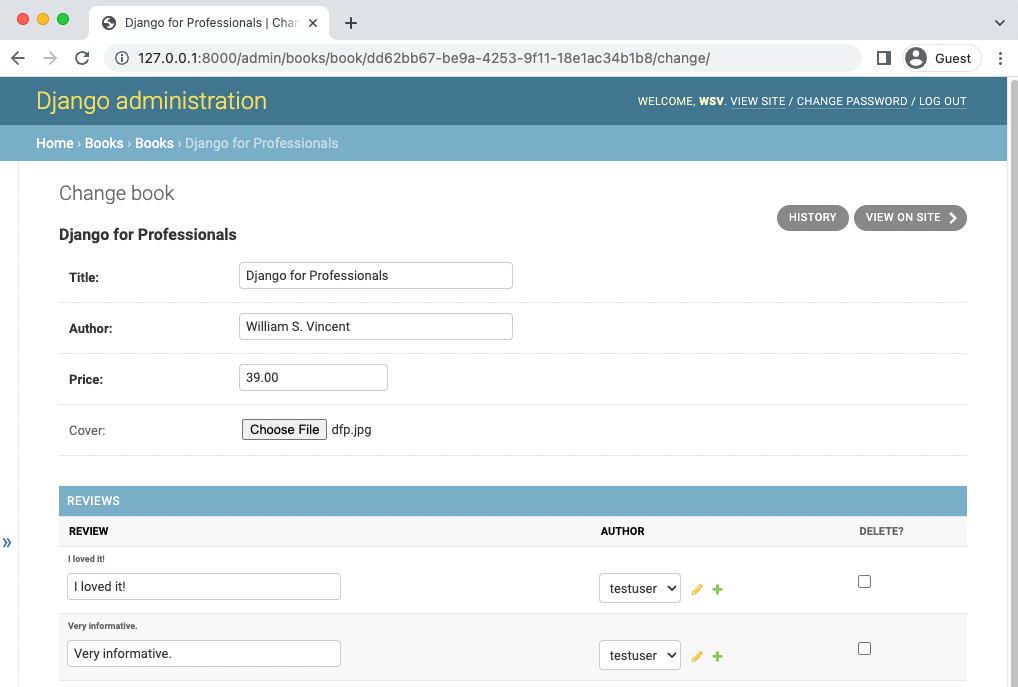
Running migrations:

Applying books.0003\_book\_cover... OK

**Admin**

We’re in the home stretch now! Navigate over to the admin and the entry for the book “Django for Professionals.” The cover field is visible already and we already have a copy of it locally within static/images/cover\_40.png so use that file for the upload and then click the “Save” button in bottom right.

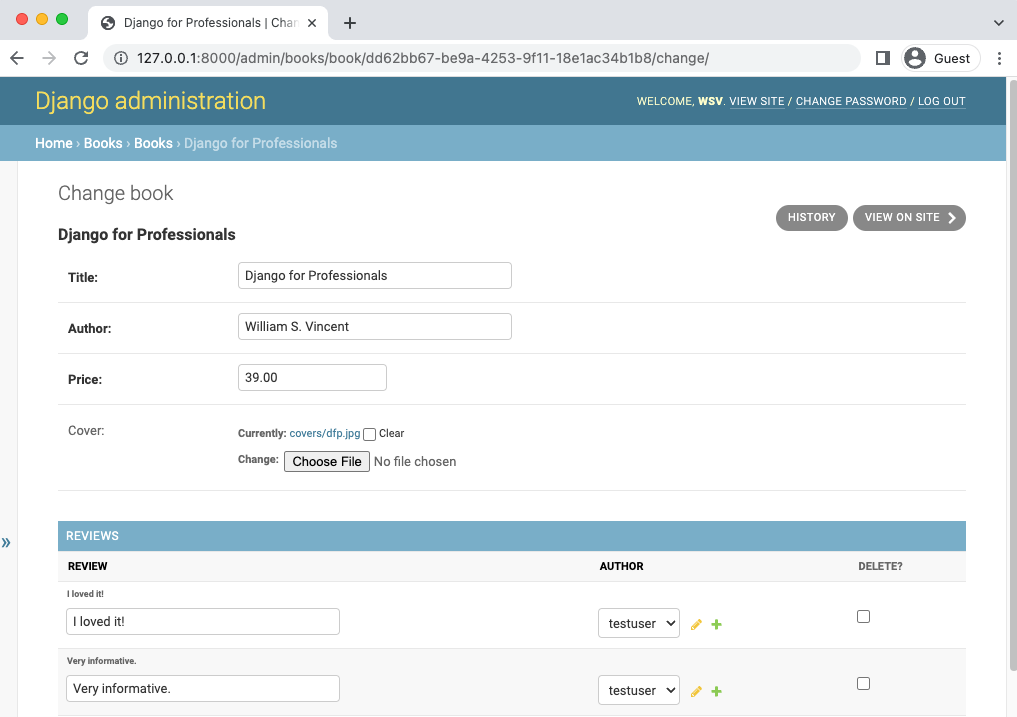
Note: I don’t have the same path, so use yours if it work.



Admin add cover

This will redirect back to the main Books section. Click on the link again for “Django for

Professionals” and we can see it currently exists in our desired location of covers/.



Admin with cover

**Template**

OK, final step. Let’s update our template to display the book cover on the individual page. The route will be book.cover.url pointing to the location of the cover in our file system.

Here’s what the updated book\_detail.html file looks like with this one line change above the title.

# templates/books/book\_detail.html

{% extends "\_base.html" %}

{% block title %}{{ book.title }}{% endblock title %}

{% block content %}

<div class="book\_detail">

<img class="bookcover" src="{{ book.cover.url}}" alt="{{ book.title }}">

<h2><a href="">{{ book.title }}</a></h2>

<p>Author: {{ book.author }}</p>

<p>Price: {{ book.price }}</p>

<div>

<h3>Reviews</h3>

<ul>

{% for review in book.reviews.all %}

<li>{{ review.review }} ({{ review.author }})</li>

{% endfor %}

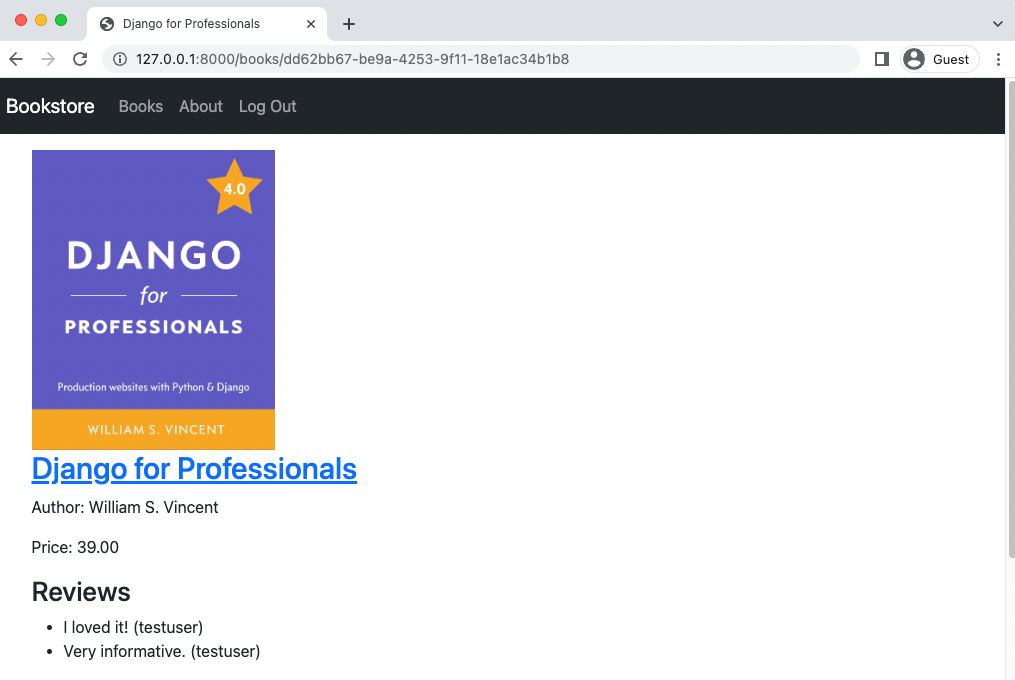
</ul>

</div>

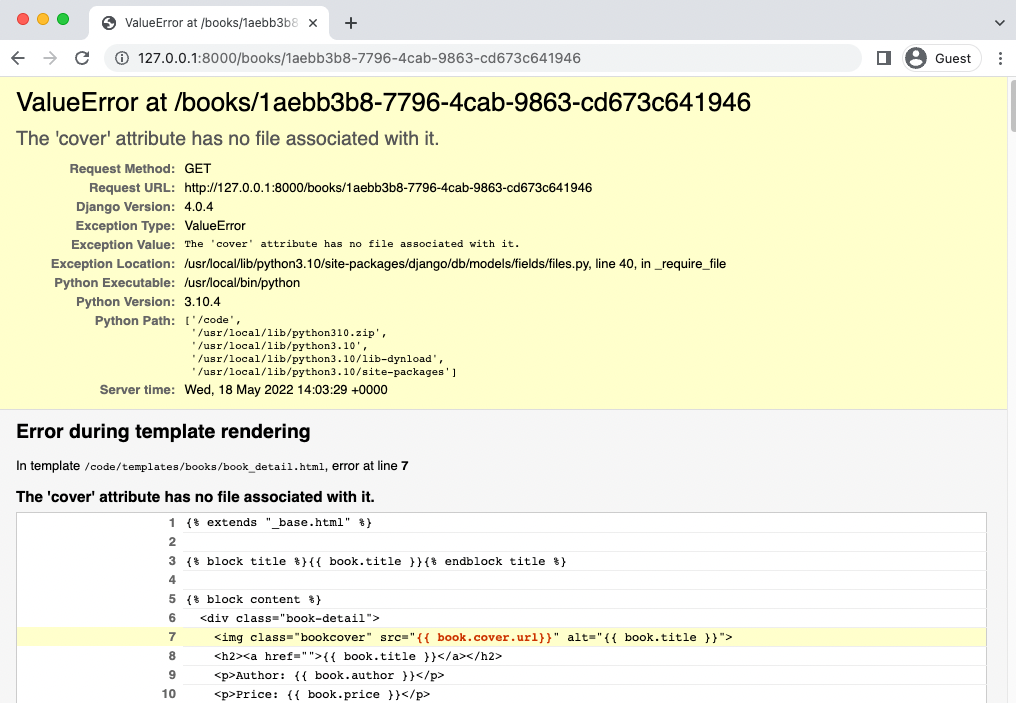
</div>

{% endblock content %}

If you now visit the page for “Django for Professionals” you’ll see the cover image proudly there!



**Cover image**

One potential gotcha is that our template now expects a cover to be present. If you navigate to either of the two other books, for which we have not added a cover, you’ll see the following ValueError message which is quite descriptive. 

We must add some basic logic to our template so that if a cover is not present the template doesn’t look for it! This can be done using an if statement that checks for book.cover and displays it if it exists.

# templates/books/book\_detail.html

{% extends "\_base.html" %}

{% block title %}{{ book.title }}{% endblock title %}

{% block content %}

<div class="book\_detail">

{% if book.cover %}

<img class="bookcover" src="{{ book.cover.url}}" alt="{{ book.title }}">

{% endif %}

<h2><a href="">{{ book.title }}</a></h2>

…

If you refresh either book page now you’ll see they display the correct page albeit without a cover.



DFA No Error

**django-storages**

There are several steps that a truly production website could take but are beyond the current scope of this book. The most important is storing all media files on a dedicated CDN (Content Delivery Network) rather than on our own server. Unlike static files, which the developer controls and can trust, media files are user-generated and should always be treated with caution. The popular third-party package django-storages allows for storing Django media files on a service like Amazon’s S3. (I will use Supabase instead is free)

Furthermore the hosting service we will be using later on, Render (it’s free), has an ephemeral file system. Each internal dyno boots with a clean copy of the file system from the most recent deploy. Static files are located on the file system; media files are not. As a result, in production media files will not remain with Render. Using django-storages is therefore basically mandatory alongside Render and will be mentioned again in the deployment chapter.

**Next Steps**

Additional steps could include extra validations on the image-uploading form to ensure that only a normal, safe image was able to be added. We could add dedicated create/edit/delete forms for the creation of books and cover image. Tests would be nice to have here too although they would be primarily focused on the form validation section not the basic image-uploading via the admin. Again this is an area that can become quite complex, but is worthy of further study.

The last recommendation is to look at the third-party django-cleanup package which automatically deletes old files. It can be quite handy.

**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 13. File & Image Uploads”

And finally back up chapter:

Copy-Item -Recurse -Path "C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter 13. File & Image Uploads" -Destination "C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter 13. File & Image Uploads - Backup"

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

**Conclusion**

This chapter demonstrated how to add user files to a project. In practice it is straightforward, but the additional layer of security concerns makes it an area worthy of focus at scale.

In the next chapter we will add permissions to our site to lock it down.

The end.