**Chapter 12: Reviews App**

In this chapter we’ll add a reviews app so that readers can leave reviews of their favorite books. It gives us a chance to discuss foreign keys, app structure, and dive into forms.

**Foreign Keys**

We’ve already used a foreign key with our user model, but didn’t have to think about it. Now we do! Fundamentally a database table can be thought of as similar to a spreadsheet with rows and columns. There needs to be a primary key field that is unique and refers to each record. In the last chapter we changed that from id to a UUID, but one still exists!

This matters when we want to link two tables together. For example, our Books model will link to a Reviews model since each review has to be connected to a relevant book. This implies a foreign key relationship.

There are three possible types of foreign key relationships:

• One-to-one

• One-to-many

• Many-to-many

A one-to-one relationship is the simplest kind. An example would be a table of people’s names and a table of social security numbers. Each person has only one social security number and each social security number is linked to only one person.

In practice one-to-one relationships are rare. It’s unusual for both sides of a relationship to only be matched to one counterpart. Some other examples though would be country-flag or personpassport.

A one-to-many relationship is more common and is the default foreign key setting within Django. For example, consider a restaurant where one customer can place many orders.

It’s also possible to have a ManyToManyField relationship. Let’s consider a list of books and a list of authors: each book could have more than one author and each author can write more than one book. That’s a many-to-many relationship. Just as with the previous two examples you need a linked Foreign Key field to connect the two lists. Additional examples include doctors and patients (every doctor sees multiple patients and vice versa) or employees and tasks (each employee has multiple tasks while each task is worked on by multiple employees).

Database design is a fascinating, deep topic that is both an art and a science. As the number of tables grow in a project over time it is almost inevitable that a refactoring will need to occur to address issues around inefficiency, bloat, and outright errors. Normalization is the process of structuring a relational database though far beyond the scope of this book.

**Reviews model**

Coming back to our basic reviews app, the first consideration is what type of foreign key relationship will there be. If we are going to link a user to a review, then it is a straightforward one-to-many relationship. However it could also be possible to link books to reviews which would be many-to-many. The “correct” choice quickly becomes somewhat subjective and certainly dependent upon the particular needs of the project.

In this project we’ll treat the reviews app as a one-to-many between authors and reviews as it’s the simpler approach.

Here again we face a choice around how to design our project. Do we add the Reviews model within our existing books/models.py file or create a dedicated reviews app that we then link to?

Let’s start by adding a Reviews model to the books app.

# books/models.py

import uuid

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

from django.db import models

from django.urls import reverse

class Book(models.Model):

…

class Review(models.Model): # new

book = models.ForeignKey(

Book,

on\_delete=models.CASCADE,

related\_name="reviews",

)

review = models.CharField(max\_length=255)

author = models.ForeignKey(

get\_user\_model(),

on\_delete=models.CASCADE,

)

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

return self.review

At the top, under imports include get\_user\_model, which is needed to refer to our CustomUser model, then create a dedicated Review model. The book field is the one-to-many foreign key that links Book to Review and we’re following the standard practice of naming it the same as the linked model. The review field contains the actual content which perhaps could be a TextField depending on how much space you want to provide for review length! For now, we’ll force reviews to be short at 255 characters or less. And then we’ll also link to the author field to auto-populate the current user with the review.

For all many-to-one relationships such as a ForeignKey we must also specify an on\_delete option. We also explicitly set the related\_name to make it easier to follow the foreign key relationship “backwards” in the future on queries. Note that this name must be unique to avoid future problems. And lastly using get\_user\_model to reference our custom user model.

Create a new migrations file for our changes and then run migrate to apply them.

docker-compose exec web python manage.py makemigrations books

Migrations for 'books':

books/migrations/0002\_review.py

- Create model Review

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.0002\_review... OK

**Admin**

For the reviews app to appear in the admin we need to update books/admin.py substantially by adding the Review model and specifying a display of TabularInline.

# books/admin.py

from django.contrib import admin

from .models import Book, Review

class ReviewInline(admin.TabularInline):

model = Review

class BookAdmin(admin.ModelAdmin):

inlines = [

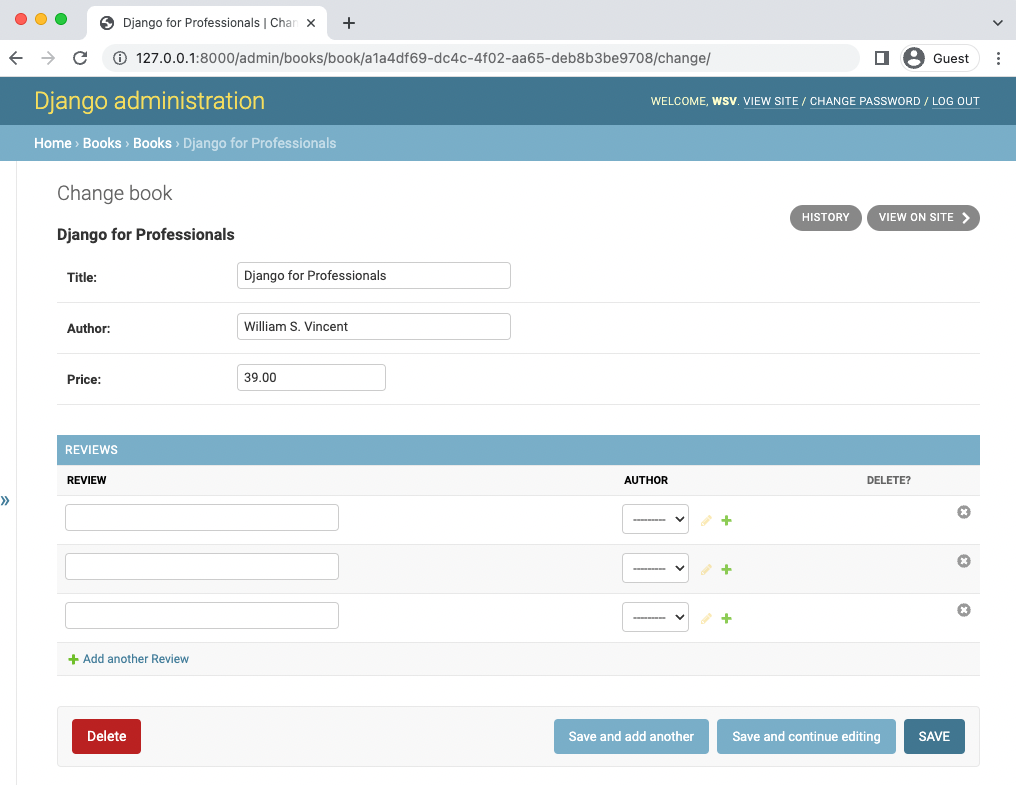
ReviewInline,

]

list\_display = ("title", "author", "price",)

admin.site.register(Book, BookAdmin)

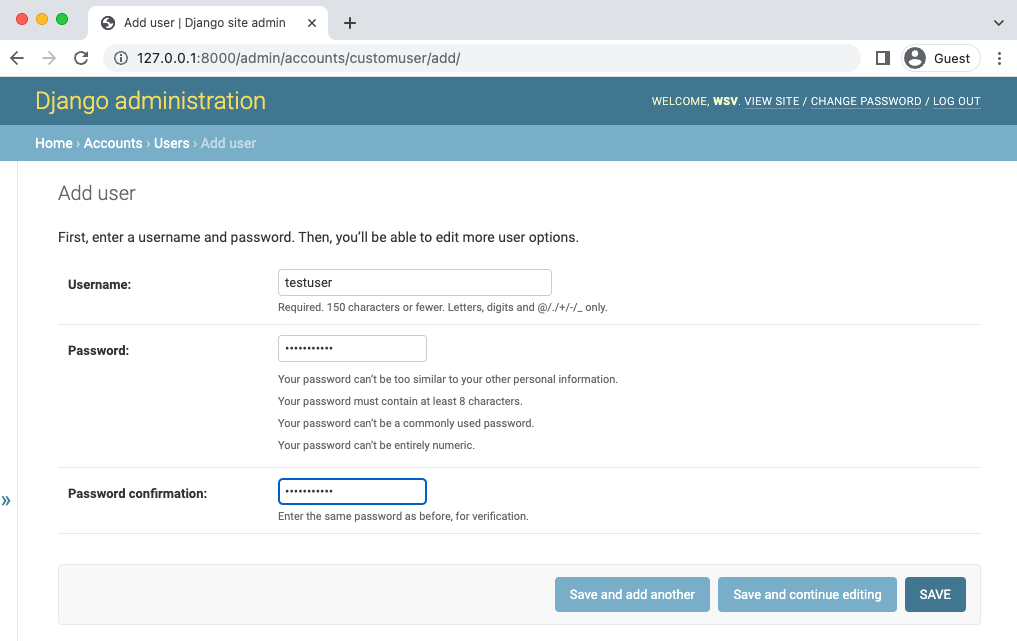
Now navigate to the books section at http://127.0.0.1:8000/admin/books/book/ and then click on any of the books to see the reviews visible on the individual book page.



Django for Professionals Admin Reviews

We’re limited to reviews by existing users at this point, although we have previously created a testuser@email.com that was deleted when we removed the database volume mount in the previous chapter. There are two options for adding this account: we could go to the main site and use the “Sign Up” link or we can add it directly from the admin. Let’s do the latter.

From the Users section on the Admin homepage click on the “+ Add” button. Add a new user called testuser and a password. Click the “Save” button.



**Admin testuser**

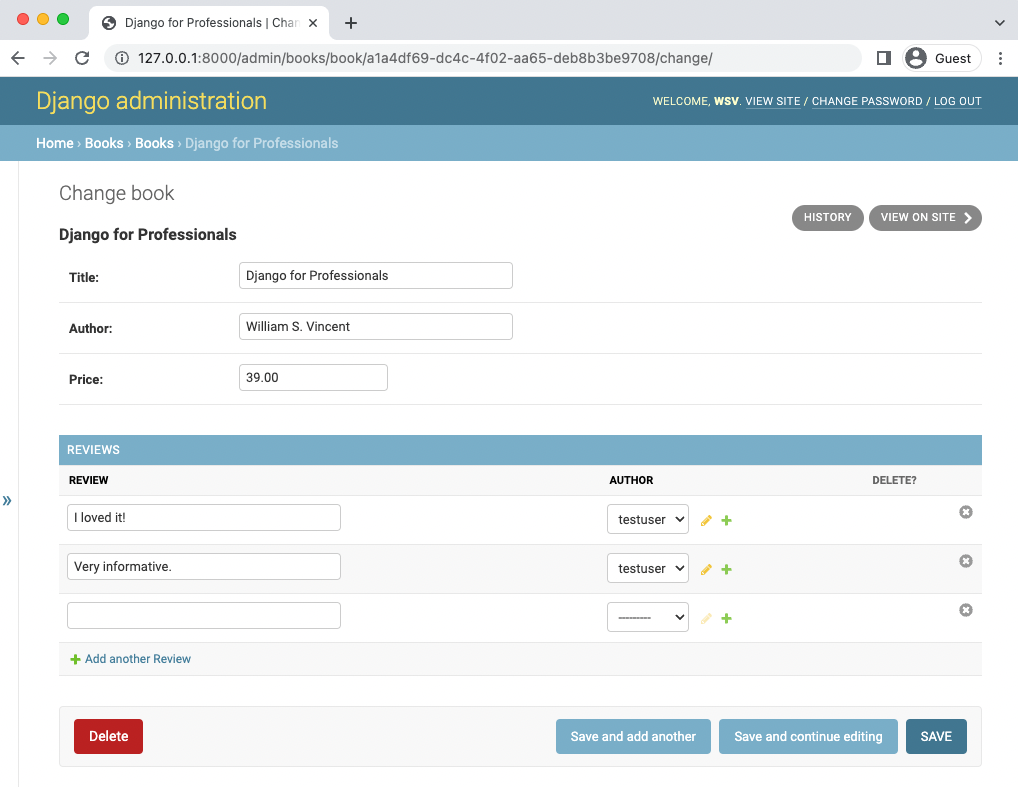
Then on the next page add testuser@email.com as the email address. Did you notice the the user’s password is encrypted now? Django does not store raw passwords which means even as a superuser we cannot see individual user passwords. We can change the password to something else but we can’t just copy and paste user passwords.

Scroll down to the bottom of the page and click the “Save” button.



Admin testuser

Ok, finally, we can add reviews to the “Django for Professionals” book using testuser. Navigate back to the Books section and click on the correct book. Write two reviews and as AUTHOR make sure to select testuser.



Add Two Reviews

**Templates**

With the reviews model set it is time to update our templates to display reviews on the individual page for each book. Add a basic “Reviews” section and then loop over all existing reviews. Since this is a foreign key relationship we follow it by using book.reviews.all. Then display the review field with review.review and the author with review.author.

# templates/books/book\_detail.html

{% extends "\_base.html" %}

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

{% block content %}

<div class="book\_detail">

<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 %}

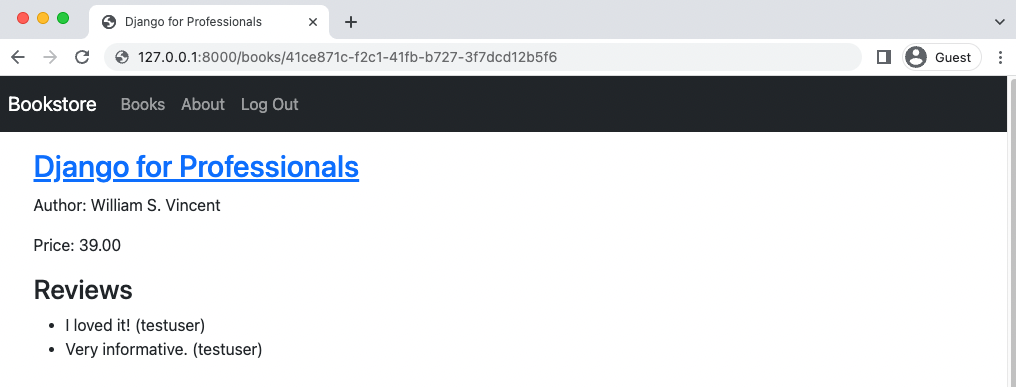
</ul>

</div>

</div>

{% endblock content %}

That’s it! Navigate over to the “Django for Professionals” individual page to see the result. Your url will be different than the one here because we’re using a UUID.



Reviews on Detail Page

**Tests**

Time for tests. We need to create a new user for our review and add a review to the setUpTestData method in our test suite. Then we can test that the book object contains

the correct review.

This involves importing get\_user\_model as well as adding the Review model at the top. We can use create\_user to make a new user called reviewuser and then a review object that is linked to our single book object. Finally under test\_book\_detail\_view we can add an additional assertContains test to the response object.

# books/tests.py

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

from django.test import TestCase

from django.urls import reverse

from .models import Book, Review # new

# Create your tests here.

class BookTests(TestCase):

@classmethod

def setUpTestData(cls):

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

username="reviewuser",

email="reviewuser@email.com",

password="testpass123"

)

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\_abook\_list\_view(self):

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\_detail\_view(self):

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") # new

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

If you run the tests now they all should pass.

docker-compose exec web python manage.py test

Creating test database for alias 'default'...

System check identified no issues (0 silenced).

.................

----------------------------------------------------------------------

Ran 17 tests in 0.260s

OK

Destroying test database for alias 'default'...

**Git**

Add our new code changes to Git and include a commit message for the chapter.

docker-compose down

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter 12. Reviews App”

The code for this chapter can be found on the official Github repository.

**Conclusion**

With more time we might update the reviews functionality with a form on the page itself, however this means AJAX calls using either htmx, jQuery, React, Vue, or another dedicated JavaScript framework. Unfortunately covering that fully is well beyond the scope of this book.

As the project grows it might also make sense to split reviews off into its own dedicated app. Doing so is a very subjective call. In general, keeping things as simple as possible–adding foreign keys within an existing app until it becomes too large to easily understand–is a solid approach.

In the next chapter we will add image uploads to our site so there can be covers for each book

The end.