**Chapter 11: Books App**

In this chapter we will build a Books app for our project that displays all available books and has an individual page for each. We’ll also explore different URL approaches starting with using an id, then switching to a slug, and finally using a UUID.

To start, we must create this new app which we’ll call books.

docker-compose exec web python manage.py startapp books

And to ensure Django knows about our new app, open your text editor and add the new app to INSTALLED\_APPS in our django\_project/settings.py file:

# django\_project/settings.py

INSTALLED\_APPS = [

'django.contrib.admin',

'django.contrib.auth',

'django.contrib.contenttypes',

'django.contrib.sessions',

'django.contrib.messages',

'django.contrib.staticfiles',

'django.contrib.sites',

# Third-party

'crispy\_forms', # new

'crispy\_bootstrap5', # new

'allauth',

'allauth.account',

# Local

'accounts.apps.AccountsConfig', # new

'pages.apps.PagesConfig', # new

'books.apps.BooksConfig', # new

]

Ok, initial creation complete!

**Models**

Ultimately we’ll need a model, view, url, and template for each page so it’s common to debate where to start. The model is a good place to start as it sets the structure. Let’s think about what fields we might want to include. To keep things simple we’ll start with a title, author, and price.

Update the books/models.py file to include our new Books model.

# books/models.py

from django.db import models

# Create your models here.

class Book(models.Model):

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

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

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

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

return self.title

At the top we’re importing the Django class models and then creating a Book model that subclasses it which means we automatically have access to everything within django.db.models.Model and can add additional fields and methods as desired.

For title and author we’re limiting the length to 200 characters and for price using a

DecimalField which is a good choice when dealing with currency.

Below we’ve specified a \_\_str\_\_ method to control how the object is outputted in the Admin and Django shell.

Now that our new database model is created we need to create a new migration record for it.

docker-compose exec web python manage.py makemigrations

Migrations for 'books':

books/migrations/0001\_initial.py

- Create model 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.0001\_initial... OK

Our database is configured. Let’s add some data to the admin.

**Admin**

We need a way to access our data for which the Django admin is perfectly suited. Don’t forget to update the books/admin.py file or else the app won’t appear! I forget this step almost every time even after using Django for years.

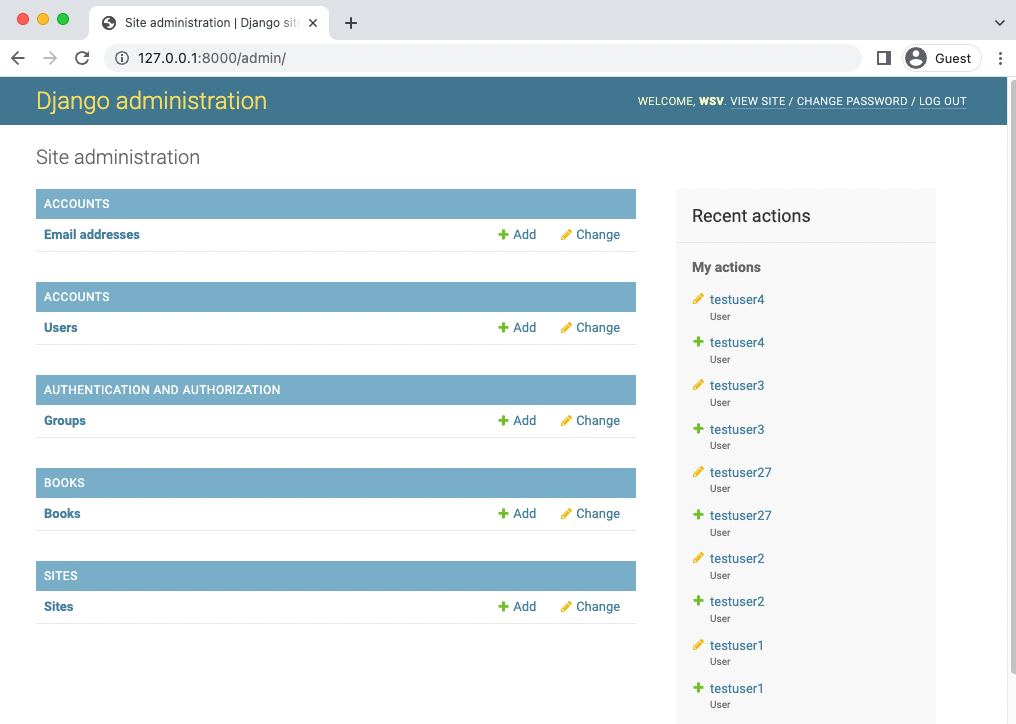
# books/admin.py

from django.contrib import admin

from .models import Book

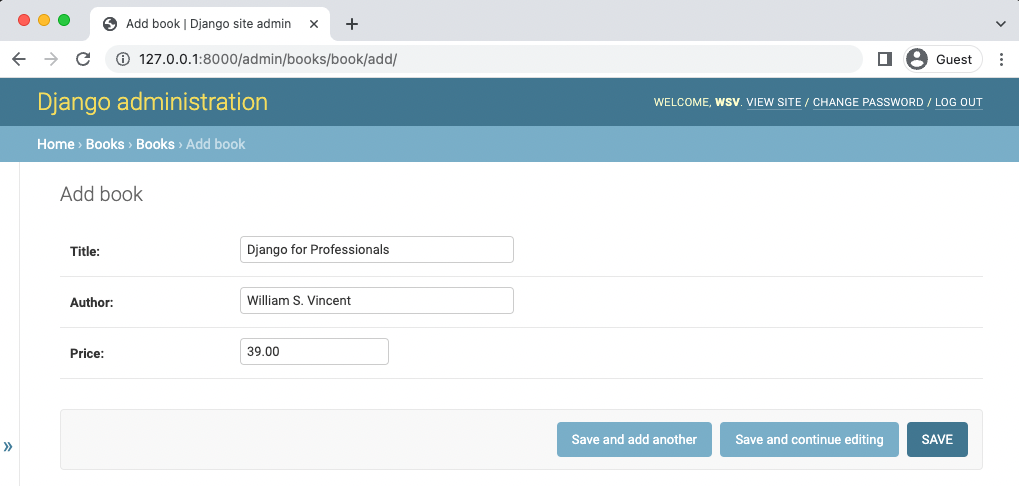
admin.site.register(Book)

If you look into the admin at http://127.0.0.1:8000/admin/ the Books app is now there.



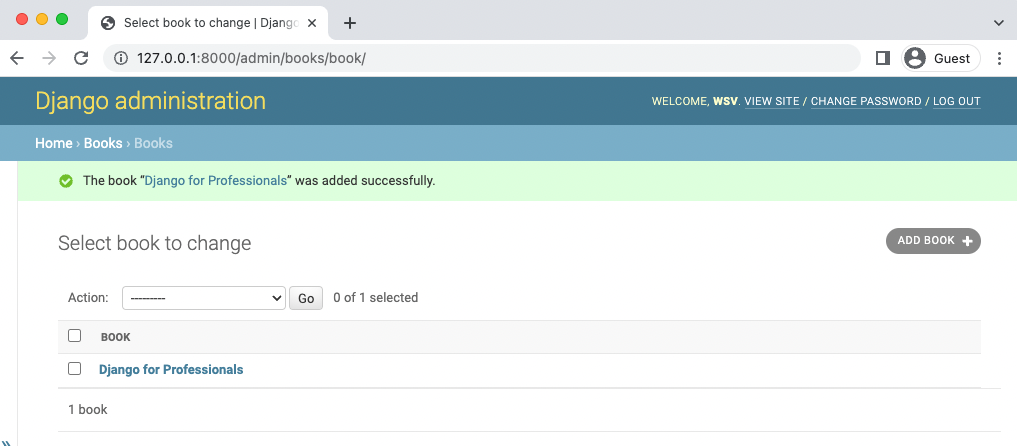
Admin Homepage

Let’s add a book entry for Django for Professionals. Click on the + Add button next to Books to create a new entry. The title is “Django for Professionals”, the author is “William S. Vincent”, and the price is $39.00. There’s no need to include the dollar sign $ in the amount as we’ll add that in our eventual template.



Admin - Django for Professionals book

After clicking on the “Save” button it redirects to the main Books page which only shows the title.



Admin Books Page

Let’s update the books/admin.py file to specify which fields we also want displayed.

# books/admin.py

from django.contrib import admin

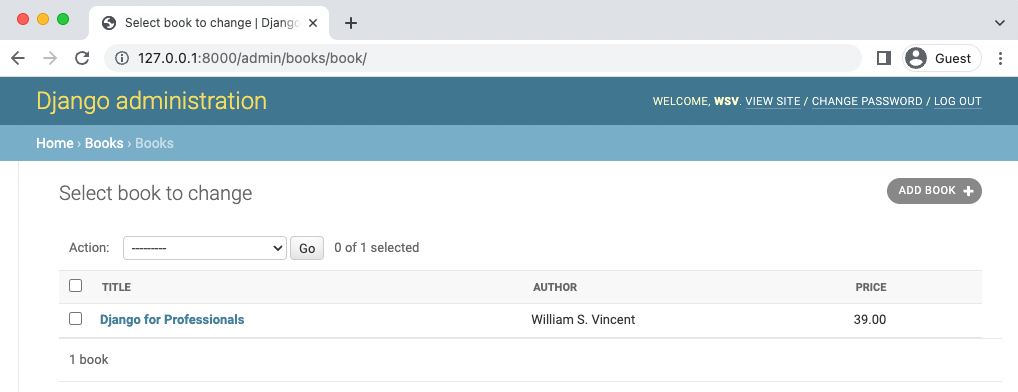
from .models import Book

class BookAdmin(admin.ModelAdmin):

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

admin.site.register(Book, BookAdmin)

Then refresh the page.



Admin Books List Page

Now that our database model is complete we need to create the necessary views, URLs, and templates so we can display the information on our web application. Where to start is always a question and a confusing one at that for developers.

Personally I often start with the URLs, then the Views, and the Templates.

URLs

We need to update two urls.py files. The first one is django\_project/urls.py. Add the new path for the books app.

from django.contrib import admin

from django.urls import path, include # new

# django\_project/urls.py

urlpatterns = [

# Django admin

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

# User management

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

# Local apps

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

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

]

Now create a books/urls.py file in your text editor for our books app URLs paths. We’ll use the empty string, "" so all books app URLs will start at books/ based on the URL path just set in django\_project/urls.py. The view it references, BookListView, has yet to be created and it will have the URL name of book\_list.

# books/urls.py

from django.urls import path

from .views import BookListView

urlpatterns = [

path("", BookListView.as\_view(), name="book\_list"),

]

**Views**

Moving along it is time for that BookListView we just referenced in our URLs file. This will rely on the built-in ListView, a Generic Class-Based View provided for common use cases like this. All we must do is specify the proper model and template to be used.

# books/views.py

from django.views.generic import ListView

from .models import Book

class BookListView(ListView):

model = Book

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

Note the template book\_list.html does not exist yet.

**Templates**

It is optional to create an app specific folder within our base-level templates directory but it can help especially as number grows in size so we’ll create one called books.

mkdir templates/books/

In your text editor create a new file called templates/books/book\_list.html.

<!-- templates/books/book\_list.html -->

{% extends "\_base.html" %}

{% block title %}Books{% endblock title %}

{% block content %}

{% for book in object\_list %}

<div>

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

</div>

{% endfor %}

{% endblock content %}

At the top we note that this template extends \_base.html and then wraps our desired code with content blocks. We use the Django Templating Language to set up a simple for loop for each book. Note that object\_list comes from ListView and contains all the objects in our view.

The final step is to spin up and then down our containers to reload the Django django\_-

project/settings.py file. Otherwise it won’t realize we’ve made a change and so there will be an error page and in the logs a message about “ModuleNotFoundError: No module named ‘books.urls’”.

Spin down and then up again our containers.

docker-compose down

docker-compose up -d

In your web browser go to http://127.0.0.1:8000/books/ and the new books page is visible.



Books Page

**object\_list**

ListView relies on object\_list, as we just saw, but this is far from descriptive. A better approach is to rename it to a friendlier name using context\_object\_name.

Update books/views.py as follows.

# books/views.py

from django.views.generic import ListView

from .models import Book

# Create your views here.

class BookListView(ListView):

model = Book

context\_object\_name = "book\_list" # new

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

And then swap out object\_list in our template for book\_list.

<!-- templates/books/book\_list.html -->

{% extends "\_base.html" %}

{% block title %}Books{% endblock title %}

{% block content %}

{% for book in book\_list %} – (we swap object\_ to book\_list)

<div>

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

</div>

{% endfor %}

{% endblock content %}

Refresh the page and it will still work as before! This technique is especially helpful on larger projects where multiple developers are working on a project. It’s hard for a front-end engineer to guess correctly what object\_list means!

To prove the list view works for multiple items add two more books to the site via the admin. I’ve added my two other Django books–Django for APIs and Django for Beginners–which both have “William S. Vincent” as the author and “39.00” as the price.



Three Books

**Individual Book Page**

Now we can add individual pages for each book by using another Generic

Class-Based View called DetailView.

Our process is similar to the Books page and starts with the URL importing BookDetailView on the second line and then setting the path to be the primary key of each book which will be represented as an integer <int:pk>.

# books/urls.py

from django.urls import path

from .views import BookListView, BookDetailView # new

urlpatterns = [

path("", BookListView.as\_view(), name="book\_list"),

path("<int:pk>/", BookDetailView.as\_view(), name="book\_detail"), # new

]

Django automatically adds an auto-incrementing primary key to our database models. So while we only declared the fields title, author, and body on our Book model, under-the-hood Django also added another field called id, which is our primary key. We can access it as either id or pk. The pk for our first book is 1. For the second one it will 2. And so on. Therefore when we go to the individual entry page for our first book, we can expect that its URL route will be books/1.

Now on to the books/views.py file where we’ll import DetailView and create a BookDetailView class that also specifies model and template\_name fields.

# books/views.py

from django.views.generic import ListView, DetailView # new

from .models import Book

# Create your views here.

class BookListView(ListView):

model = Book

context\_object\_name = "book\_list"

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

class BookDetailView(DetailView): # new

model = Book

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

And finally create the new template file templates/books/book\_detail.html. It will display all the current fields. We can also showcase the title in the title tags so that it appears in the web browser tab.

<!-- templates/books/book\_detail.html -->

{% extends "-base.html" %}

{% block title %}{{ object.title }}{% endblock title %}

{% block content %}

<div class="book\_detail">

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

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

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

</div>

{% endblock content %}

In your web browser navigate to http://127.0.0.1:8000/books/1/ and you’ll see a dedicated page for our first book.



Book Detail Page

**context\_object\_name**

Just as ListView defaults to object\_list which we updated to be more specific, so too

DetailView defaults to object which we can make more descriptive using context\_object\_name. We’ll set it to book.

# books/views.py

...

class BookDetailView(DetailView):

model = Book

context\_object\_name = "book" # new

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

Don’t forget to update our template too with this change, swapping out object for book for our three fields.

<!-- templates/books/book\_detail.html -->

{% extends "\_base.html" %}

{% block title %}{{ object.title }}{% endblock title %}

{% block content %}

<div class="book\_detail">

<h2><a href="">{{ book.title }}</a></h2> - swap objrect to book -

<p>Author: {{ book.author }}</p> - swap objrect to book -

<p>Price: {{ book.price }}</p> - swap objrect to book -

</div>

{% endblock content %}

**Adding URLs**

We want the links on the book list page to point to individual pages. With the url template tag we can point to book\_detail – the URL name set in books/urls.py – and then pass in the pk.

<!-- templates/books/book\_list.html -->

{% extends "\_base.html" %}

{% block title %}Books{% endblock title %}

{% block content %}

{% for book in book\_list %}

<div>

<h2><a href="{% url 'book\_detail' book.pk %}">{{ book.title }}</a></h2> # new

</div>

{% endfor %}

{% endblock content %}

Refresh the book list page at http://127.0.0.1:8000/books/ and links are now all clickable and direct to the correct individual book page.

As a final step, let’s add a navbar link for “Books” so we don’t have to type out the full URL each time. The URL name for our book list view, book\_list, can be used along with the url template tag to do this. Here is what the updated code looks like in templates/\_base.html.

…

<div class="collapse navbar-collapse" id="navbarCollapse">

<ul class="navbar-nav me-auto mb-2 mb-md-0">

<li class="nav-item"> # new

<a class="nav-link" href="{% url 'book\_list' %}">Books</a> # new

</li>

<li class="nav-item">

<a class="nav-link" href="{% url 'about'}">About</a>

</li>

…

Refresh any page on the website and the working “Books” navbar link is now there.



Book Navbar Link

**get\_absolute\_url**

One recommended step we haven’t made yet is to add a get\_absolute\_url() method which sets a canonical URL for the model. It is also required when using the reverse() function.

Here’s how to add it to our books/models.py file. Import reverse at the top. Then add the get\_absolute\_url method which will be the reverse of our URL name, book\_detail, and passes in the id as a string.

# books/models.py

from django.db import models

from django.urls import reverse # new

# Create your models here.

class Book(models.Model):

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

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

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

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

return self.title

def get\_absolute\_url(self): # new

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

Then we can update the templates. Currently our a href link is using {% url 'book\_detail' book.pk %}. However we can instead use get\_absolute\_url directly which already has the pk passed in.

<!-- templates/books/book\_list.html -->

{% extends "\_base.html" %}

{% block title %}Books{% endblock title %}

{% block content %}

{% for book in book\_list %}

<div>

<h2><a href="{{ book.get\_absolute\_url }}">{{ book.title }}</a></h2> # new

</div>

{% endfor %}

{% endblock content %}

There’s no need to use the url template tag now for the link. Instead there is one canonical reference in the books/models.py file. This is a cleaner approach and should be used whenever you need individual pages for an object.

**Primary Keys vs. IDs**

It can be confusing whether to use a primary key (PK) or an ID in a project, especially since Django’s DetailView treats them interchangeably. However there is a subtle difference.

The id is a model field automatically set by Django internally to auto-increment. So the first book has an id of 1, the second entry of 2, and so on. This is also, by default, treated as the primary key pk of a model.

However it’s possible to manually change what the primary key is for a model. It doesn’t have to be id, but could be something like object\_id depending on the use case. Additionally Python has a built-in id() object which can sometimes cause confusion and/or bugs.

By contrast the primary key pk refers to the primary key field of a model so you’re safer using pk when in doubt. And in fact in the next section we will update the id of our model!

**Slugs vs. UUIDs**

Using the pk field in the URL of our DetailView is quick and easy, but not ideal for a real-world project. The pk is currently the same as our auto-incrementing id. Among other concerns, it tells a potential hacker exactly how many records you have in your database; it tells them exactly what the id is which can be used in a potential attack; and there can be synchronization issues if you have multiple front-ends.

There are two alternative approaches. The first is called a “slug,” a newspaper term for a short label for something that is often used in URLs. For example, in our example of “Django for Professionals” its slug could be django-for-professionals. There’s even a SlugField model field that can be used and either added when creating the title field by hand or auto-populated upon save. The main challenge with slugs is handling duplicates, though this can be solved by adding random strings or numbers to a given slug field. The synchronization issue remains though.

A better approach is to use a UUID (Universally Unique IDentifier) which Django now supports via a dedicated UUIDField.

Let’s implement a UUID now by adding a new field to our model and then updating the URL path.

Import uuid at the top and then update the id field to actually be a UUIDField that is now the primary key. We also use uuid4 for the encryption. This allows us to use DetailView which requires either a slug or pk field; it won’t work with a UUID field without significant modification.

# books/models.py

import uuid # new

from django.db import models

from django.urls import reverse

# Create your models here.

class Book(models.Model):

id = models.UUIDField( # new

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)

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

return self.title

def get\_absolute\_url(self):

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

In the URL path swap out int for uuid in the detail view.

# books/urls.py

from django.urls import path

from .views import BookListView, BookDetailView

urlpatterns = [

path("", BookListView.as\_view(), name="book\_list"),

path("<uuid:pk>/", BookDetailView.as\_view(), name="book\_detail"), # new

]

But now we are faced with a problem: there are existing book entries, three in fact, with their own ids as well as related migration files that use them. Creating a new migration like this causes real problems. The simplest approach, which we will use, is the most destructive: to simply delete old books migrations and start over.

docker-compose exec web rm -r books/migrations

docker-compose down

One last issue is that we are also persisting our PostgreSQL database via a volume mount that still has records to the older id fields. You can see this with the docker volume ls command.

docker volume ls

DRIVER VOLUME NAME

local books\_postgres\_data

The simplest approach is again to simply delete the volume and start over with Docker. As we’re early enough in the project we’ll take this route; a more mature project would require considering a more complex approach.

The steps involve starting up our web and db containers; adding a new initial migration file for the books app, applying all updates with migrate, and then creating a superuser account again.

docker volume rm books\_postgres\_data

docker-compose up -d

docker-compose exec web python manage.py makemigrations books

docker-compose exec web python manage.py migrate

docker-compose exec web python manage.py createsuperuser

Now go into admin and add the three books again. If you then navigate to the main books page and click on an individual book you’ll be taken to a new detail page with a UUID in the URL.



Django for Professionals book UUID

By removing the volume we also lost our various testuser accounts but that’s ok. We can recreate them as needed going forward.

**Tests**

We need to test our model and views now. We want to ensure that the Books model works as expected, including its str representation. And we want to test both ListView and DetailView.

Here’s what sample tests look like in the books/tests.py file.

# books/tests.py

from django.test import TestCase

from django.urls import reverse

from .models import Book

# Create your tests here.

class BookTests(TestCase):

@classmethod

def setUpTestData(cls):

cls.book = Book.objects.create(

title="Harry Potter",

author="JK Rowling",

price="25.00",

)

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

We import TestCase and introduce the setUpTestData method to add a sample book to test. Using setUpTestData often dramatically increases the speed of your tests because the initial data is created once rather than each time for each unit test.

The first unit test, test\_book\_listing, checks that both its string representation and content are correct. Then we use test\_book\_list\_view to confirm that our homepage returns a 200 HTTP status code, contains our body text, and uses the correct books/book\_list.html template.

Finally, test\_book\_detail\_view tests that our detail page works as expected and that an

incorrect page returns a 404. It’s always good both to test that something does exist and that something incorrect doesn’t exist in your tests.

Go ahead and run these tests now. They should all 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 0.208s

OK

Destroying test database for alias 'default'...

**Git**

We’ve done a lot of work in this chapter so add it all to version control now with Git by adding new files and adding a commit message.

docker-compose down

Remove-Item -Recurse -Force .git

git init

git status

git add .

git commit -m “Chapter X. title of chapter”

Create a back up:

Copy-Item -Recurse -Path "C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter X. title of chapter " -Destination "C:\Users\Jean-Marc H\Documents\Django for professionals\Chapter X. title of chapter - Backup"

The official source code for this chapter is available on Github for reference.

**Conclusion**

We’re at the end of quite a long chapter, but the architecture of our Bookstore project is now much clearer. We’ve added a books model, learned how to change the URL structure, and switched to the much more secure UUID pattern.

In the next chapter we’ll learn about foreign key relationships and add a reviews option to our project.

The end.